Renesas RA Family

Driving an LCD Using the External Memory Bus on the RA6M4

Introduction

This application note describes operating the External Memory Bus Interface (ECBIU) on the EK-RA6M4 to drive an externally connected LCD controller. Additionally, the note covers designing a custom graphics application in Azure RTOS GUIX Studio and properly integrating the GUIX Studio design with an e^2 studio application. All graphics topics discussed in this note are demonstrated with the accompanying Hello World application project.

The purpose of this application is multifaceted. Firstly, it provides the reader with a low-level reference for driving the External Memory Bus Interface (ECBIU), which is available on select RA MCUs. Secondly, the application note serves as a guide for designing graphical applications using GUIX Studio for any Renesas or non-Renesas LCD controller, targeting RA MCUs without a GLCDC FSP module. Lastly, it explains how to improve performance using the DMA Controller to rotate the framebuffer.

The accompanying Hello World graphics application project provides a practical example of configuring and using the external memory bus to send data over the external address space. It also demonstrates the basics of designing a GUIX Studio project and how to develop a middleware layer to integrate the GUIX library drivers with the external memory bus (ECBIU) drivers.

Note that there are 2 versions of the Hello World application project. The main version, titled *HelloWorld_GUIX_ECBIU_EK_RA6M4* uses GUIX's built-in software rotation functionality to display the landscape GUI onto the portrait LCD. This main version is referenced throughout the entire application note as an example. The secondary version demonstrates the rotation by using the DMAC controller. The project is titled *HelloWorld_GUIX_ECBIU_EK_RA6M4__dmac_rotation* and will only be referenced in this application note under the Section Using DMAC to Rotate the Framebuffer.

Target Device

RA6, RA8

Required Resources

To build and run the example projects accompanying this application note, you will need the following:

Development tools and software:

- e^2 studio ISDE, version 2024-04 (24.04.0) or later
- RA Family Flexible Software package (FSP) v5.3.0 or later

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

Hardware:

- EK-RA6M4 (https://www.renesas.com/ek-ra6m4)
- Micro USB cable
- External LCD Controller (NHD-2.8-240320AF-CSXP-F)
- 40-pin FFC connector breakout board (NHD-FFC40)
- Jumper cables

Prerequisites and Intended Audience

This application note assumes you have some experience with the Renesas e² studio ISDE and RA Family Flexible Software Package (FSP). Before you perform the procedures in this application note, follow the procedure in the *FSP User Manual* to build and run the Blinky project. Doing so enables you to become familiar with e² studio and the FSP and validates that the debug connection to your board functions properly. Additionally, this application note assumes that you have some theoretical background on graphical applications with parallel RGB LCD controllers.

Contents

1. RA MCUs with an External Memory Bus

The RA6 Family Arm Cortex Microcontrollers support an external memory bus interface, which allows users to connect and interface external peripherals and/or memory devices to their RA6 MCU.

Note: The RA8 Family also supports an external memory bus, but the details differ from the RA6 line as described in this application note. Please refer to the appropriate reference documents for details regarding the external memory bus's implementation in the RA8 line.

The width of the external bus depends on the specific RA6 device part in question. In the RA6 MCU line, there are devices with 8-bit and 16-bit external memory bus widths, and for some devices with a very low output pin count, the external bus may not be present at all. Therefore, it's important to understand how to verify from hardware documentation which RA6 devices support the external bus and how to determine the bus's width.

First, check the device's Hardware User's Manual chapter on Buses to determine if the External Memory Interface CS Bus Interface Unit (ECBIU) is listed. As an example, the image below is from the RA6M4 Device Group Hardware User's Manual and highlights where in the Buses Chapter's overview table the ECBIU is listed.

Table 1. Identifying the ECBIU in the MCU Hardware Users' Manual Buses chapter

Classification	Bus Master/Slave name	Bus I/F Max Freq	Sync Clock	Specifications
Bus Masters	Code bus (Cortex-M33)	200 MHz	ICLK	Connected to the CPU Instruction Cache for instructions and operands
	System bus (Cortex-M33)	200 MHz	ICLK	Connected to the CPU Data Cache for system
	DMAC / DTC	200 MHz	ICLK	Connected to the DMAC/DTC
	EDMAC (Ether)	100 MHz	PCLKA	Connected to the EDMAC
Bus Slaves	FHBIU	200 MHz	ICLK	Connected to Code Flash memory and Configuration area
	FLBIU	50 MHz	FCLK	Connected to Data Flash memory, FACI
	S0BIU	200 MHz	ICLK	Connected to SRAM0 (Standby RAM)
	PSBIU	200 MHz	ICLK	Connected to peripheral system modules (DTC, DMAC, ICU, Flash, MPU, CSC, SRAM, Debug/Trace module, System controller and BUS controller)
	PLBIU	50 MHz	PCLKB	Connected to peripheral modules (CAC, ELC, I/O ports, POEG, RTC, WDT, IWDT, AGT, IIC, CAN, USBFS, SDHI, SSIE, TSN, and CTSU)
	PHBIU	100 MHz	PCLKA	Connected to peripheral modules (GPT, ETHERC, EDMAC, SCI, SPI, CRC, DOC, ADC12, DAC12 and SCE9)
	EQBIU (QSPI area)	100 MHz	PCLKA	Connected to the QSPI (External Memory Interface)
	EOBIU (OSPI area)	100 MHz	PCLKA	Connected to the OSPI (External Memory Interface)
	ECBIU (CS area)	100 MHz	BCLK	Connected to the external devices (External Memory Interface)

Note: BCLK (external-bus clock): 100MHz (max.) (The CSC (CS area controller) operate in synchronization with the BCLK)
BCLK pin output: The frequency is the same as the default of BCLK. 1/2 BCLK can be supplied by setting the EBCLK pin output select bit (BCKCR.BCLKDIV) in the external bus clock control register. For details, see section 8, Clock Generation Circuit.
FHBIU: Flash High speed Bus Interface Unit.
FLBIU: Flash Low speed Bus Interface Unit.
S0BIU: SRAM0 Bus Interface Unit.
PSBIU: Peripheral System Bus Interface Unit.
PLBIU: Peripheral Low speed Bus Interface Unit.
PHBIU: Peripheral High speed Bus Interface Unit.
EQBIU: External memory interface Qspi Bus Interface Unit.
EOBIU: External memory interface Ospi Bus Interface Unit.
ECBIU: External memory interface Csc Bus Interface Unit.

After verifying that the ECBIU is present, the bus width can be confirmed. The RA6 Hardware User's Manual **Overview chapter > Pin Lists** section table contains an "Ex. Bus" column that maps the external bus's data, address, chip select, and control signals to their respective physical I/O port on the MCU.

LQFP144	LQFP100	LQFP64	Power, System, Clock, Debug, CAC	I/O ports	Ex. Bus	Ex. Interrupt
110	—	—	—	P801	D15	—

Figure 1. Snippet of the Pin List confirming a 16-bit data bus width

Cross-reference the I/O port mapping with your device's pinout, to determine which external bus signals are physically available on your device. If D0-D15 signals are present in the pinout, the external bus can be configured to either 8-bit or 16-bit width. If only D0-D7 signals are present, the external bus can only be configured to 8-bit width.

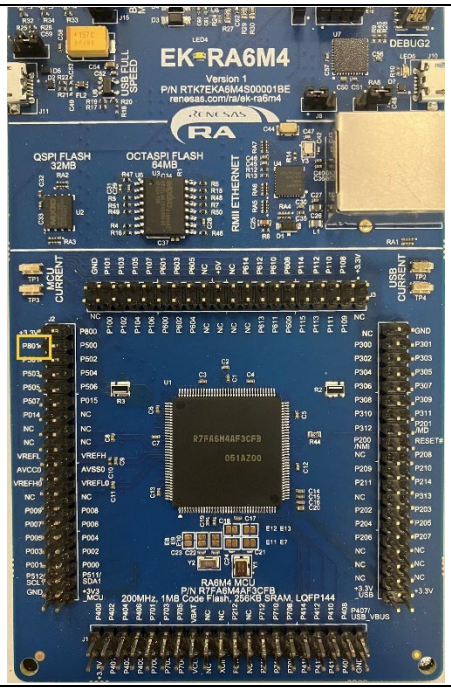


Figure 2. Confirming D15 physical pinout on the EK-RA6M4

1.1 RA6M4 Device Group

The information in this application note regarding the configuration and operation of the external bus, along with the accompanying application project code, is designed for the RA6M4 evaluation kit (EK-RA6M4). While some of the information may be transferrable to other RA devices, the exercise is left to the reader to utilize the respective device references to determine the correct operation and implementation details for their MCU.

For more information on the RA6M4 group, please visit the following references on the Renesas website:

- RA6M4 Group Hardware User's Manual: https://www.renesas.com/kr/en/document/man/ra6m4-group-user-s-manual-hardware
- RA6M4 Quick Start Guide: https://www.renesas.com/us/en/document/qsg/ek-ra6m4-quick-start-guide
- EK-RA6M4 device: https://www.renesas.com/us/en/products/microcontrollers-microprocessors/ra-cortex-m-mcus/ek-ra6m4-evaluation-kit-ra6m4-mcu-group

2. The External Memory Bus Interface

The external memory bus interface allows users to connect and interface with external peripheral and/or external memory devices from the MCU. This section will provide an overview of the specifications and features of the external memory bus module on the RA6M4 group devices. The section also covers the important ECBIU registers and their available settings, to provide the reader with knowledge on how to operate the external memory bus.

Code snippets from the accompanying application project are provided throughout this section to illustrate operating the ECBIU by writing directly to the peripheral registers. The pictures are intended to bring awareness to the reader for how the ECBIU registers will look in code. For a deeper analysis of the values assigned to the registers based on the target application, please refer to the details in Section 6.2.

2.1 Overview

The external memory bus (ECBIU) operates on the chip select area (CS area). The CS area controller (CSC) is the control unit for the external memory bus. The external memory bus includes the address, data, and control signals that are input and output from the MCU to external peripherals and memory devices. This section summarizes the key features of the external bus, maps out the address space of the CS area, and diagrams the external memory bus as part of the RA6M4 bus matrix.

Table 2. Overview of External Memory Bus Interface features

Parameter	Description
External address space	• The external address space is divided into 8 CS areas (CS0 to CS7). • Chip select signals can be output for each area. • The bus width can be set for each area: – Separate bus: Selectable to 8-bit or 16-bit bus space – Address/data multiplexed bus: Selectable to 8-bit or 16-bit bus space • Endian mode can be specified for each area.
CS area controller	• Recovery cycles can be inserted. – Read recovery: Up to 15 cycles – Write recovery: Up to 15 cycles • Cycle wait function: Wait for up to 31 cycles (for page access, up to 7 cycles) • Use wait control to set up the following: – Assertion and negation timing of chip select signals (CS0# to CS7#) – Assertion timing of the read signal (RD#) and write signals (WR0#/WR# and WR1# to WR3#) – Timing of data output starts and ends. • Write access modes: – Single-write strobe mode and byte strobe mode • Separate bus or address/data multiplexed bus can be set for each area.
Write buffer function	When write data from the bus master is written to the write buffer, write access by the bus master is complete.
Frequency	The CS area controller (CSC) operates in synchronization with the external bus clock (BCLK).
TrustZone Filter	Security attribution is always non-secure

Table 3 below lists the input and output pins of the external memory bus, as implemented by the bus structure internally.

Table 3. Pin descriptions for the ECBIU

Pin Name	I/O	Description
A23 to A0*1	Output	Address output pins
D15 to D0	I/O	Data input/output pins D15 to D0 pins are enabled when the 16-bit bus space is specified. D7 to D0 pins are enabled when the 8-bit bus space is specified.
BC0#*1	Output	A strobe signal; (the BC0# signal being at the low level) during access to an external address space in single write strobe mode indicates that D7 to D0 are valid. When an 8-bit bus space is specified, this output pin is always held low regardless of write access mode.
BC1#	Output	A strobe signal; (the BC1# signal being at the low level) during access to an external address space in single write strobe mode indicates that D15 to D8 are valid. This pin is not used when the 8-bit bus space is specified.
CS0#	Output	A chip select signal for area 0 (CS0)
CS1#	Output	A chip select signal for area 1 (CS1)
CS2#	Output	A chip select signal for area 2 (CS2)
CS3#	Output	A chip select signal for area 3 (CS3)
CS4#	Output	A chip select signal for area 4 (CS4)
CS5#	Output	A chip select signal for area 5 (CS5)
CS6#	Output	A chip select signal for area 6 (CS6)
CS7#	Output	A chip select signal for area 7 (CS7)
RD#	Output	A strobe signal indicating that reading from an external address space (CS0 to CS7) is in progress
WR0#/WR#*2	Output	WR0# signal is a strobe signal indicates that (the WR0# signal being at the low level) writing to an external address space is in progress in byte strobe mode, and D7 to D0 are valid. WR# signal is a strobe signal that indicates writing to an external address space is in progress in single write strobe mode. When an 8-bit bus space is specified, this output pin is held low during a write access regardless of write access mode.
WR1#	Output	A strobe signal; (the WR1# signal being at the low level) during writing to an external address space in byte strobe mode indicates that D15 to D8 are valid. This signal is invalid in single write strobe mode. This pin is not used when the 8-bit bus space is specified.
ALE	Output	Address latch signal when address/data multiplexed bus is selected.
WAIT#	Input	A wait request signal when accessing the external address space (CS0 to CS7) (Low: Wait request)

Note 1. The A0 and BC0# pin functions share the same pin, and either become effective according to the area, with the function being A0 in byte strobe mode and BC0# in single write strobe mode. Setting the 8-bit external bus width is prohibited in single write strobe mode. For information on other multiplexed pin functions, see section 19, I/O Ports.

Note 2. The WR0# signal and WR# signal are identical. The WR0# signal is particularly referred to as WR# in single write strobe mode.

2.1.1 External address space

It's important to note that the actual available pinout from an MCU may differ from the internal hardware modules' specified signals. Therefore, you should review your device's **Hardware User's Manual > Overview chapter > Pin Lists** section to understand which IO signals are physically routed in and out of the MCU, and to become aware of any shared assignments on physical pins.

For the external memory bus on the EK-RA6M4 there are physically 21 address pins available for use in the Pin Lists section, from A0 through A20. That means that each CS area can access an address size of 0x1F_FFFF. The external address space (CS area) begins at the address 0x8000_0000. The following image shows how the CS area is divided into 8 different regions, one for each CS signal:

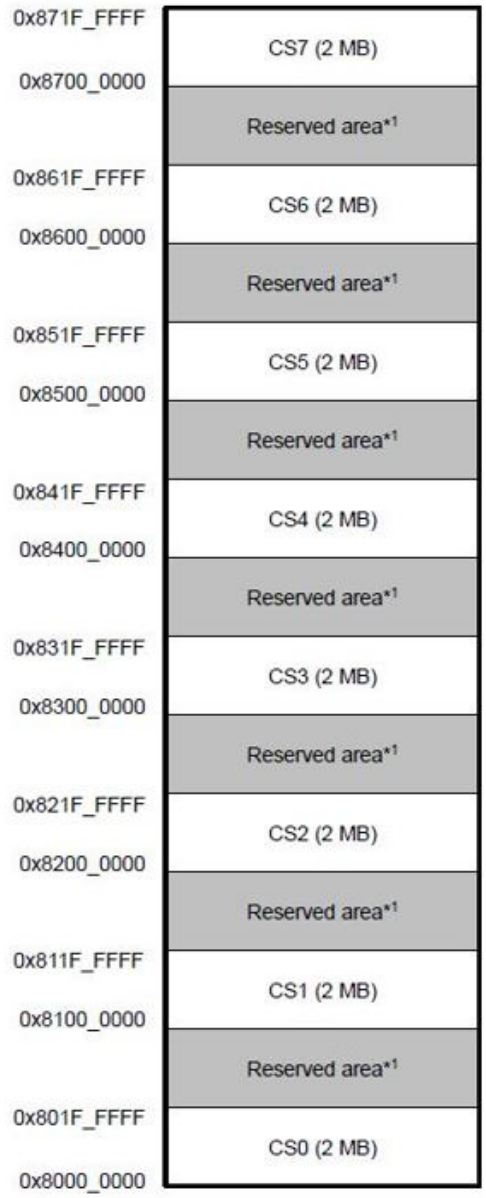


Figure 3. The address space of the CS area on the RA6M4

2.1.2 RA6M4 Bus Matrix

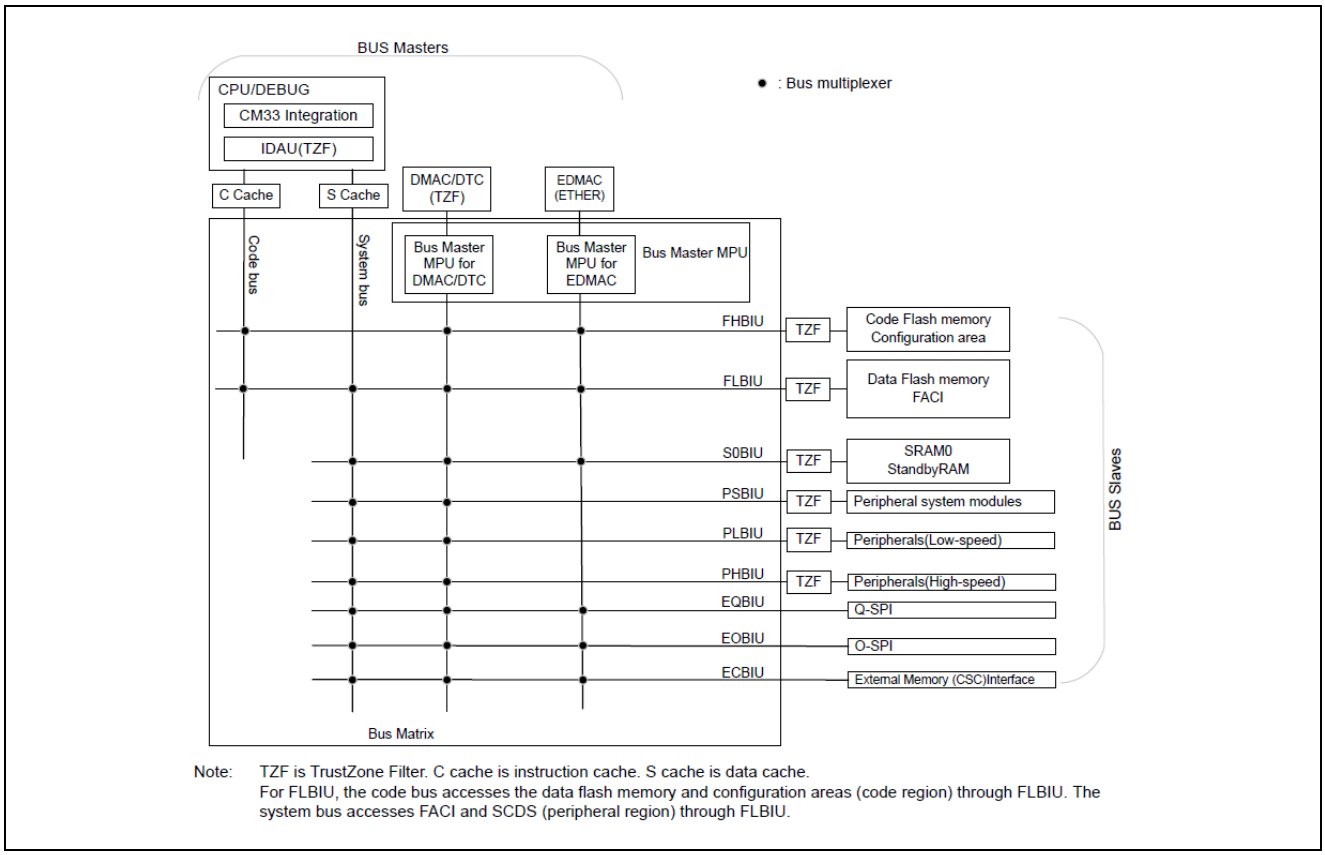
The RA6M4 has a 32-bit Advanced High-performance Bus (AHB) matrix that contains several bus masters and bus slaves. The following table lists the specifications of the buses on the RA6M4 and Figure 4 depicts the connections between buses in the matrix.

Table 4. Bus Masters and Slaves on RA6M4

Classification	Bus Master/Slave name	Bus I/F Max Freq	Sync Clock	Specifications
Bus Masters	Code bus (Cortex-M33)	200 MHz	ICLK	Connected to the CPU Instruction Cache for instructions and operands
	System bus (Cortex-M33)	200 MHz	ICLK	Connected to the CPU Data Cache for system
	DMAC / DTC	200 MHz	ICLK	Connected to the DMAC/DTC
	EDMAC (Ether)	100 MHz	PCLKA	Connected to the EDMAC
Bus Slaves	FHBIU	200 MHz	ICLK	Connected to Code Flash memory and Configuration area
	FLBIU	50 MHz	FCLK	Connected to Data Flash memory, FACI
	S0BIU	200 MHz	ICLK	Connected to SRAM0 (Standby RAM)
	PSBIU	200 MHz	ICLK	Connected to peripheral system modules (DTC, DMAC, ICU, Flash, MPU, CSC, SRAM, Debug/Trace module, System controller and BUS controller)
	PLBIU	50 MHz	PCLKB	Connected to peripheral modules (CAC, ELC, I/O ports, POEG, RTC, WDT, IWDT, AGT, IIC, CAN, USBFS, SDHI, SSIE, TSN, and CTSU)
	PHBIU	100 MHz	PCLKA	Connected to peripheral modules (GPT, ETHERC, EDMAC, SCI, SPI, CRC, DOC, ADC12, DAC12 and SCE9)
	EQBIU (QSPI area)	100 MHz	PCLKA	Connected to the QSPI (External Memory Interface)
	EOBIU (OSPI area)	100 MHz	PCLKA	Connected to the OSPI (External Memory Interface)
	ECBIU (CS area)	100 MHz	BCLK	Connected to the external devices (External Memory Interface)

Note: BCLK (external-bus clock): 100MHz (max.) (The CSC (CS area controller) operate in synchronization with the BCLK)
BCLK pin output: The frequency is the same as the default of BCLK. 1/2 BCLK can be supplied by setting the EBCLK pin output select bit (BCKCR.BCLKDIV) in the external bus clock control register. For details, see section 8, Clock Generation Circuit.
FHBIU: Flash High speed Bus Interface Unit.
FLBIU: Flash Low speed Bus Interface Unit.
S0BIU: SRAM0 Bus Interface Unit.
PSBIU: Peripheral System Bus Interface Unit.
PLBIU: Peripheral Low speed Bus Interface Unit.
PHBIU: Peripheral High speed Bus Interface Unit.
EQBIU: External memory interface Qspi Bus Interface Unit.
EOBIU: External memory interface Ospi Bus Interface Unit.
ECBIU: External memory interface Csc Bus Interface Unit.

The external bus controller arbitrates requests for bus mastership on the external address space and external bus controller registers (CSC) from the bus masters, the CPU, DMAC/DTC and EDMAC (ethernet) buses. The arbitration between the three master busses for each slave bus can be configured to fixed-priority or round-robin methods by the user.



Note: TZF is TrustZone Filter. C cache is instruction cache. S cache is data cache.
For FLBIU, the code bus accesses the data flash memory and configuration areas (code region) through FLBIU. The system bus accesses FACI and SCDS (peripheral region) through FLBIU.

Figure 4. RA6M4 Bus Connections Matrix

2.2 Operating the External Bus

There are many ways to customize the external memory bus's operation timing sequence, which makes the external bus compatible with a large variety of peripheral and memory devices that a user may wish to add to their design. Adding external devices provides users with greater flexibility and complexity in their system designs, beyond what the RA6M4 offers on-chip.

This section will review the important external memory bus registers and explain how to set them for a custom bus timing sequence for generic operation, such as setting the bus's data bit-width, chip select options, bus endianness, and wait state settings. This section will provide the reader with a more detailed look into operating the external bus at the register level. However, readers should also refer to the complete registers' descriptions in the Buses chapter of Hardware User's Manual when developing custom applications.

Figure 5 illustrates the signal timing diagram for a normal read and write operation. The remaining sections will explain the I/O signals in greater detail and will review the effect on the signals based on the bus operation mode selected and the specified signals' cycle extensions.

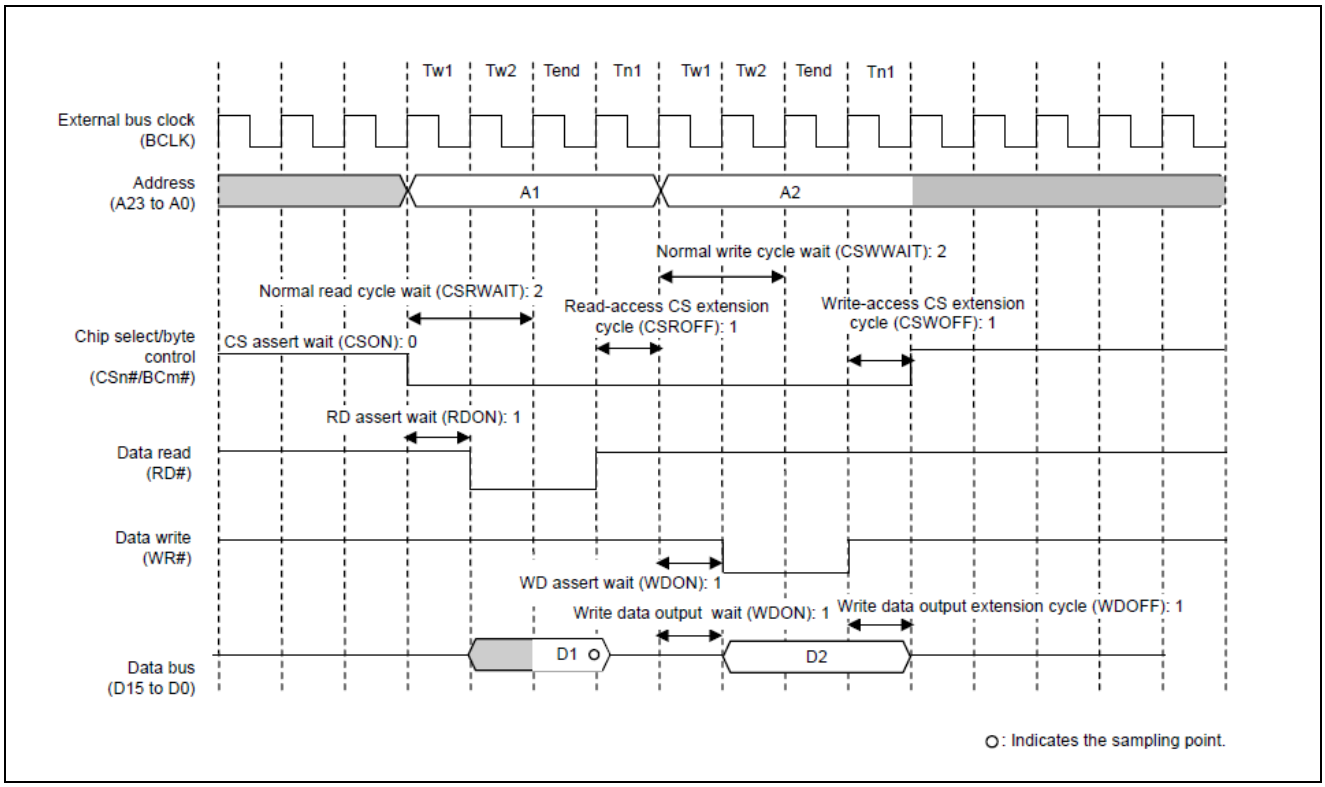


Figure 5. Normal access operation for a read and write sequence

2.2.1 External Bus Clock

The CSC operates in synchronization internally with the external bus clock (BCLK). Any operation cycles, such as wait cycles specified in the CSC register, are counted on BCLK.

The division ratios of the BCLK signal can be set by using the BCK[2:0] bitfield of the System Clock Division Control Register (SCKDIVCR).

The external bus clock can also be supplied externally by using the EBCLK signal. Typically, this signal acts in synchronicity with BCLK, but it can be set with a different frequency. To enable the EBCLK set the Pin Output Control EBCKOEN bitfield of the External Bus Clock Output Control Register (EBCKOCR).

The following table shows the clock generation circuit specifications for the external memory bus clocks BCLK and EBCLK:

Table 5. External bus clock specifications

Clock	Clock Source	Clock Supply	Specification
External bus clock (BCLK)	MOSC/SOSC/HOCO/MOCO /LOCO/PLL	External bus	Up to 100 MHz Division Ratio: 1/2/4/8/16/32/64
EBCLK pin output (EBCLK)	BCLK or ½ BCLK	EBCLK pin	Up to 50 MHz Division Ratio: 1 or 2

The division ratio and frequency of BCLK and EBCLK can be set in an FSP project by editing the Clocks in the configuration.xml file. The FSP's code generation tool will generate an automatic routine to set all clock registers according to the configuration.xml settings (including but not limited to EBCKOCR and SCKDIVCR).

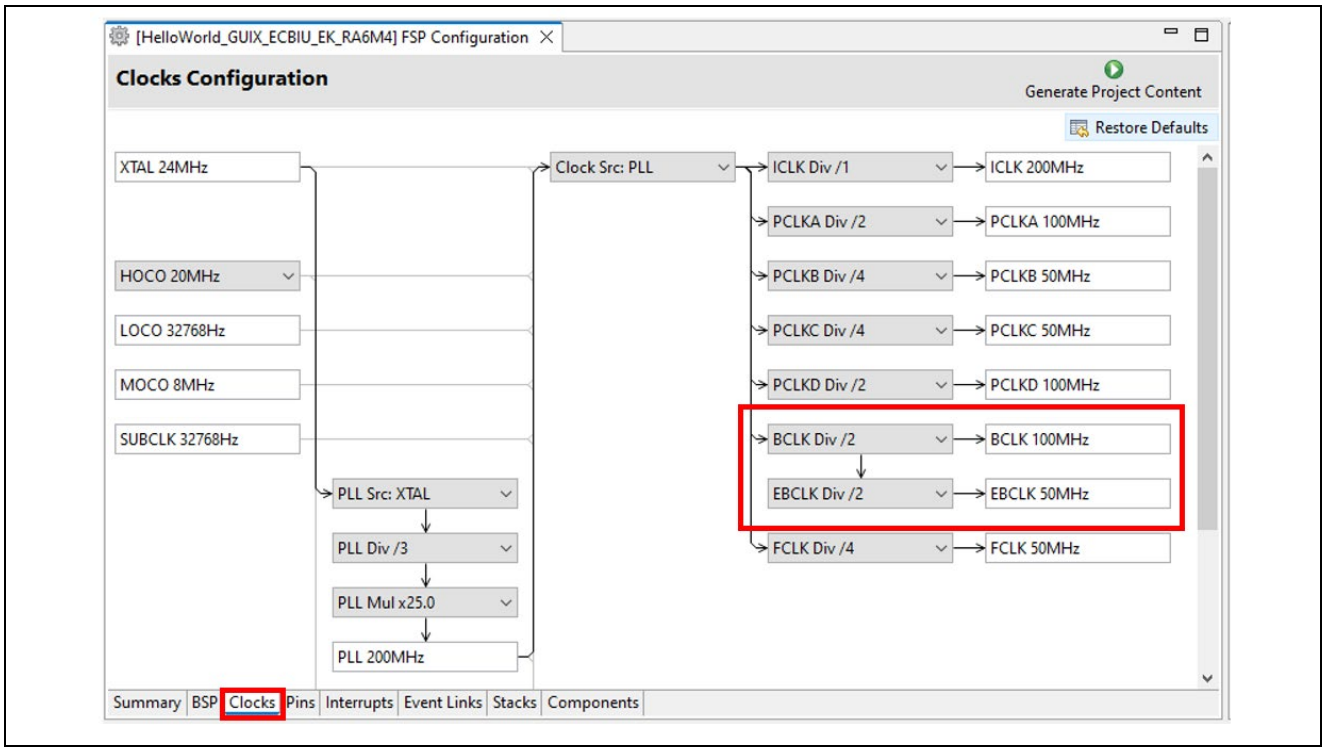


Figure 6. Setting the frequencies of BLCK and EBCLK in the configuration.xml

If you would like to supply EBCLK to an external device, you can set the pin output by editing the Pins in the configuration.xml. EBLCK configuration is located under the System:BUS > BUS_ASYNC0 for the RA6M4.

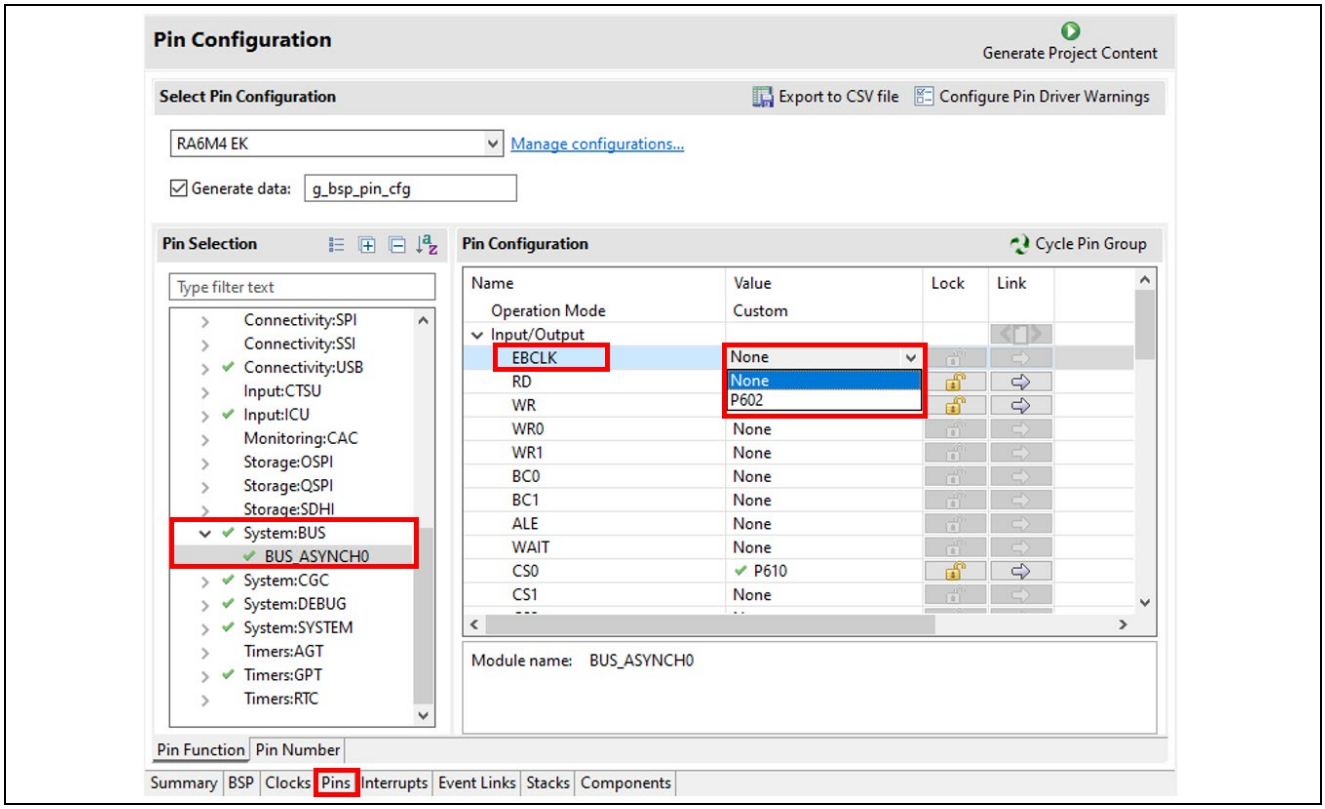


Figure 7. Enabling EBCLK output on P602 on the RA6M4

2.2.2 Chip Select Signals

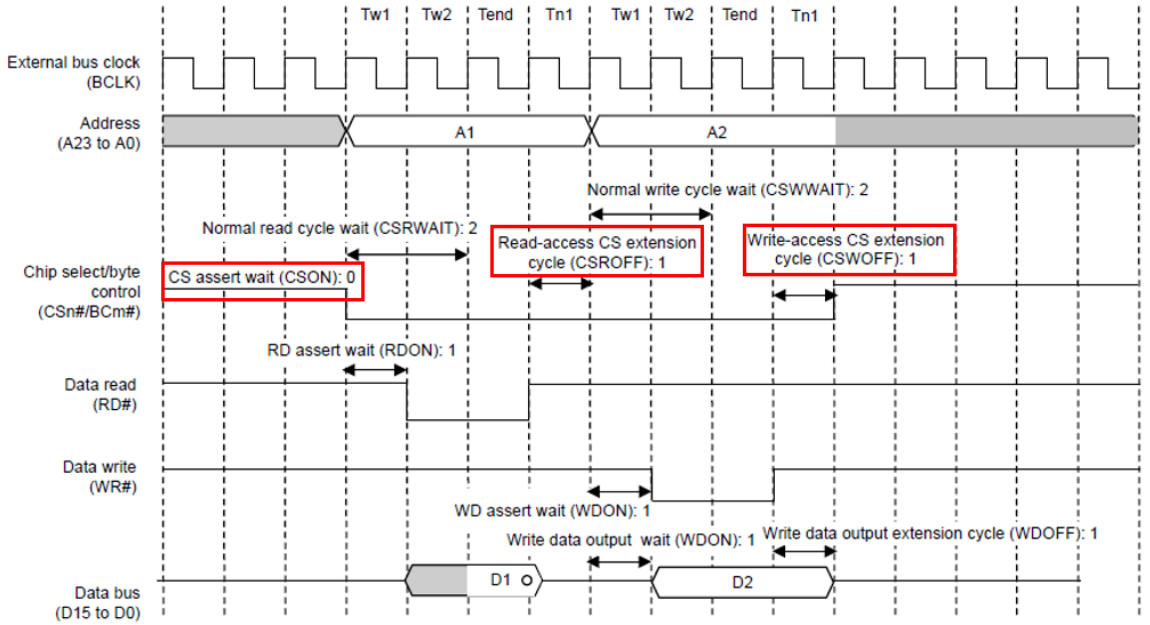
The external address space is divided into 8 chip select areas, from CS0 through CS7. Each chip select area has a unique chip select signal which is output to pin CSn (n = 0-7), as listed in Table 3. The physical pinout

of the chip select signals is detailed in the **Hardware User's Manual > Overview chapter > Pin Lists** section.

Wait control can set up the assertion and negation timing of the chip select signals CS0# through CS7#. The CSn Wait Control Register 2, abbreviated CSnWCR2 (n = 0-7), contains 3 different bitfields that affect the timing of the CSn# signals:

1. CSnWCR2.CSROFF[2:0] bits specify the number of wait cycles to be inserted in a time period from the end of a wait cycle (RD signal negated) until the CSn# signal (n = 0-7) is negated in read access mode.
2. CSnWCR2.CSWOFF[2:0] bits specify the number of wait cycles to be inserted in a time period from the end of a wait cycle (WR0 and WR1 signals negated) until the CSn# signal (n = 0-7) is negated in write access mode.
3. CSnWCR2.CSON[2:0] bits specify the number of wait cycles to be inserted before the CSn# signal (n = 0-7) is asserted. Please visit the HW UM for the conditions that this bitfield must satisfy, depending on the read/write access mode and the address/data I/O interface mode.

Review the figure below to see a sample routine that will set the specified CS signal timing registers to match the diagram:




```
#include "bsp_api.h"

void ecbiu_config_cs_signal(void)
{
    R_BUS->CSa[0].WCR2_b.CSROFF = 1;    //Set read-access CS extension cycle select, adds 1 BCLK cycle
    R_BUS->CSa[0].WCR2_b.CSWOFF = 1;    //Set write-access CS extension cycle select, adds 1 BCLK cycle
    R_BUS->CSa[0].WCR2_b.CSON   = 0;    //Set CS assert wait select, adds 0 BCLK cycles
}
```

Figure 8. Example of configuring the signal timing of the CS signal on the RA6M4's ECBIU

2.2.3 Data Width

The external memory bus's data width can be set for each chip select area. Each CS area can be set to either 8-bit or 16-bit width by through the CSn (n=0-7) Control Register (abbreviated CSnCR), by setting the right value in the BSIZE[1:0] bitfield.

The valid settings for CSnCR.BSIZE[1:0] are as follows:

- 0b00: 16-bit bus space selected
- 0b10: 8-bit bus space selected

All other settings are prohibited. The following image is a simple routine for setting the data width of the ECBIU:

```
#include "bsp_api.h"

#define     DATA_WIDTH_16  (1)

void ecbiu_config_data_width(void)
{
#if  DATA_WIDTH_16
    R_BUS->CSb[0].CR_b.BSIZE = 0;   //Set 16-bit data width on ECBIU
#else
    R_BUS->CSb[0].CR_b.BSIZE = 2;   //Set 8-bit data width on ECBIU
#endif
}
```

Figure 9. Example of setting the RA6M4's ECBIU data width to 16 bits

2.2.4 Endianness

The endianness of the data output to the external address area can be specified for each chip select area with the CSn Control Register, abbreviated CSnCR (n=0-7), using the EMODE bitfield.

The valid settings for CSnCR.EMODE are as follows:

- 0: little-endian
- 1: big-endian

Because the CPU is fixed to little-endian alignment, when the endian setting for each area is different from that for the MCU, no instruction code can be allocated in the area. The instruction code should only be allocated to the external space's chip select areas that are set to little-endian.

```
#include "bsp_api.h"

#define LITTLE_ENDIAN   (1)

ecbiu_config_endianness(void)
{
#if LITTLE_ENDIAN
    R_BUS->CSb[0].CR_b.EMODE = 0;   //Set little-endian mode
#else
    R_BUS->CSb[0].CR_b.EMODE = 1;   //Set big-endian mode
#endif
}
```

Figure 10. Example of setting the RA6M4's ECBIU data output to little endian

2.2.5 External Bus Read and Write Modes: CSnMOD

The Chip Select Mode Register, abbreviated CSnMOD (n=0-7) contains several bitfields that specify the various read and write access modes, and whether their operation is enabled or disabled for each chip select area. Each bitfield of the CSnMOD register is explained further below:

(1) Write Access Mode Select

The external bus's write access mode can be specified for each chip select area with the CSn Mode Register, abbreviated CSnMOD (n=0-7) using the WRMOD bitfield.

The valid settings for CSnMOD.WRMOD are as follows:

- 0: Byte strobe mode selected
- 1: Single write strobe mode

As seen in Table 6. Control Signals for Write Access ModeTable 6, while in byte strobe mode, the data write operation is controlled by the WR0 and WR1 signals corresponding to the respective byte positions. In single

write strobe mode, the data write operation is controlled by the BC0, BC1, and WR signals corresponding to respective byte positions. Setting the external bus width of 8 bits is prohibited in single write strobe mode.

Table 6. Control Signals for Write Access Mode

Mode	Pin Name			
Write Access Mode	**WR1#**	**WR0#/WR#**	**BC1#**	**BC0#**
Byte strobe mode	✓	✓ (WR0#)	—	—
Single write strobe mode	—	✓ (WR#)	✓	✓

Note: ✓: Enabled, —: Disabled

(2) External Wait Enable

The external wait enable bit EWNB is used to enable or disable the external wait signal WAIT#. When the external wait signal is enabled, then it's possible to control the number of waits in each cycle of the WAIT# signal. Wait cycles will be inserted while the WAIT# signal is at the low level.

The valid settings for CSnMOD.EWNB are:

- 0: Disable the WAIT# signal
- 1: Enable the WAIT# signal

Wait cycles can be extended by the WAIT# signal for both the normal read/write cycle and for the page access read/write cycle. See Section 2.2.7 for an explanation of setting the number of wait cycle extensions.

(3) Page Read Access Enable

This Page Read Access Enable PRENB bit enables or disables page read access. The valid settings for CSnMOD.PRENB are:

- 0: Page read access is disabled
- 1: Page read access is enabled

Note: When the address/data multiplexed I/O interface is selected (CSnCR.MPXEN bit set to 1) this bit should not be set for page read access. Page read access is not supported in the address/data multiplexed I/O interface.

(4) Page Write Access Enable

This Page Write Access Enable PWENB bit enables or disables page write access. The valid settings for CSnMOD.PWENB are:

- 0: Page write access is disabled
- 1: Page write access is enabled

Note: When the address/data multiplexed I/O interface is selected (CSnCR.MPXEN bit set to 1) this bit should not be set for page write access. Page write access is not supported in the address/data multiplexed I/O interface.

(5) Page Read Access Mode Select

This bit selects a page read access operating mode. The valid settings for CSnMOD.PRMOD are:

- 0: Normal access compatible mode
- 1: External data read continuous assertion mode

In normal access compatible mode, the RD# signal is negated, and RD# assert wait is inserted each time data is read. However, when there is no RD# assert wait, the RD# signal is negated only in the final transfer of the external bus access.

In external data read continuous assertion mode, the RD# assert wait is inserted and the RD# signal is continuously asserted during this time period.

Review the following image for an example routine that configures the read and write modes mentioned:

```
#include "bsp_api.h"

void ecbiu_config_read_write_modes(void_)
{
    R_BUS->CSa[0].MOD_b.WRMOD = 0;      /* Set the Write Access Mode Select bit
                                           (0 = byte strobe mode, 1 = single write strobe mode) */

    R_BUS->CSa[0].MOD_b.EWENB = 0;      /* Set the External Wait Enable bit
                                           (0 = disable wait signal, 1 = enable wait signal) */

    R_BUS->CSa[0].MOD_b.PRENB = 0;      /* Set the Page Read Access Enable bit
                                           (0 = disable page read access, 1 = enable page read access) */

    R_BUS->CSa[0].MOD_b.PWENB = 0;      /* Set the Page Write Access Enable bit
                                           (0 = disable page write access, 1 = enable page write access) */

    R_BUS->CSa[0].MOD_b.PRMOD = 0;      /* Set the Page Read Access Mode Select bit
                                           (0 = normal access compatible mode, 1 = external data read continuous assertion mode */
}
```

Figure 11. Example of setting the RA6M4's ECBIU read and write modes

2.2.6 Recovery Cycles

For each external bus chip select area, recovery cycles can be inserted between consecutive rounds of external bus access. This capability allows users to match the timing of the bus to the specifications of the read and write cycles of their external peripheral or memory device.

Recovery cycles can be inserted on any of the following conditions:

- After a read access to the external bus, a read access is made to the external bus in the same area
- After a read access to the external bus, a read access is made to the external bus in a different area
- After a read access to the external bus, a write access is made to the external bus in the same area
- After a read access to the external bus, a write access is made to the external bus in a different area
- After a write access to the external bus, a read access is made to the external bus in the same area
- After a write access to the external bus, a read access is made to the external bus in a different area
- After a write access to the external bus, a write access is made to the external bus in the same area
- After a write access to the external bus, a write access is made to the external bus in a different area

The recovery cycle starts at the end of the preceding bus cycle, for example when the CSn# signal (n = 0-7) is held low. A high-level period of the CSn# signal is inserted for the specified recovery cycle period starting from this point.

In the fastest case, the CSn# signal for the next round of bus access is asserted immediately after the end of recovery cycles. Even if the next request for access to an external address space is generated during the recovery period, the next round of access over the external bus will start immediately after the end of recovery cycles.

The CSn Recovery Cycle Insertion Enable Register, abbreviated CSRECEN, contains bitfields for enabling the bus recovery cycle insertion, based on the separate/multiplexed data setting for chip select area n.

The CSn Recovery Cycle Register, abbreviated CSnREC (n = 0-7), contains bitfields specifying the number of recovery cycles to be inserted after a read and/or access to the external bus for chip select area n.

The image below shows how the CSRECEN and the CSnREC registers were set for the accompanying application project on the RA6M4 MCU. Please refer to section 6.2 for an analysis on the signal timing and why these values were chosen.

```
R_BUS->CSb[0].REC_b.RRCV = RECOVERY_CYCLES_3;       // Read recovery cycles: 3
R_BUS->CSb[0].REC_b.WRCV = RECOVERY_CYCLES_3;       // Write recovery cycles: 3

R_BUS->CSRECEN_b.RCVEN0 = RECOVERY_CYCLES_1;        // Recovery cycles specified by RRCV are inserted between cycles: read to read, in same area
R_BUS->CSRECEN_b.RCVEN2 = RECOVERY_CYCLES_1;        // Recovery cycles specified by RRCV are inserted between cycles: read to write, in same area
R_BUS->CSRECEN_b.RCVEN4 = RECOVERY_CYCLES_1;        // Recovery cycles specified by WRCV are inserted between cycles: write to read, in same area
R_BUS->CSRECEN_b.RCVEN6 = RECOVERY_CYCLES_1;        // Recovery cycles specified by WRCV are inserted between cycles: write to write, in same area
```

2.2.7 Wait States

There are a variety of timing scenarios during which wait cycles can be inserted to extend certain external memory bus signal lengths to achieve the desired signal timing.

This section attempts to broadly summarize the key external wait state scenarios, but it is a complicated subject so it is strongly recommended to carefully review the Buses Chapter of the Hardware User's Manual as well. The manual will provide greater detail on signal timing, depending on the specific desired operation and clock settings.

2.2.7.1 Normal Access External Wait

The following applies if the external wait is enabled by the CSnMOD.EWNB bit, and the bus is configured to operate in normal read and/or write access mode.

For the normal read and write access cycles, the extended length of the wait signal is specified in number of clock cycles with the Normal Read Cycle Wait Select CSRWAIT[4:0] bitfield and the Normal Write Cycle Wait Select CSWWAIT[4:0] bitfield of the Chip Select Wait Control Register 1 CSnWCR1. All wait cycles specified in CSnWCR1 are inserted independently of the WAIT# signal.

2.2.7.2 Page Access External Wait

The following applies if the external wait is enabled by the CSnMOD.EWNB bit, and the bus is operating in page read and/or write access mode.

For the page read and write access cycles, the extended length of the wait signal is specified in number of clock cycles with the Page Read Cycle Wait Select CSPRWAIT[2:0] bitfield and the Page Write Cycle Wait Select CSPWWAIT[2:0] bitfield of the Chip Select Wait Control Register 1 CSnWCR1. All wait cycles specified in CSnWCR1 are inserted independently of the WAIT# signal.

Figure 12 and Figure 13 depict the page read and page write timing diagrams to visualize how the CSPRWAIT and CSPWWAIT cycles impact the length of the wait signal. For these example timing diagrams, note that the scenario is for the 16-bit bus space when BCLK is chosen with the EBCLK Pin Output Select bit.

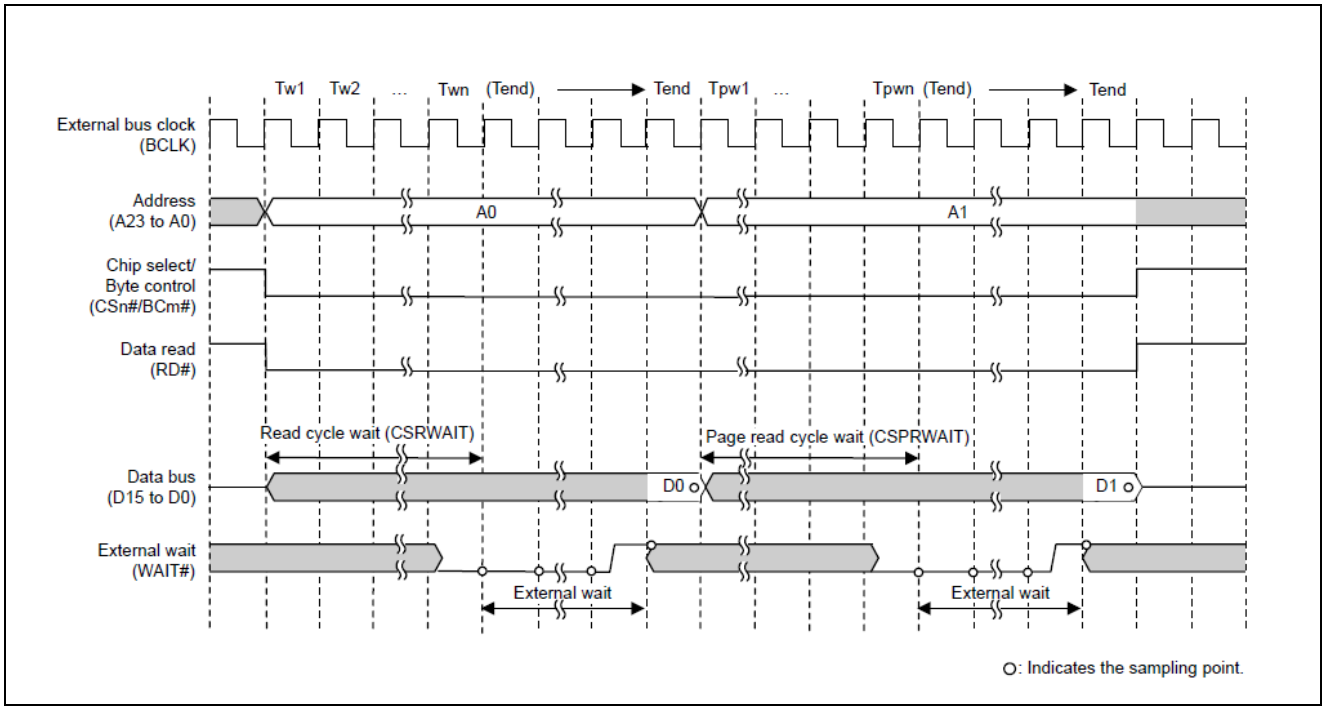


Figure 12. External wait timing diagram for page read access

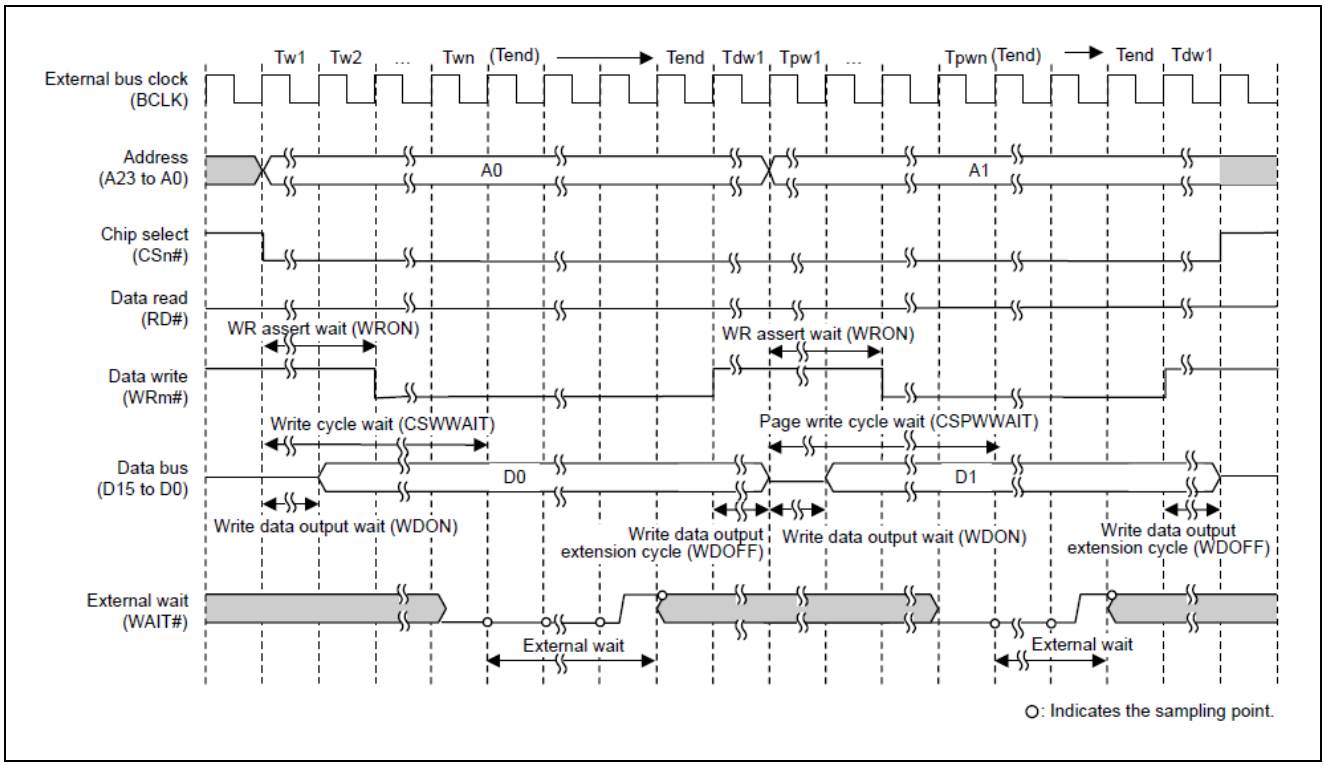


Figure 13. External wait timing diagram for page write access

2.2.7.3 Address/Data Multiplexed I/O Interface

When the address/data multiplexed I/O interface is selected, any programmed waits and pin waits in the data cycle that use the WAIT# pin can be inserted in the same manner as the separate bus interface.

Address cycles are not affected by the wait control settings. Figure 14 shows an example of external wait insertion timing with the address/data multiplexed I/O interface.

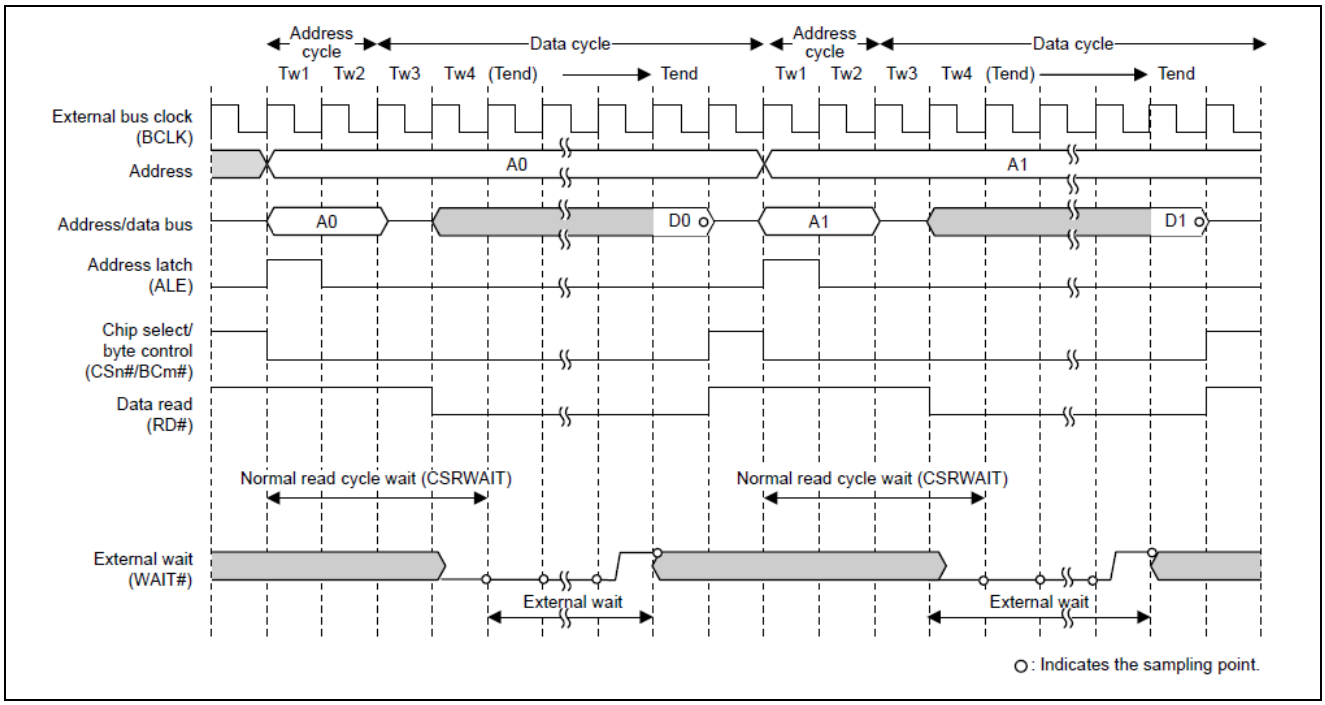


Figure 14. External wait insertion timing diagram with address/data multiplexed I/O interface

2.2.8 Bus Arbitration

The external bus controller arbitrates requests for bus mastership on the external address space and external bus controller registers (CSC) from CPU, DMAC/DTC and EDMAC. The arbitration between the three master busses for each slave bus can be configured to fixed-priority or round-robin methods by the user.

The Slave Bus Control Register BUSSCNTECBIU, sets the method of priority by setting the desired function with the Arbitration Select ARBS[1:0] bitfield.

The valid settings for BUSSCNTECBIU.ARBS[1:0] are as follows (> indicates the fixed priority order and ↔ indicates round-robin):

- 0b00: EDMAC > DMAC/DTC > CPU
- 0b01: Setting Prohibited
- 0b10: (EDMAC ↔ DMAC/DTC) > CPU
- 0b11: (EDMAC ↔ DMAC/DTC) ↔ CPU

```
#include "bsp_api.h"

void ecbiu_config_bus_arbitration(void)
{
    R_BUS->BUSS[5].CNT_b.ARBS = 3;      /* Set bus arbitration to: (EDMAC <-> DMAC/DTC ) <-> CPU
}
```

Figure 15. Example setting the RA6M4's ECBIU's bus arbitration control select register

3. Designing with GUIX Studio

The application project GUI was designed with the Azure GUIX Studio PC application. Azure GUIX Studio is a free Windows GUI design software intended for use with the Azure RTOS GUIX runtime library in embedded applications.

This section will give a brief overview of the GUIX Studio Windows application, covering where to download the software, how to personalize a GUIX project, and how to export a GUIX project as C code for use in an Azure RTOS ThreadX e^2 studio project.

3.1 GUIX Studio Download and References

Azure RTOS GUIX Studio is a Microsoft Windows-based rapid UI development environment specifically designed for the GUIX runtime library from Microsoft. It is intended for the embedded real-time software developer using the ThreadX Real-Time Operating System (RTOS) and the Azure RTOS GUIX UI run-time Library (or more simply, GUIX Library). Users designing with GUIX Studio should first familiarize themselves with standard Azure RTOS ThreadX and GUIX Library concepts.

For more information on the Azure RTOS GUIX run-time library, please visit the GUIX User Guide on the Microsoft Website.

For more information about the GUIX Studio UI application for Windows, please visit the GUIX Studio User Guide on the Microsoft Website.

The Renesas Flexible Software Package (FSP) includes the Azure RTOS ThreadX real-time operating system, the Azure RTOS GUIX library and hardware drivers unified under a single robust software package. This powerful suite of tools provides a comprehensive integrated framework for rapid development of complex embedded applications.

To download the FSP, which includes the ThreadX RTOS and GUIX Library and supporting functionality for RA devices, please visit the Renesas FSP Github page.

You can download the GUIX Studio PC application from the Microsoft Store already installed on your Windows device, or by visiting the Microsoft Store's Azure RTOS GUIX Download page.

3.2 Creating Custom GUIs with GUIX Studio

Embedded graphical developers can utilize the GUIX Studio screen designer to quickly create and update an embedded UI design with the GUIX run-time environment. GUIX Studio designs are saved and maintained in a GUIX Studio project file, with the extension *.gxp.*

This section explains how to configure a GUIX Studio project for any target LCD and how to add custom designs to the project. This section references the software rotation version of the Hello World example's GUIX Studio project *hello_world_ra6m4.gxp* as an example. The details that are LCD-specific are emphasized so this section can serve as a general reference on using GUIX Studio with any external LCD.

This section is not meant as a replacement for the GUIX documentation. When designing graphical interfaces for the Renesas FSP platform, you are encouraged to refer to the documentation for GUIX Studio and the Azure GUIX run-time library.

3.2.1 GUIX Studio Project Layout

GUIX Studio allows you to manage all the UI resources your application will use, including colors, fonts, pixel-maps, strings, and more. There are some basic fonts and pixelmap graphics included in a default GUIX Studio project to help get a simple design up and running, but you can also add your own external images and font files to customize the design fully to your needs.

When you navigate to the project path *HelloWorld_GUIX_ECBIU_EK_RA6M4\guix_studio\GNU* and open the GUIX Studio project *hello_world_ra6m4.gxp*, GUIX Studio opens with the default project view.

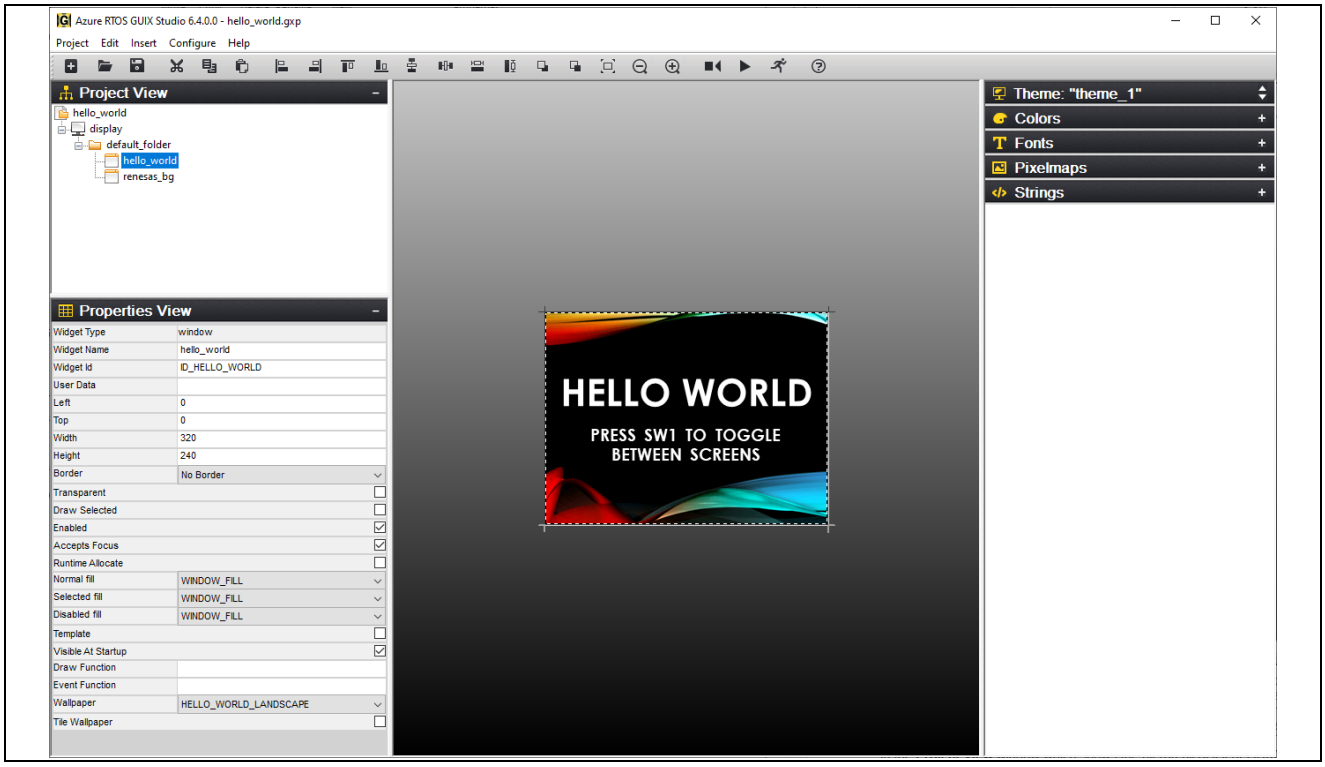


Figure 16. The default GUIX Studio project view

The organization of GUIX Studio is straightforward.

The **Toolbar** across the top of the window shows the buttons available to the GUIX Studio developer for interacting with the project.

The **Target View** is the middle window, which shows the design and layout of the display screen. Objects in the target view can be selected, moved, and resized using the mouse.

When you select an item in **Target View** you will see all the properties for that object displayed in the **Properties View** window, the bottom left panel.

That action will also highlight the object in the **Project View** window, the top left panel, which shows the hierarchical list of GUIX objects that comprise the embedded UI.

The **Resource View** window is the panel on the right, and it's where you can manage the colors, fonts, pixelmaps, and strings available to the application screens defined for each display.

3.2.2 Configuring a GUIX Studio Project

When you create a new GUIX Studio project the first window to pop up is the Configure Project dialog. Here you can set the number of hardware displays available on the target and specify the properties of each

display. Properties include the display's name, x/y resolution, color depth and format, among others. You can view and edit the project configurations at any moment by selecting **Configure** > **Project/Displays** from the GUIX Studio menu at the top of the project window.

The project properties of *hello_world_ra6m4.gxp* are shown below:

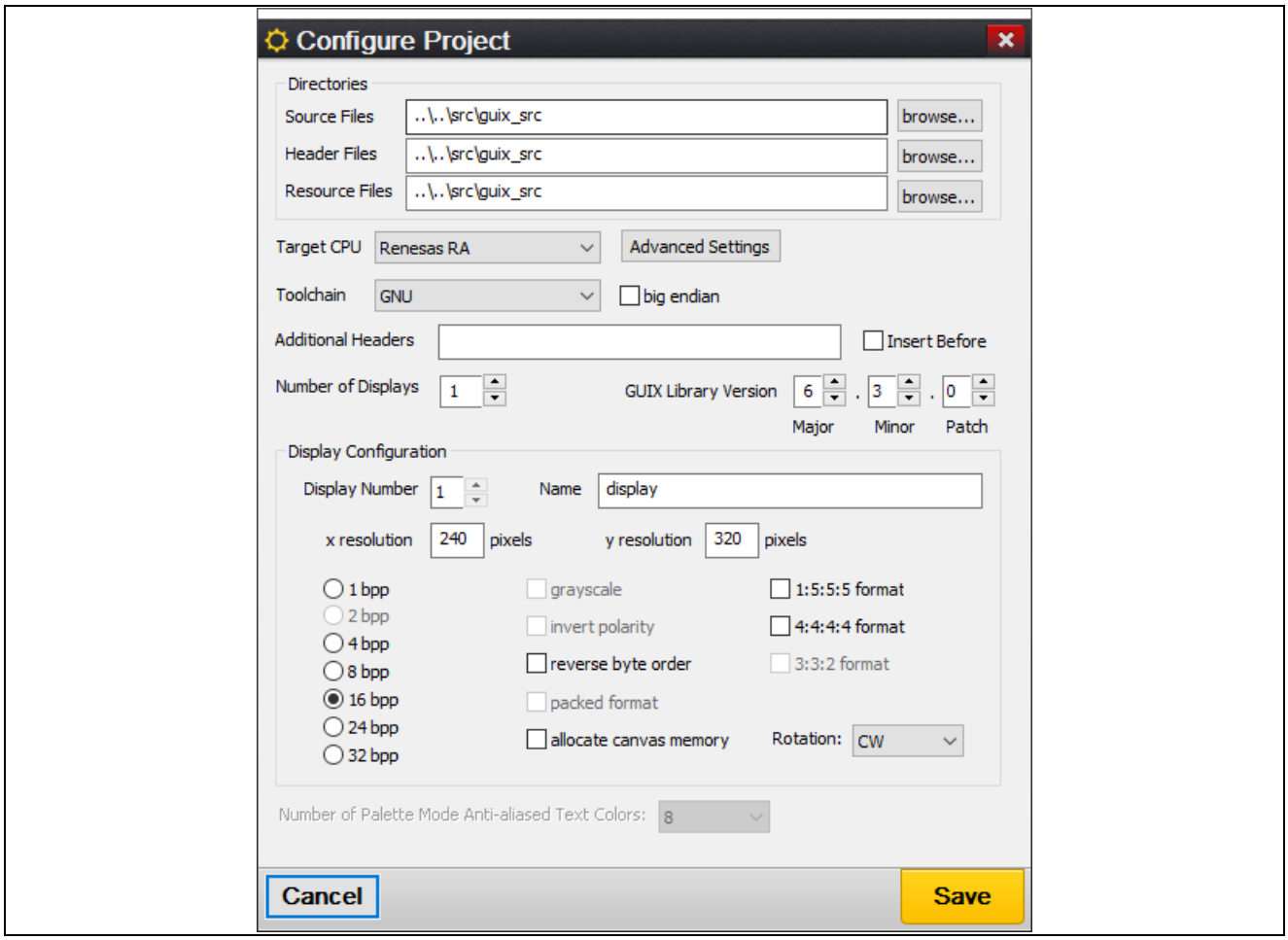


Figure 17. Configure Project window enables setting the project and/or display settings

The source files, header files, and resource file paths are specified relative to the .gxp project's location in the e^2 studio project file system, and are generated under the *<project>\src\guix_gen* folder. The target CPU is Renesas RA, and the Toolchain is set to GNU, which corresponds to the e^2 studio's project settings. The Display Configuration settings are based on the specifications of the NHD LCD (outlined in Section 4 The NHD LCD Controller), so the display resolution is set to 240x320 pixels with 16 bits representing each pixel. The rotation is set to CW, meaning 90^0 clockwise, to setup a landscape design on a portrait LCD.

Any time you make edits you want to keep, be sure to click save before closing the Project Configurations.

Note: Please refer to Section 7 for the details regarding the DMAC rotation version of the application project, including how the GUIX project configuration differs from the above.

3.2.3 Adding a Custom Pixelmap

The **Resource View** window is used to manage the project resources, like colors, fonts, pixelmaps, and strings, for each of the defined displays in the **Project View**.

Custom pixelmaps can be added to projects with this window using the Pixelmap tab. In this tab there are 2 folders called System and Custom which contain default GUIX pixelmaps and user-added pixelmaps, respectively.

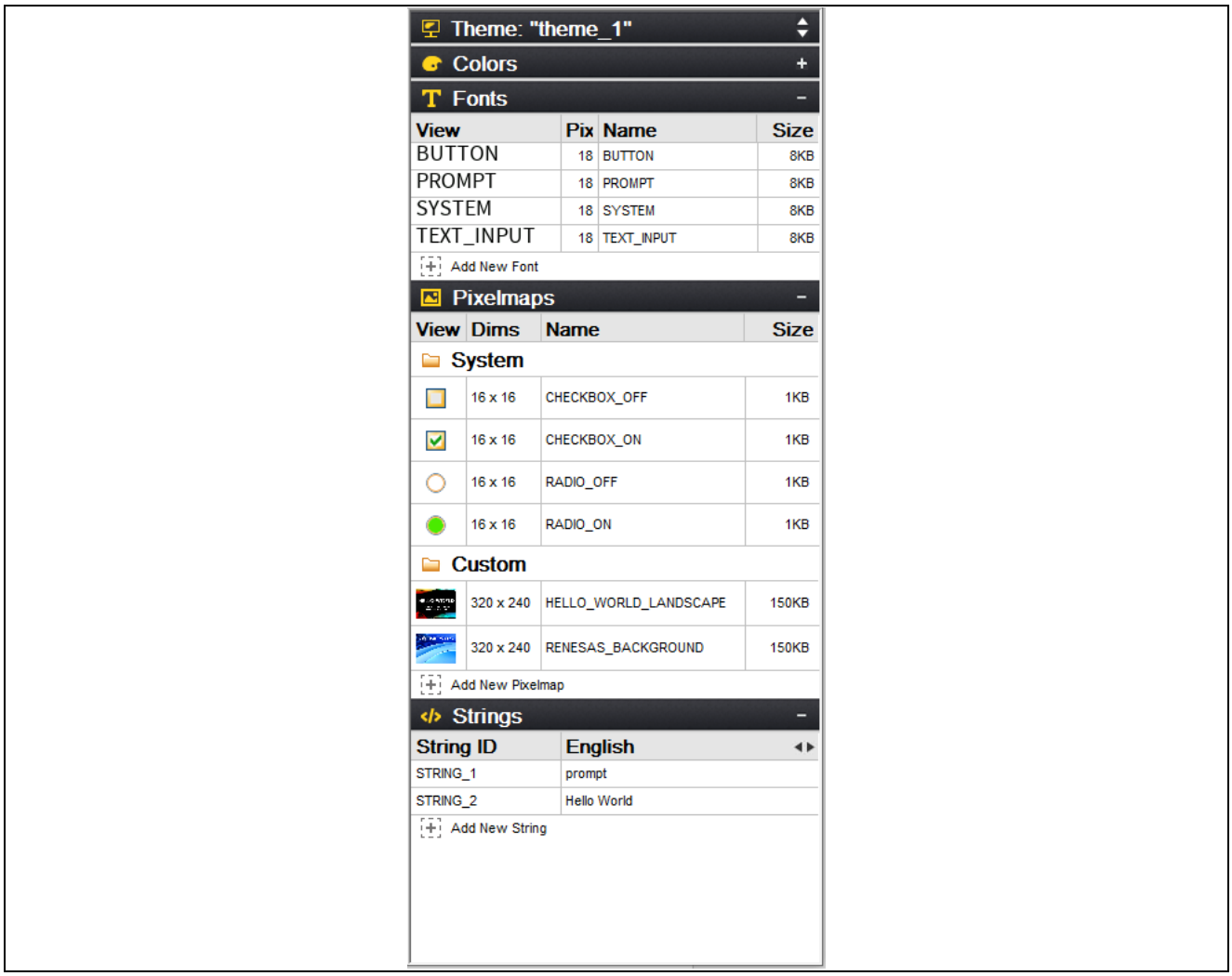


Figure 18. The Resource View manages the project's system and custom resources

To add a new custom pixelmap, click on the **Pixelmaps tab > Custom folder > Add New Pixelmap** button, and the Select Image File(s) window will let you browse for the image that you would like to import to GUIX Studio.

If you need to modify a pixelmap resource to match an application's UI needs, double-click on the pixelmap resource in question to bring up the Edit Pixelmap dialog. From here, you can specify properties like the source image's path, the pixelmap's ID name, the output file(s) where GUIX will generate the pixelmap data for use with the GUIX library, and the output format of the pixelmap data.

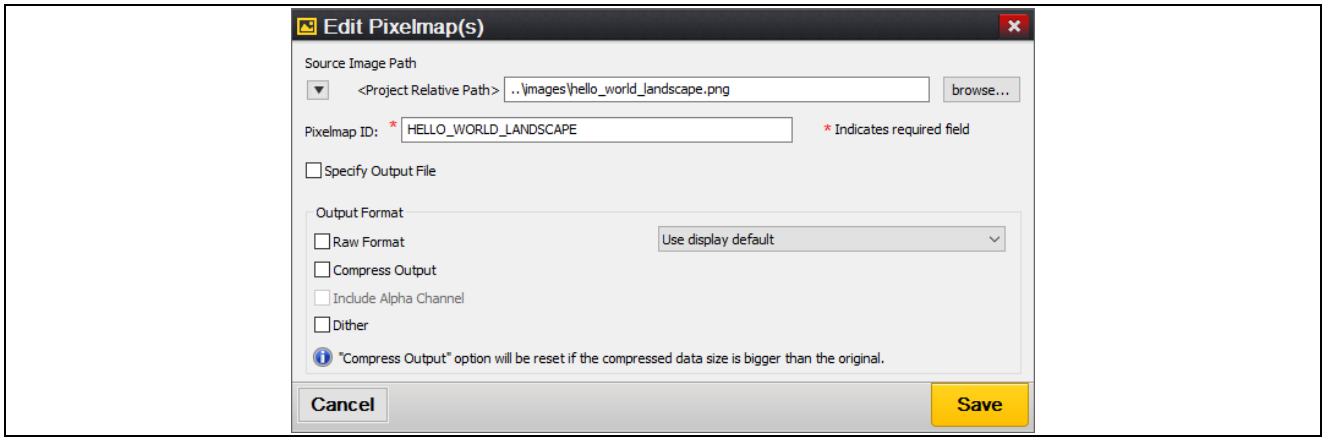


Figure 19. Edit Pixelmap window modifies the content of the selected pixelmap resource

Once all options are set as desired, click the **OK** button to produce a new pixelmap resource. GUIX Studio will read the input image file, decompress it, perform color space conversion and dithering, optionally re-compress the data, and save the data in GUIX compatible `GX_PIXELMAP` format. The new pixelmap is added to the project resources and made available for the application to use.

For the Hello World project, there are 2 images titled “hello_world_landscape” and “renesas_background”, which are used as full display images for the hello_world and renesas_bg windows, respectively. The window images were created with other software and are saved as PNGs inside the e^2 studio project’s file system under the *<project>\guix_studio\images* folder. Their output format is set to “Use display default” to match the 565 RGB color format specified by the NHD LCD.

3.2.4 Configuring the Screen Flow

The key to a comprehensive GUI application is creating events from user inputs that trigger the execution of various transitions and changes to content being displayed. GUIX Studio supports automatic generation and execution of screen transition logic, which is defined by the user through the Screen Flow configuration. With Screen Flow, you can define what event type(s) will trigger an action, and then define the specific actions that will be executed when that trigger event is received.

When a screen flow diagram is added to the project, it enables two important features: 1) The application can be executed from within the Studio environment and 2) GUIX Studio automatically generates event handlers and screen transition logic to implement the designated screen flow within the generated `specifications.c` file, removing this effort from the application program.

To create and edit a current project’s graphical screen flow diagram, click the menu option **Configure** > **Screen Flow**.

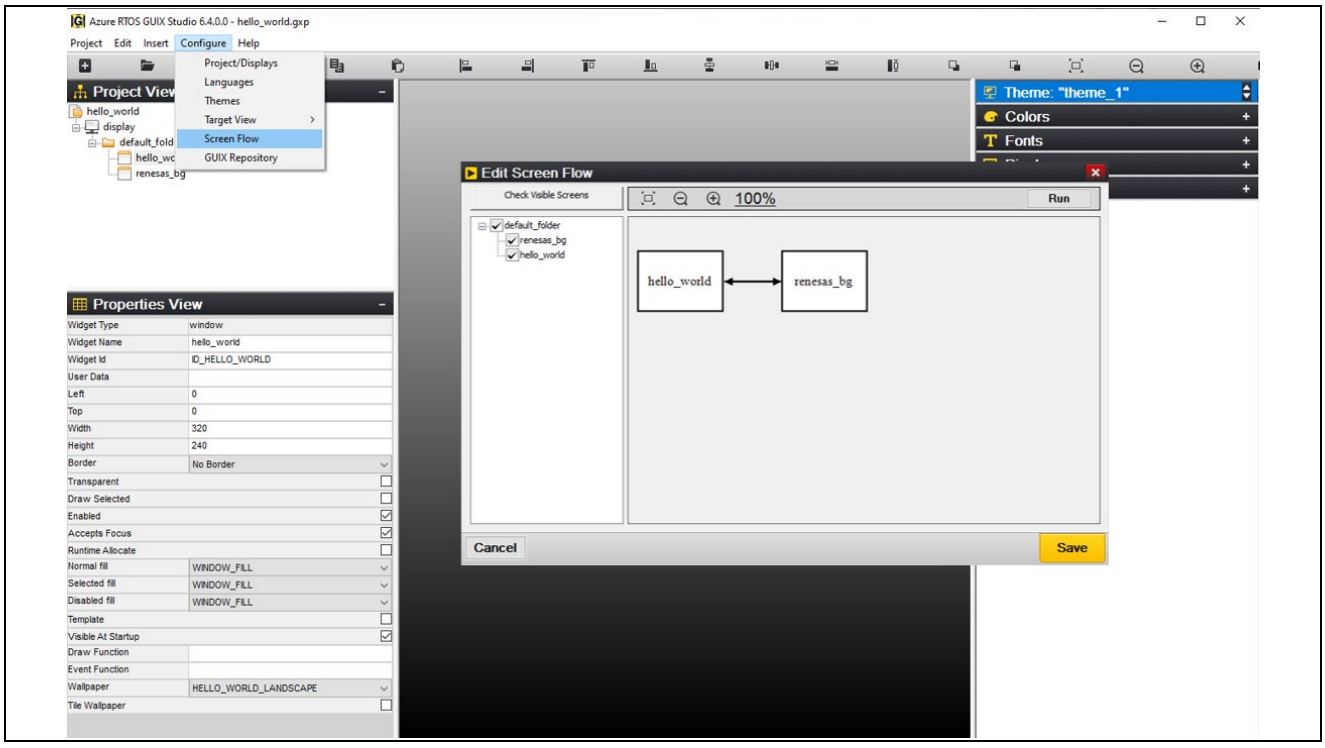


Figure 20. Screen Flow Diagram of Hello World GUI

In the screen flow diagram, right click on the screen you want to define a new trigger event for, and that will bring up the Edit Trigger List prompt for the selected screen. From this window you can add, delete, and edit trigger event types, and then edit the associated actions defined for each event trigger.

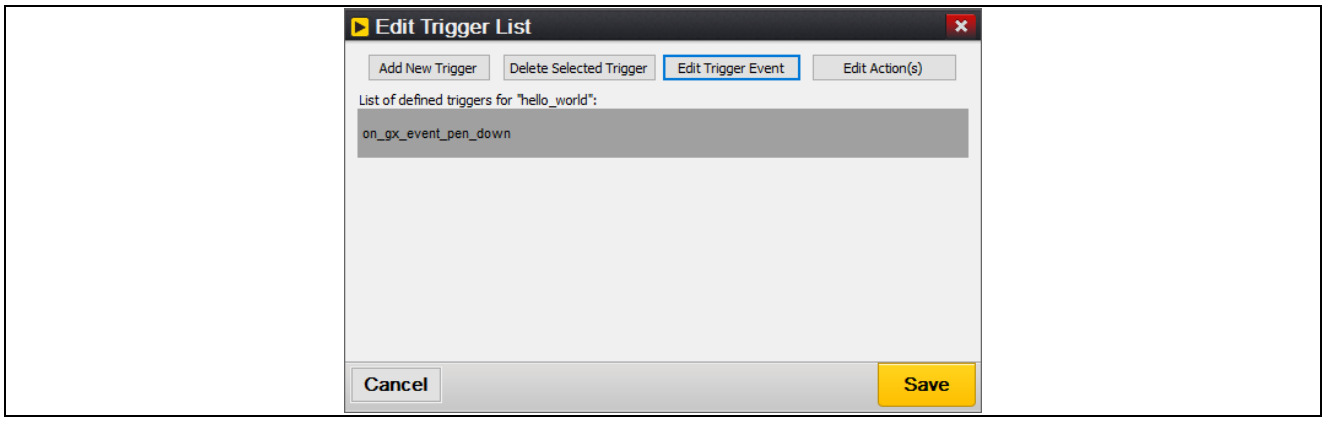


Figure 21. Edit Trigger List window for the hello_world screen

The button **Add New Trigger** lets you first define the event type screen triggers, and the button **Edit Trigger Event** can be used later to redefine the list of defined triggers.

Figure 22. The defined system trigger event for the hello_world screen is GX_EVENT_PEN_DOWN

After defining the triggers for the screen, then you can define a unique set of actions to be executed by GUIX when each trigger event occurs. First, select the trigger from the Edit Trigger List prompt and then click the **Edit Action(s)** button. This will bring up the Edit Actions for Trigger <trigger event type> window as shown in Figure 23. From the window you can define any number of actions, set up the actions' parameters, and give them meaningful names.

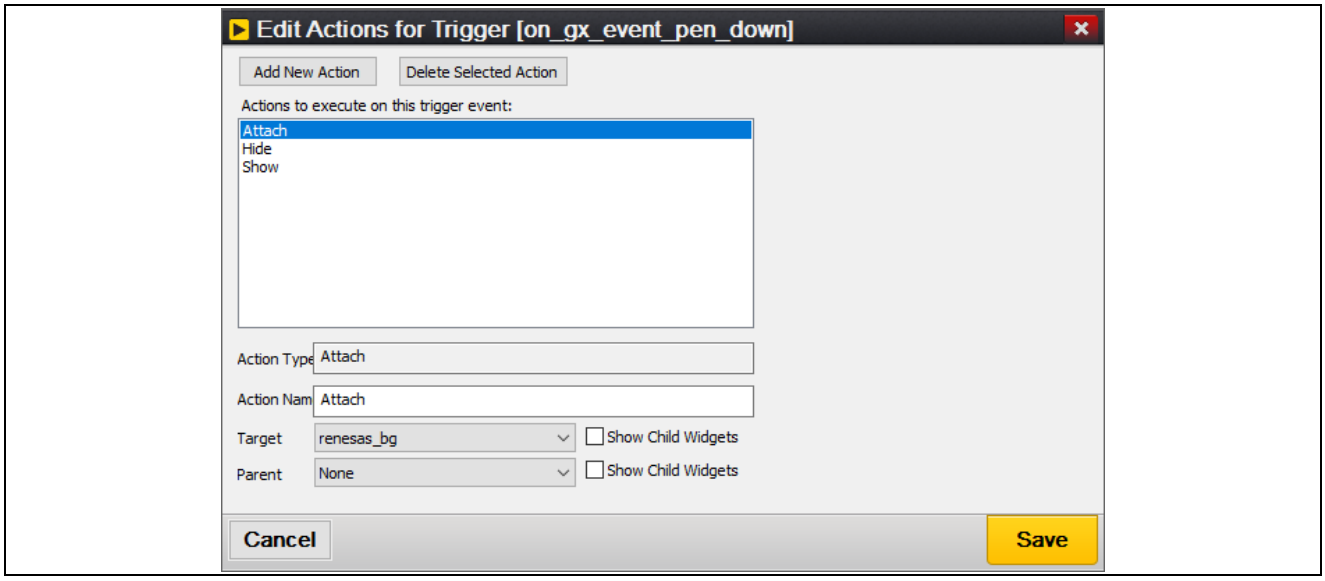


Figure 23. The defined actions will occur on GX_EVENT_PEN_DOWN

After saving your screens' defined actions and triggers, you can preview and run your application within GUIX Studio by selecting the Run Application toolbar button  . Or by navigating to **Configure > Screen Flow > Run**. When you run your application, you will see the screen(s) you have designated as "Visible At

Startup" display within a new window. The child widgets on these screens are fully operational. You can click on buttons, operate sliders and scroll wheels, etc.

Screen Flow is a useful tool for designing and creating a full GUI with screen transition logic, before you even write any of the target application code. Running the application on your desktop from within the Studio environment is a time-saving feature because you are not required to go through a compile/link cycle to execute your application. However, please note that custom drawing functions, custom event handlers, and complex event handling are not available when running the application from within the GUIX Studio environment.

3.2.4.1 The Hello World Project's Screen Flow

In the *hello_world.gxp* project, the events "pen up" and "pen down" will trigger the screens to toggle between the Hello World screen (hello_world) and the Renesas Background (renesas_bg) screen. The hello_world screen is the startup display, so in the Properties View window, the option Visible at Startup is checked.

For the hello_world screen, we want the defined event `GX_EVENT_PEN_DOWN` to trigger the display to transition to the renesas_bg screen. When the "pen down" event occurs, the defined actions for GUIX to execute are: attach the renesas_bg screen, hide the current hello_world screen, and show the renesas_bg screen.

In a similar and opposite fashion, the renesas_bg screen has the defined event `GX_EVENT_PEN_UP` which triggers a transition back to the hello_world screen. When a "pen up" event occurs, the defined actions for GUIX to execute are: attach the hello_world screen, hide the current renesas_bg screen, and show the hello_world screen.

Pressing the Run button from the Edit Screen Flow window will simulate the GUI application. You should observe that when you perform a left click down on the mouse (`GX_EVENT_PEN_DOWN`) the display toggles to the Renesas Background screen. When you release the left click button on the mouse (`GX_EVENT_PEN_UP`) the display toggles back to the Hello World screen.

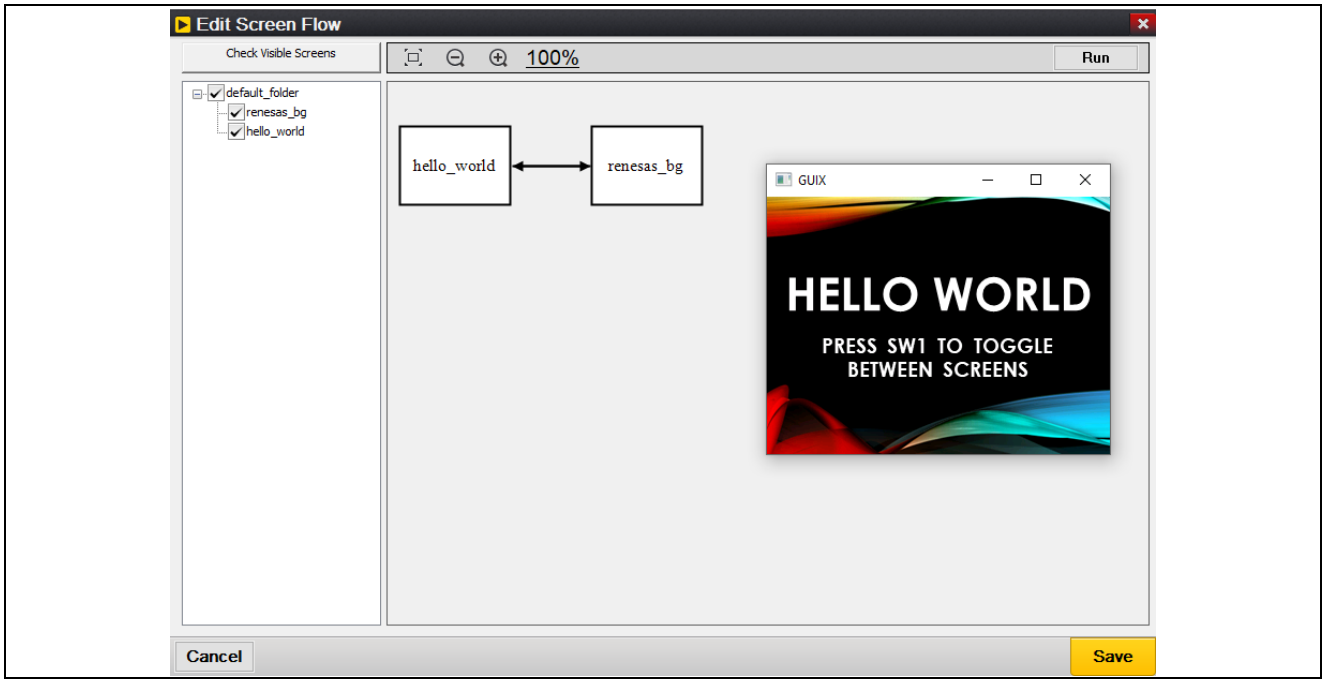


Figure 24. Simulating the hello_world.gxp GUI using Screen Flow

3.3 Integrating a GUIX Project with an e² studio Project

Developers can use the GUIX Studio PC software to design and simulate GUIs for custom embedded graphic applications. GUIX Studio will generate the designs' source code files, which in turn can be integrated into an e² studio graphics application targeting any RA MCU. The GUI designs can be edited, compiled, run, re-compiled and re-run on your RA device for testing and evaluation purposes without needing to change any e² studio application code.

The first section will explain how to set up the e² studio project's environment for running a GUIX Studio GUI application. Then the following section will detail the steps for importing GUIX's generated design source code files into the e² studio workspace.

3.3.1 Setup e² studio Project to Support GUIX Project

GUI applications designed with the GUIX Studio PC software can be run on RA MCUs by using the ThreadX Real-Time Operating System (RTOS) and the Azure RTOS GUIX Library. The ThreadX RTOS can be configured during the initial e² studio project's setup menu using the drop-down option.

The GUIX Library can be added in 2 different ways, depending on whether the MCU has the GLCDC and graphics framework hardware or it does not.

(1) RA Devices Natively Supporting GUIX

If the GLCDC and supporting graphics hardware are present, then navigate to the **FSP Configuration > Stacks** tab and use the **New Stack > Graphics > Azure RTOS GUIX** to select and add the module and library to your e² studio project.

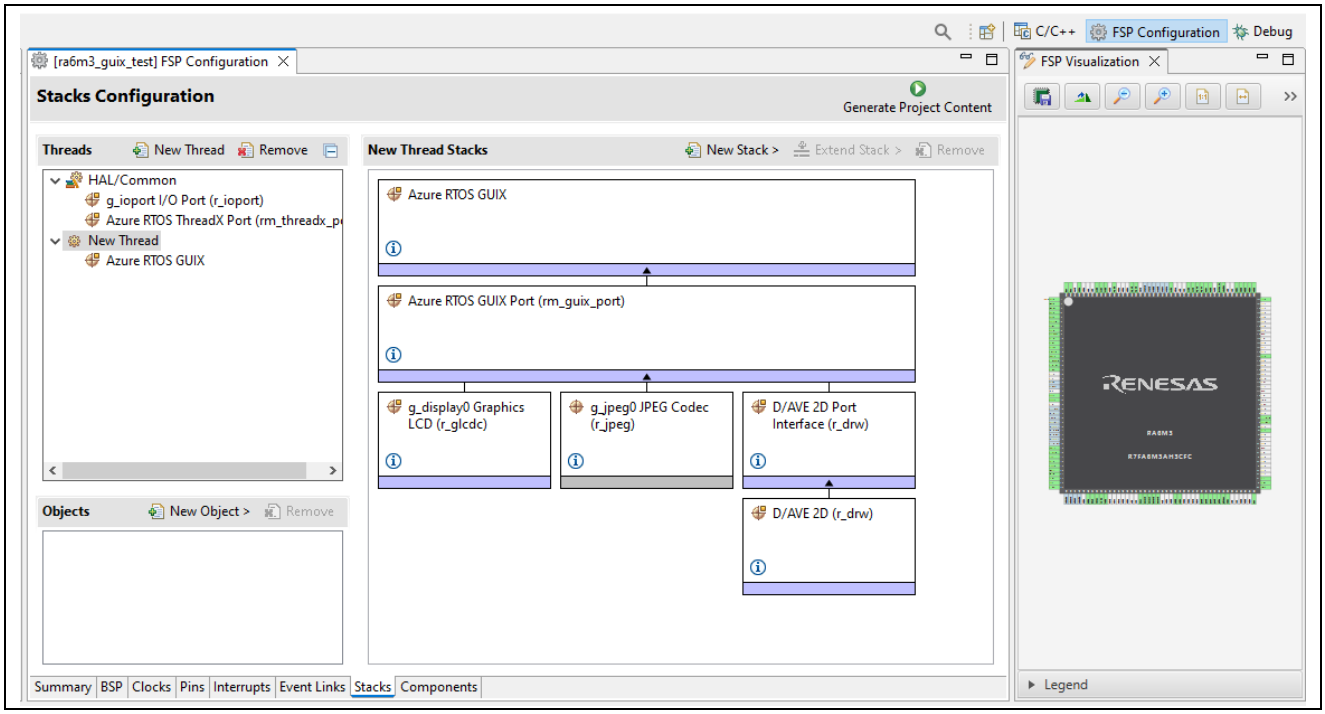


Figure 25. Adding the Azure RTOS GUIX Library and Middleware Port Stacks on MCUs with the supporting GLCDC hardware

(2) RA Devices Not Natively Supporting GUIX

The GUIX FSP Stack is not natively supported for RA Devices that lack the GLCDC and graphics framework modules on them, which is the case with the RA6M4 MCU family. Instead, the Azure RTOS GUIX Library can be manually added to a project using the **FSP Configuration View > Components** tab.

There are 2 important components to select for all necessary files in the GUIX library to be to the project:

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

NOTE: If you have multiple FSP versions installed in your e² studio instance, make sure you select the component version that corresponds to the project's FSP version.

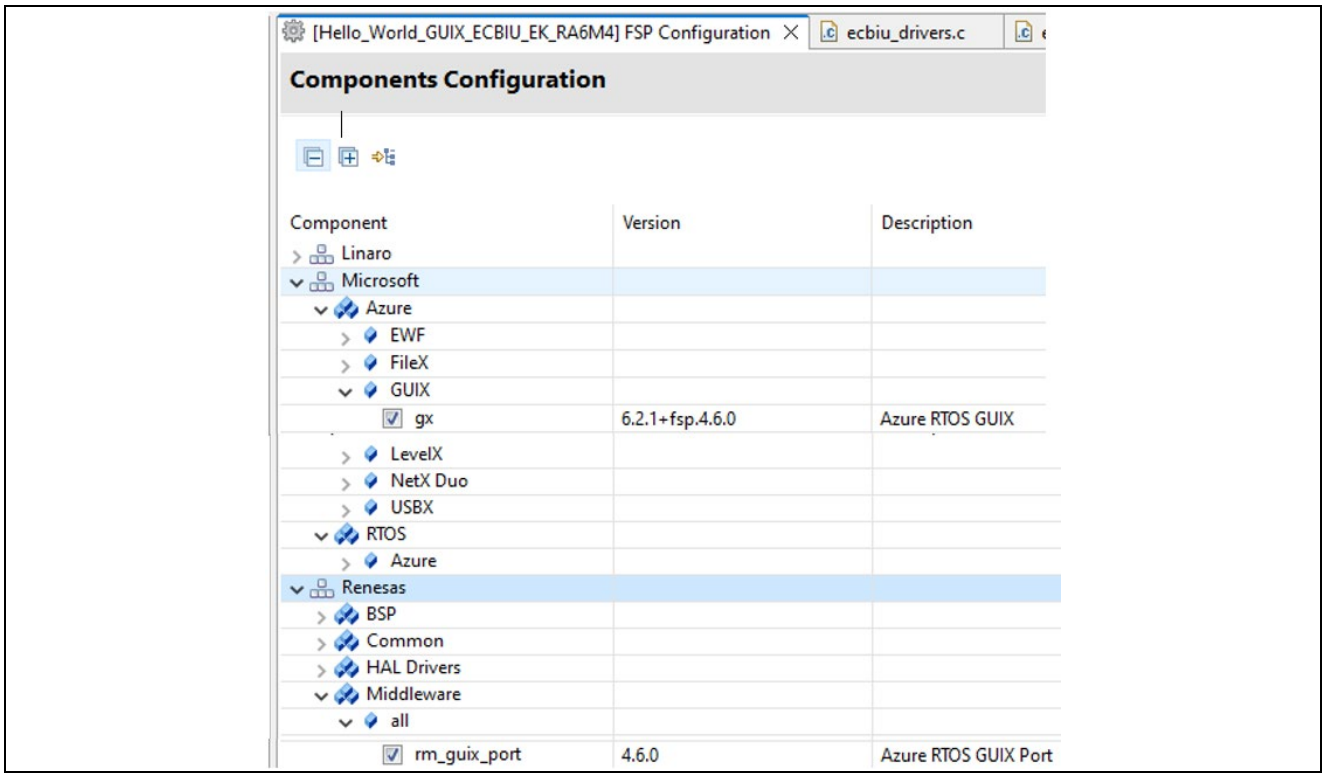


Figure 26. Adding the GUIX Library Component on MCUs without GUIX stack support

The **rm_guix_port** component includes files that reference a GLCDC module and a Dave2D drawing engine, both of which are NOT hardware resources present on the RA6M4. Therefore, the files referencing the absent modules need to be excluded from the project build to avoid build errors. Follow the steps below to do this:

After clicking “Generate Project Content” in the FSP Configuration view, the **rm_guix_port** middleware source files can be found under the project directory *<project_name>\ra\fsp\src\rm_guix_port*. Only the `gx_port.h` file in this folder is needed.

To exclude the other three extraneous files `gx_display_driver_dave2d_8bit_palette.c`, `gx_display_driver_dave2d.c`, and `rm_guix_port.c` right click on each and select **Resource Configurations > Exclude from Build…** Once a file is successfully removed from the e² studio project build, it will appear grayed out in the Project Explorer as shown below.

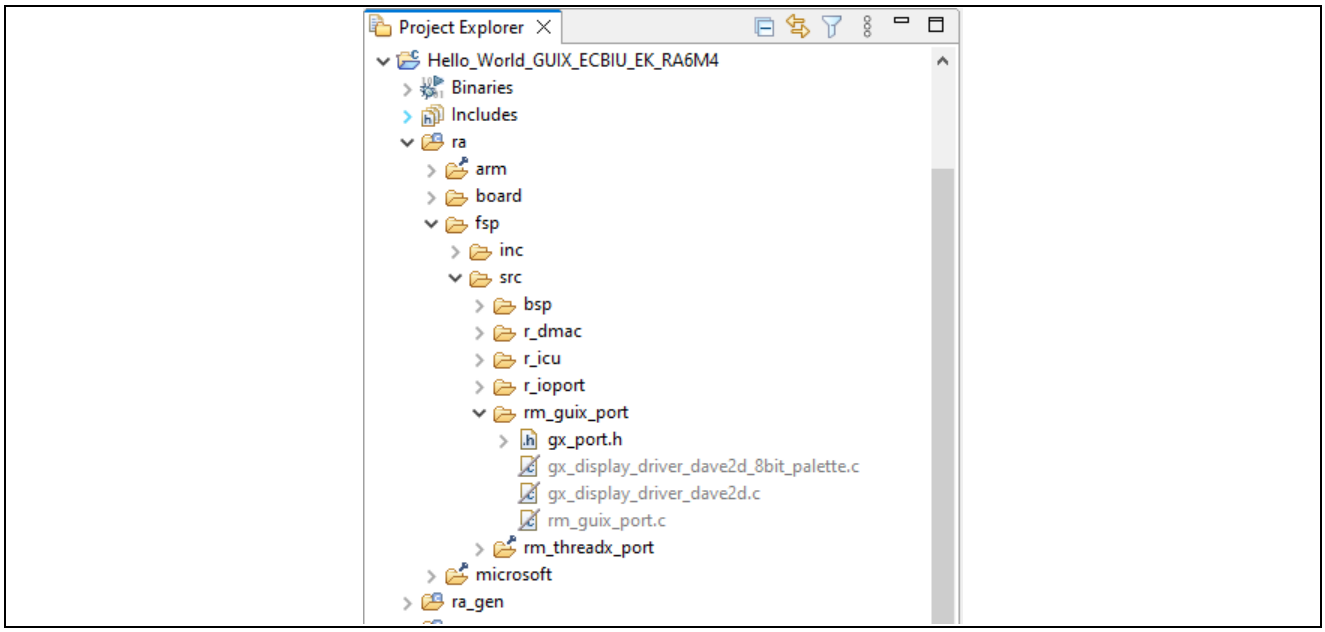


Figure 27. Example of excluding extraneous GUIX files in the Hello World e² studio project

At this point, the project environment should include all dependencies required for running the GUI application on the RA device of choice, and the GUIX project source code can be imported.

3.3.2 Add GUIX Generated Code as e² studio Source Code

When you are done editing the screens and resources in the GUIX project and the design is ready for execution on the target RA MCU, GUIX Studio can generate C code source files that contain all the necessary UI information and supporting functions to run the graphics application.

The resource files generated contain preset data structures that define all of the resources that are defined in the GUIX project. (colors, fonts, pixelmaps, and strings). These resource files can be generated in source code or binary forms. By default, there are two files generated; one file is a standard C source code file, and the other is a C header file that provides external references and constants that are necessary for the application code to access the GUIX resources defined in the project. The file names have the following convention:

- `{project-name}_resources.h`
- `{project-name}_resources.c`

The specification files generated contain all the C code to create the designed UI. This code also references the resource files generated for this project. The user's application code will make calls to the specification code, to create the UI objects defined in the project. Also, the user's application code contains all custom widget drawing, event handling, and memory allocation functions specified in the project. By default, there are two files generated, one file is a standard C source code file, and the other is a C header file that provides external references and constants that are necessary for the application code to access the GUIX Studio project specifications. The file names have the following convention:

- `{project-name}_specifications.h`
- `{project-name}_specifications.c`

Follow the steps below to generate these GUIX project source code files and integrate them into your e² studio project workspace:

1. Create a folder in e² studio under the project's *src* folder to hold the application code generated by GUIX Studio. A descriptive name like "guix_gen" or "guix_src" will help.
2. Ensure that the Project Configurations for the GUIX project point to the e²studio folder you just created. To do this, go to **Configure Project > Directories** and set the **source files**, **header files**, and **resource files** to all point to relative to the e² studio project's workspace path: **.\src\<folder_name>**.

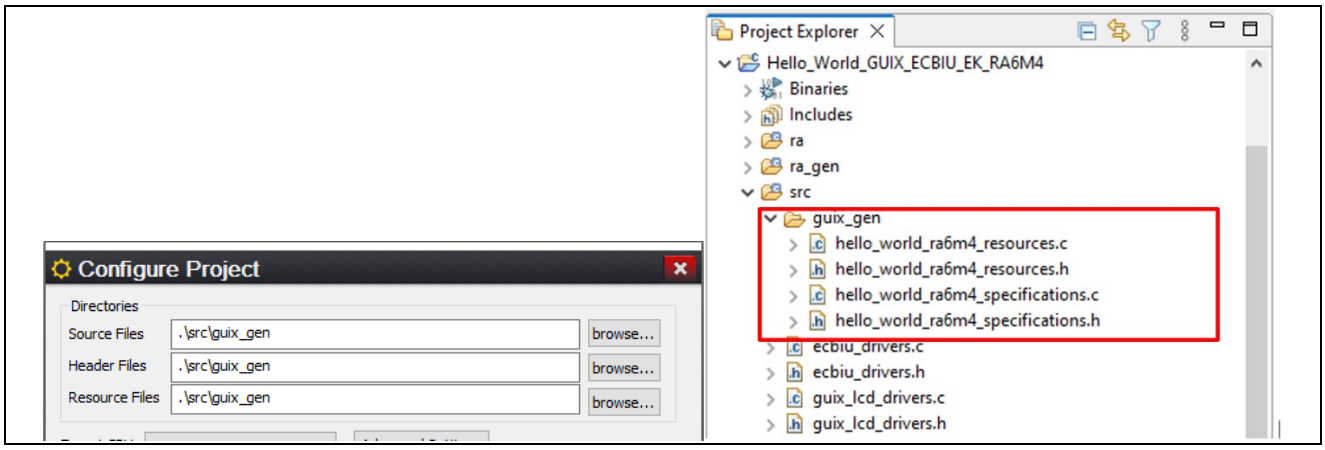


Figure 28. In the Hello World project, the GUIX generated files are in the src\guix_gen folder

It's recommended to place the GUIX generated code folder inside of the *src* folder in the e² studio project, because all files inside the *src* folder are automatically recognized as source code files when compiling the project.

4. The NHD LCD Controller

The Hello World graphics application project accompanying this document demonstrates using the external memory bus interface on the EK-RA6M4 to drive the NewHaven Display 2.8" Liquid Crystal Display (LCD) Controller (part no. NHD-2.8-240320AF-CSXP-F). A 40-pin FFC connector breakout board (part no. NHD-FFC40) is used to connect the RA6M4 MCU header pins more easily to the LCD controller's 40-pin flexible flat cable.

This section provides a brief introduction to the NewHaven Display LCD controller used in the design, linking important specification documents and providing an overview of the LCD controller's functionality. While certain application details in this application note must be tailored to the specifications of this LCD controller, the general process described for integrating and driving an LCD can be extrapolated to driving other LCDs, peripherals, or memory devices.

Section 7 Understanding the Application Project's Key Mechanisms provides an in-depth analysis of operating the external memory bus within the specifications of this LCD controller.

4.1 NHD LCD References

The NHD-2.8-240320AF-CSXP-F part is listed on the NewHaven Display website at:
https://newhavendisplay.com/2-8-inch-ips-tft-without-touchscreen/

The 40-pin FFC connector breakout board is listed on the NewHaven Display website at:
https://newhavendisplay.com/40-pin-0-5mm-pitch-ffc-connector-breakout-board/

Specifications for the LCD are linked on the above website page, and added here for convenience:
https://newhavendisplay.com/content/specs/NHD-2.8-240320AF-CSXP-F.pdf

The LCD has a built-in Sitronix ST7789Vi controller. The full specifications of the ST7789V controller are available for download at: https://support.newhavendisplay.com/hc/en-us/articles/4414907838487-ST7789V

4.2 Overview of NHD LCD Specifications

Table 7 lists important technical specifications of the NHD LCD controller. Other chip-on-glass controllers will have properties that may vary in value from those specified for the NHD LCD, so readers should carefully review the specifications for the details of operation.

Table 7. Specifications of the NHD-2.8-240320AF-CSXP-F

Specification	Value
Display Type	TFT – Color (In-Plane Switching)
Display Mode	Transmissive
Graphics Color	Red, Green, Blue (RGB)

Touchscreen	None
Diagonal Screen Size	2.8”
Resolution	240x320
Interface	8080-II series parallel, 8-bit/16-bit
Controller Type	ST7789Vi
Connection Type	40 pin FFC
Supply Voltage	2.8V

4.3 NHD LCD Controller Timing Cycles

The NHD LCD controller interfaces to the MCU with the Intel 8080-II series parallel bus communication protocol. The 8080 protocol consists of 8 or 16-bit bi-directional data lines, one chip-select line, one active low writing-latch line (WR), one active low reading-latch line (RD) and one data/command select line.

This section illustrates the relevant timing cycle specifications for the 8080-II interface, which are taken from the ST7789Vi controller specifications.

4.3.1 Parallel 16/8-bit Read and Write Cycle Timing

Figure 29 illustrates the timing requirements for a read and write cycle of the 8080-II parallel interface and Table 8 provides the timing specifications for each signal in the diagram.

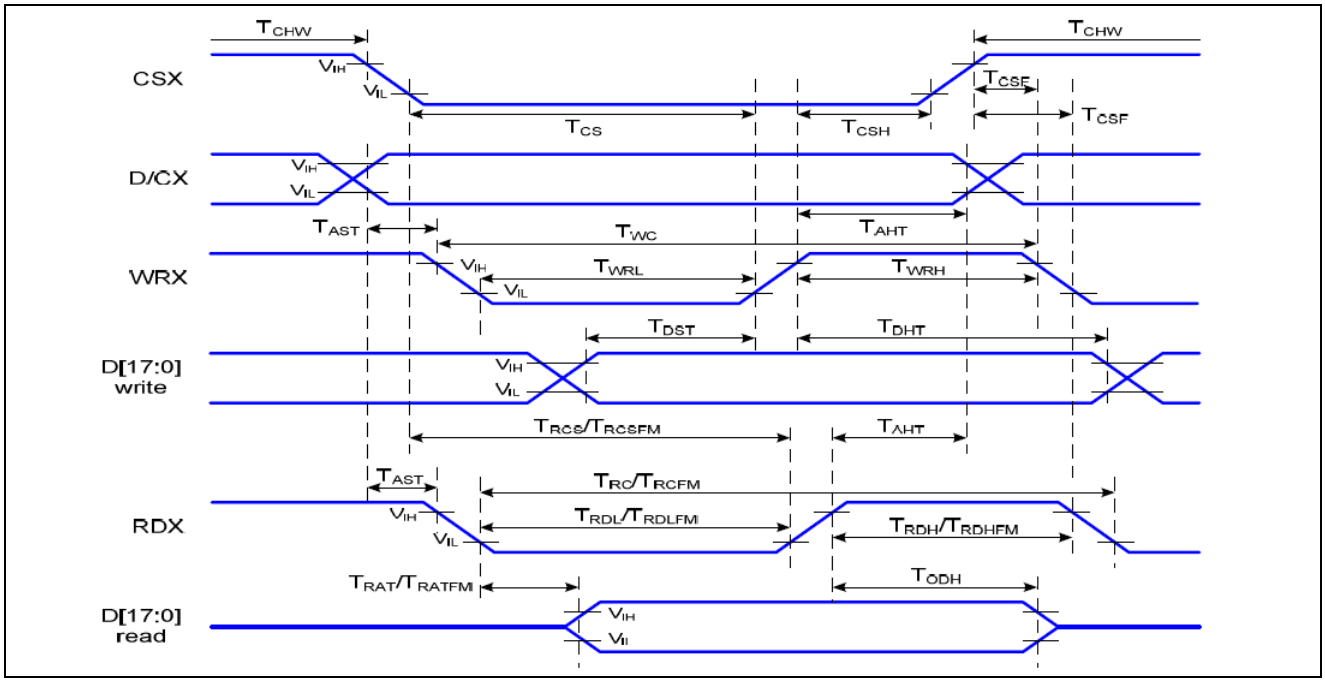


Figure 29. Parallel interface timing characteristics (8080-II Series MCU interface)

Table 8. Timing specifications for each signal and symbol in the above diagram

Signal	Symbol	Parameter	Min	Max	Unit	Description
D/CX	T_{AST}	Address setup time	0		ns	-
	T_{AHT}	Address hold time (Write/Read)	10		ns	
CSX	T_{CHW}	Chip select "H" pulse width	0		ns	-
	T_{CS}	Chip select setup time (Write)	15		ns	
	T_{RCS}	Chip select setup time (Read ID)	45		ns	
	T_{RCSFM}	Chip select setup time (Read FM)	355		ns	
	T_{CSF}	Chip select wait time (Write/Read)	10		ns	
	T_{CSH}	Chip select hold time	10		ns	
WRX	T_{WC}	Write cycle	66		ns	
	T_{WRH}	Control pulse "H" duration	15		ns	
	T_{WRL}	Control pulse "L" duration	15		ns	
RDX (ID)	T_{RC}	Read cycle (ID)	160		ns	When read ID data
	T_{RDH}	Control pulse "H" duration (ID)	90		ns	
	T_{RDL}	Control pulse "L" duration (ID)	45		ns	
RDX (FM)	T_{RCFM}	Read cycle (FM)	450		ns	When read from frame memory
	T_{RDHFM}	Control pulse "H" duration (FM)	90		ns	
	T_{RDLFM}	Control pulse "L" duration (FM)	355		ns	
D[17:0]	T_{DST}	Data setup time	10		ns	For CL=30pF
	T_{DHT}	Data hold time	10		ns	
	T_{RAT}	Read access time (ID)		40	ns	
	T_{RATFM}	Read access time (FM)		340	ns	
	T_{ODH}	Output disable time	20	80	ns	

4.3.2 Power-on and Reset Cycle Timing

The following image illustrates the timing requirements for a power/off sequence and a reset sequence of NHD LCD using the 8080-II parallel interface:

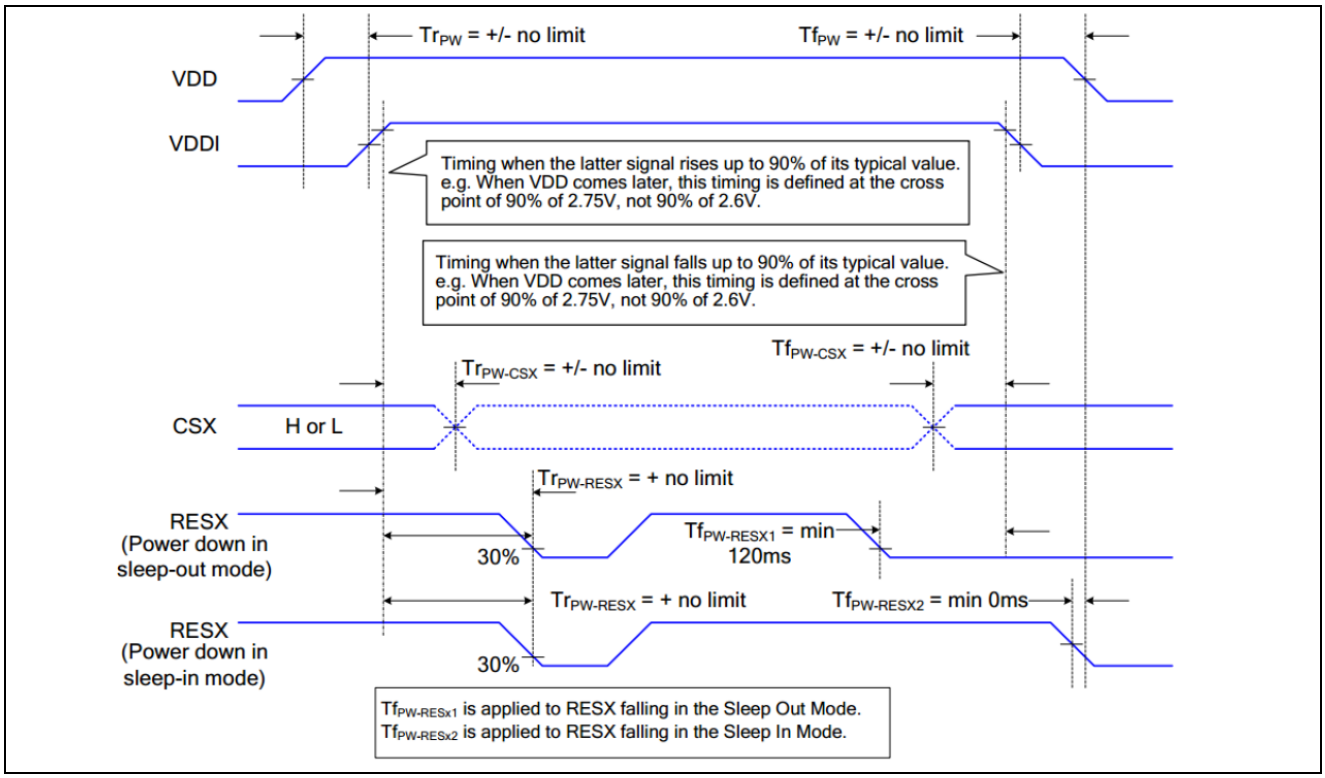
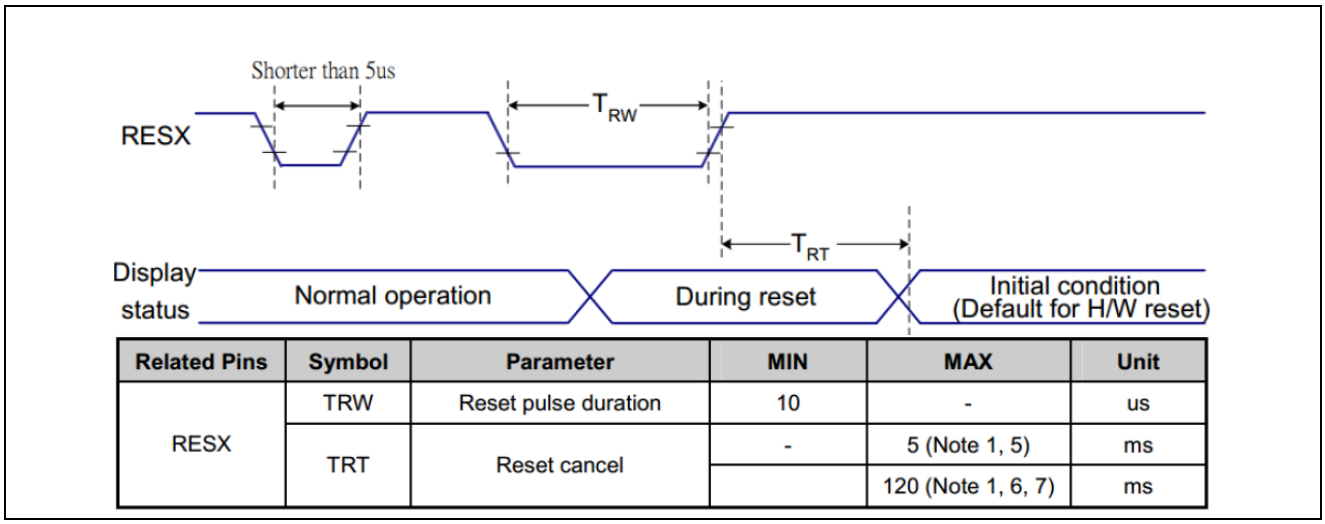


Figure 30. Power ON/OFF timing characteristics (8080-II Series MCU interface)



Related Pins	Symbol	Parameter	MIN	MAX	Unit
RESX	TRW	Reset pulse duration	10	-	us
	TRT	Reset cancel	-	5 (Note 1, 5)	ms
				120 (Note 1, 6, 7)	ms

Figure 31. Reset timing characteristics (8080-II Series MCU interface)

5. The Hello World Application

The Hello World graphics application is a simple two-screen GUI designed using Azure RTOS GUIX Studio for the EK-RA6M4 and the above-mentioned NHD LCD controller.

The GUIX Studio project *hello_world_ra6m4.gxp* demonstrates an example of creating a custom graphics GUI, which can be used as a guide for creating a GUIX Studio project for any LCD and RA MCU. The e^2 studio project *Hello_World_GUIX_ECBIU_EK_RA6M4* provides examples of external memory bus drivers, middleware between GUIX studio APIs and LCD drivers, and custom LCD drivers.

This section describes the high-level system overview of the Hello World e^2 studio project and provides instructions for verifying the example's operation on your own hardware.

5.1 Hello World Project Overview

This section illustrates from a higher level how the FSP modules, the ECBIU drivers, the NHD LCD drivers, the middleware layer between Azure RTOS GUIX APIs and the LCD drivers, and the Azure RTOS GUIX

interface framework operate together to create a synergistic ThreadX-based graphical application targeting the EK-RA6M4.

The Hello World application consists of 2 screens which are toggled by a user pressing a hardware button on the MCU. When the program begins the ECBIU, FSP modules, GUIX system, and LCD are all initialized, and a Hello World Screen is displayed on the screen. Pressing the hardware button SW1 on the EK-RA6M4 triggers a GUIX event responsible for drawing the Renesas Background Screen to the display. Any subsequent presses of SW1 will toggle back and forth between the two screens.

Look at the following application block diagram to get a better idea of the complete system and how the components interact. In the diagram, a “custom” block indicates that routines and functions for the hardware drivers and middleware stacks were designed specifically for this example’s use case. The non-custom blocks have been designed for general use by the FSP or by Azure.

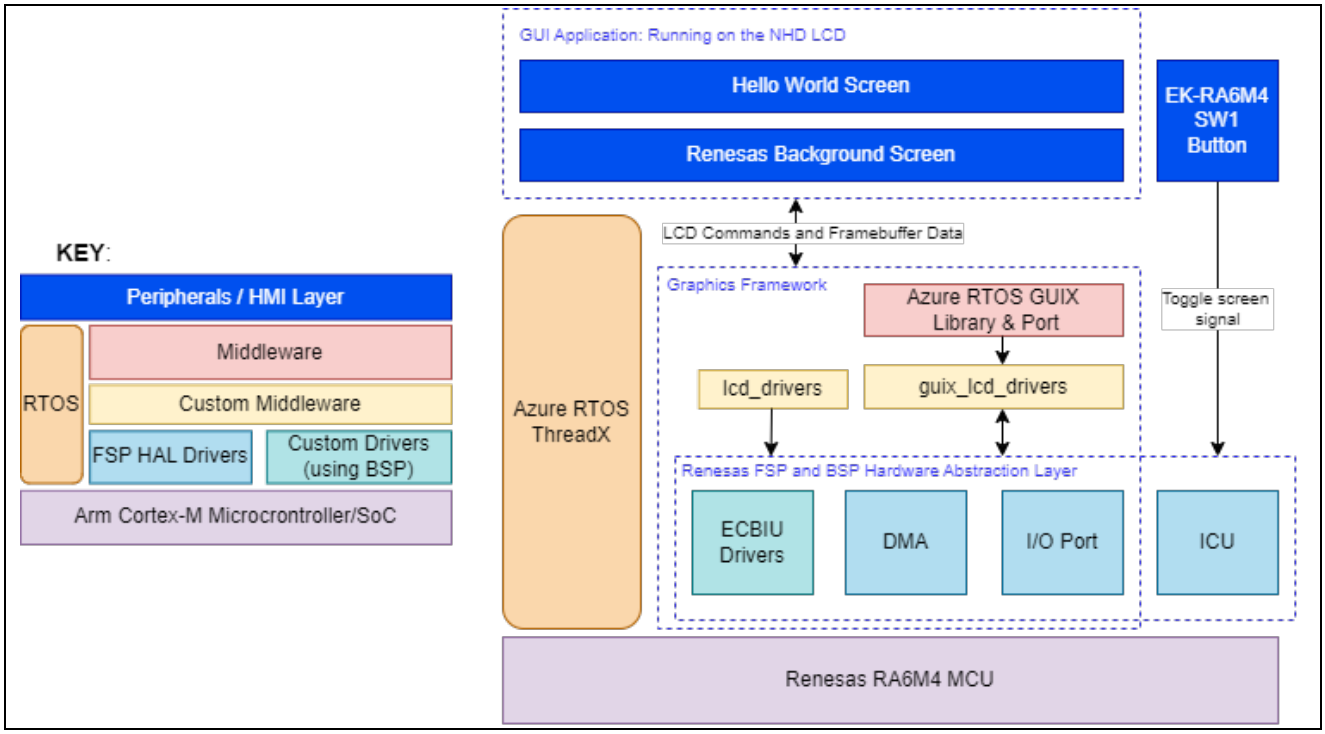


Figure 32. Application block diagram of the Hello World GUI

The graphics framework of the Hello World application on the RA6M4 is composed of the Azure RTOS GUIX Library & Port, the custom middleware, and the hardware layer consisting of the custom ECBIU drivers and the I/O Port FSP HAL drivers. The DMAC hardware module is only present with the DMAC rotation project but is shown here for illustration purposes. The graphics framework’s modules work together to send LCD commands and pixel data out the NHD LCD to display the GUI screens.

The ICU hardware layer is responsible for handling external interrupt signals coming from users interacting with the physical hardware switch SW1. All modules are supported by the Azure RTOS ThreadX kernel running on the RA6M4.

5.2 Running the Hello World Project

This section will cover the steps to get the Hello World application project up and running on the EK-RA6M4.

5.2.1 Connecting the MCU to the LCD

If you have the NHD LCD and connection hardware, this section explains how to connect the display controller to the EK-RA6M4. If you choose to run the application project without the LCD feel free to jump to the next section 5.2.2.

The table below shows the pin connections between the NHD LCD and the pins on the EK-RA6M4:

Table 9. Pin assignments of the NHD LCD and the EK-RA6M4

LCD Pin No.	LCD Symbol	MCU Pin Label	MCU Pin Assignment	Description

1	GND	GND	GND	Ground
2-6	NC	-	-	No Connect
7	VDD	VDD	VDD	Supply voltage for LCD (2.8V)
8	IOVDD	VDD	VDD	Supply voltage for logic (tie to VDD)
9	NC	-	-	No Connect
10	/CS	P610	CS0	Active low chip select signal
11	D/C	P114	A02	LCD Data/Command select / external memory bus address line 2. 0 – Command, 1 - Data
12	/WR	P601	WR/WR0	Active low write to LCD signal
13	/RD	P600	RD	Active low read from LCD signal
14	DB0	P100	D0	Bi-directional data bus signals. 8-bit: DB8-15 16-bit: DB0-15
15	DB1	P101	D1	
16	DB2	P102	D2	
17	DB3	P103	D3	
18	DB4	P104	D4	
19	DB5	P105	D5	
20	DB6	P106	D6	
21	DB7	P107	D7	
22	DB8	P612	D8	
23	DB9	P613	D9	
24	DB10	P614	D10	
25	DB11	P605	D11	
26	DB12	P604	D12	
27	DB13	P603	D13	
28	DB14	P800	D14	
29	DB15	P801	D15	
30	/RES	P113	GPIO	Active low reset signal
31	IM0	P115	GPIO	8080-II series parallel interface bus width. The ECBIU is always set to 16-bit. 0 – 16-bit, 1 – 8-bit
32	NC	-	-	No Connect
33	GND	GND	GND	Ground
33-37	LED K1-4	GND		Backlight cathode (ground)
38	LED-A	VCC		Backlight anode (100mA @3.1V)
39	GND	GND	GND	Ground
40	NC	-	-	No Connect

Note that the backlight anode pin on the LCD controller has specific signal requirements for a 100mA current supplied at 3.1 volts. For best performance it's recommended to use a signal generator or backlight control

unit to supply the specific signal for powering LCD's backlight LEDs. However, for testing and verification purposes you can supply this power using the VCC signal on the MCU, but proper operation is not guaranteed.

If you have the NHD-FFC40 adapter you will need to solder pin headers that extend from the bottom of the adapter board. Use the diagram below concurrently with Table 9 to connect the adapter board to the EK-RA6M4:

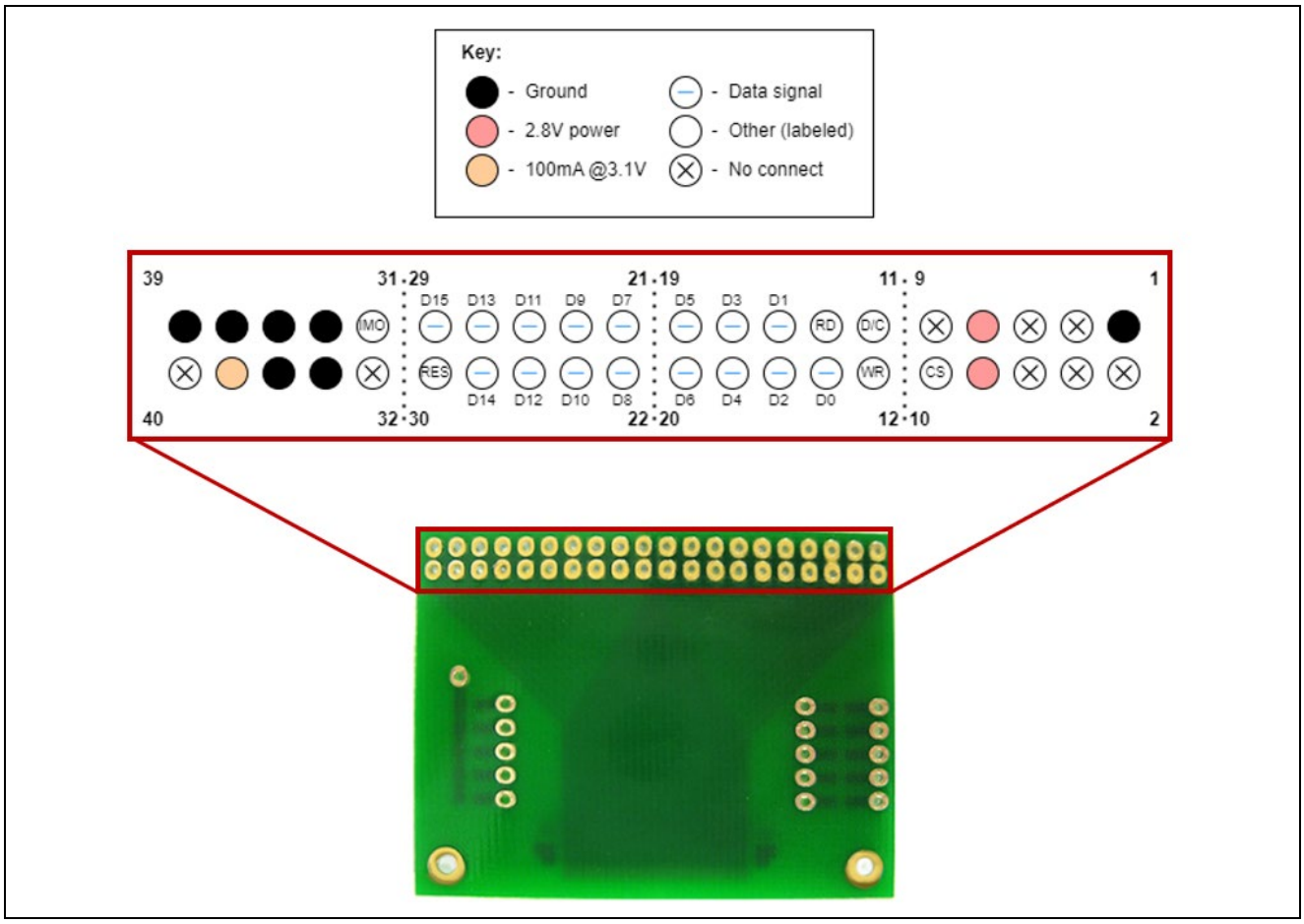


Figure 33. Diagram depicting how the LCD signals route through the FCC40 adapter

Next, attach the ribbon cable on the LCD to the front of the adapter board as shown in Figure 34:

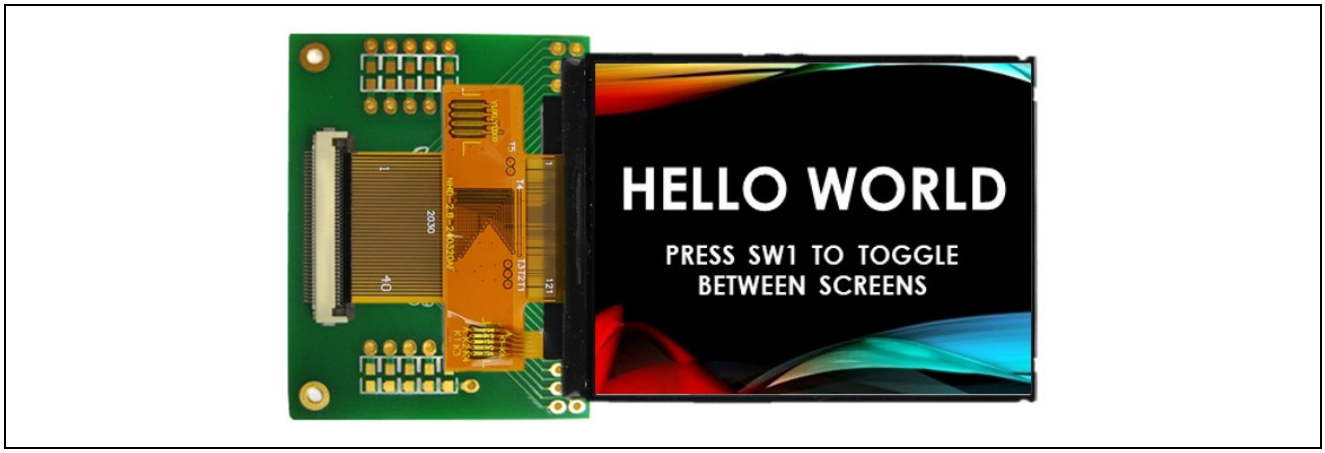


Figure 34. Connection of the LCD to the FFC40 adapter

With the LCD attached to the clamshell connector of breakout board, and the breakout board pins connected to the EK-RA6M4 pins you are now ready to verify the Hello World application.

5.2.2 Import, Build, and Run

Connect the LCD controller to the EK-RA6M4 with jumper cables according to the table in Section 5.2.1 above. You can still run the project without the LCD; see Section 5.2.3 below for viewing the framebuffer directly from memory.

The following instructions apply whether you have connected the LCD or not:

1. Ensure the application project folder, located under *r11an0813\Hello_World_GUIX_ECBIU_EK_RA6M4* is downloaded to your host machine.
2. Connect the EK-RA6M4 micro-USB debug port to your host machine with the micro-USB cable.
3. Open an instance of e^2 studio IDE.
4. In the workspace launcher, browse to the workspace location of your choice and select it.
5. In e^2 studio, navigate to **File > Import**.
6. In the Import dialog box select **General > Existing Projects into Workspace.**
7. Click **Select root directory** and use the **Browse…** button to point to the location of the *Hello_World_GUIX_ECBIU_EK_RA6M4* folder. If the folder is zipped, click **Select archive file** and use the **Browse…** button to point to the zip folder location.
8. Make sure the option **Copy projects into workspace** is selected. Click **Finish**.
9. Open the *configuration.xml* file in the Project Explorer view and click **Generate Project Content**.
10. Build the project.
11. Debug and run the project.

5.2.3 View Framebuffer Using e^2 studio's Memory View

The Memory View tab allows users to view and edit the memory presented with "memory monitors". Each monitor represents a segment of memory specified by its base address location. The memory data for each monitor can be viewed with different "memory renderings" which are predefined data formats like hex integer, signed integer, ASCII image, RAW image, etc. Memory View is only available with an active e^2 studio debug session.

The Hello World project's display functionality can be observed without the NHD LCD controller, just with the EK-RA6M4 and an active e^2 studio debug session. When your session is still running but in a paused state, you can view the current state of the framebuffer in RAM using the Memory View tab. This should give you an idea of what the display would be showing if it were connected. Use the following steps to view the `FrameBuffer` variable as an image in Memory View:

To pause the active debug session, either set breakpoints in the code or use the pause button. After the Hello World program is paused, click **Window > Show View > Memory.** Click the Add Memory Monitor button  ,enter the expression **&FrameBuffer**, and click **OK**. The default rendering of the data at the address of the `FrameBuffer` variable is in the form of hex integer values.

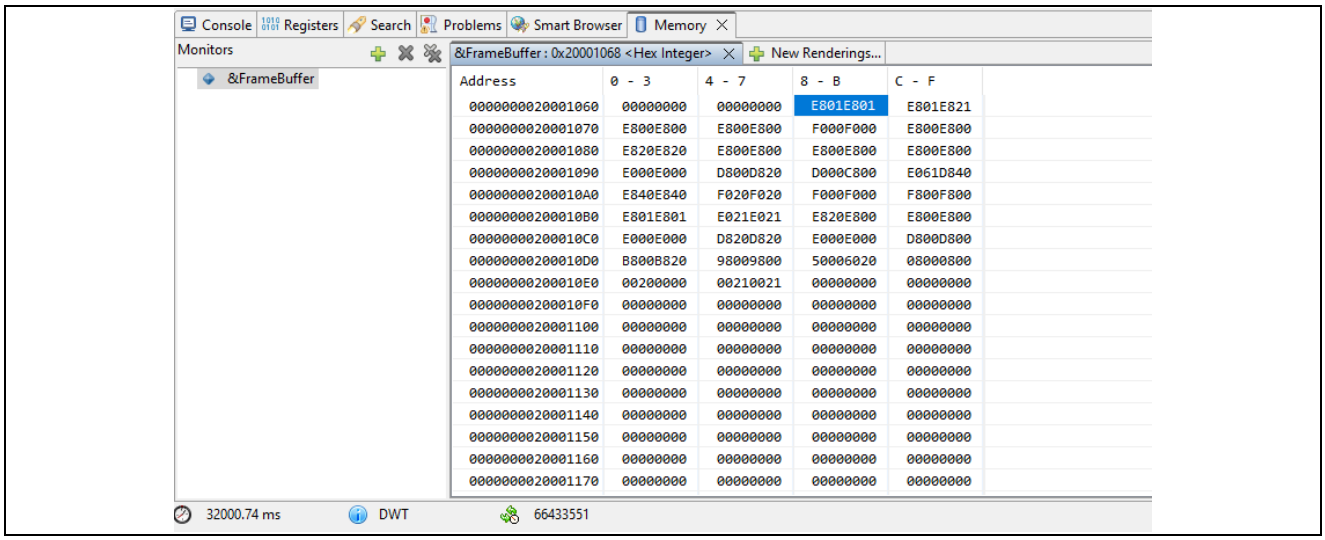


Figure 35. The FrameBuffer variable represented by hex values in the Memory View tab

To view the raw image as it would appear on a display, rather than viewing the pixels' hex values, click **New Renderings…** > **Raw Image** > **Add Rendering(s).** Set the width to 240, the height to 320 and the encoding to RGB with 16bpp (5:6:5) format. The settings should look like the image below. Click **OK**.

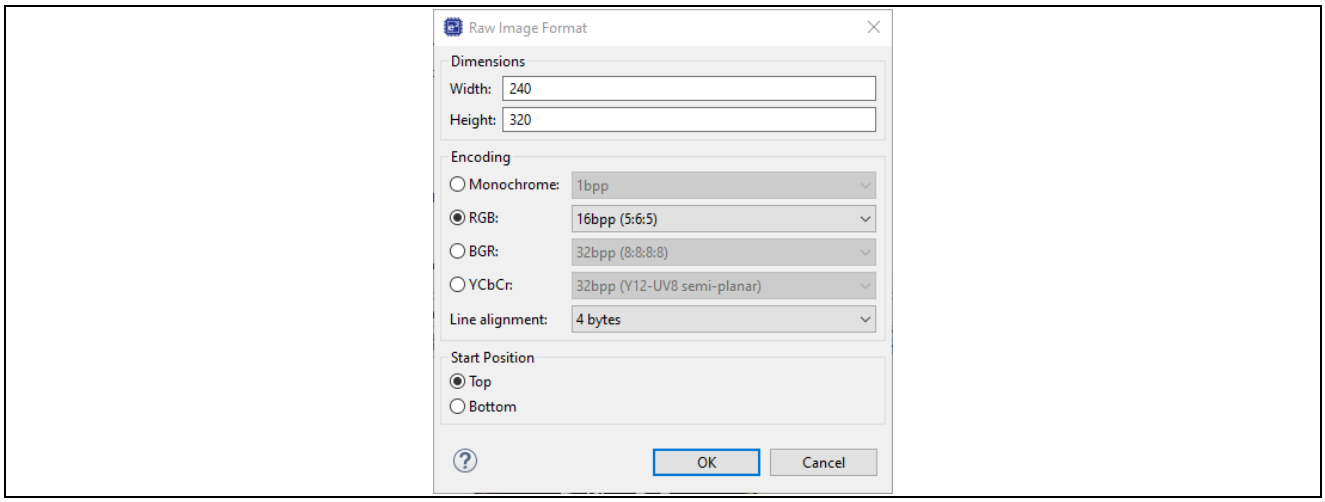


Figure 36. Raw Image Format specifications for properly viewing the FrameBuffer variable

A rendering of the current state of the `FrameBuffer` pixel array will be displayed. The image is the graphical representation of the frame buffer pixel data of the LCD screen, which is stored in RAM on the RA6M4.

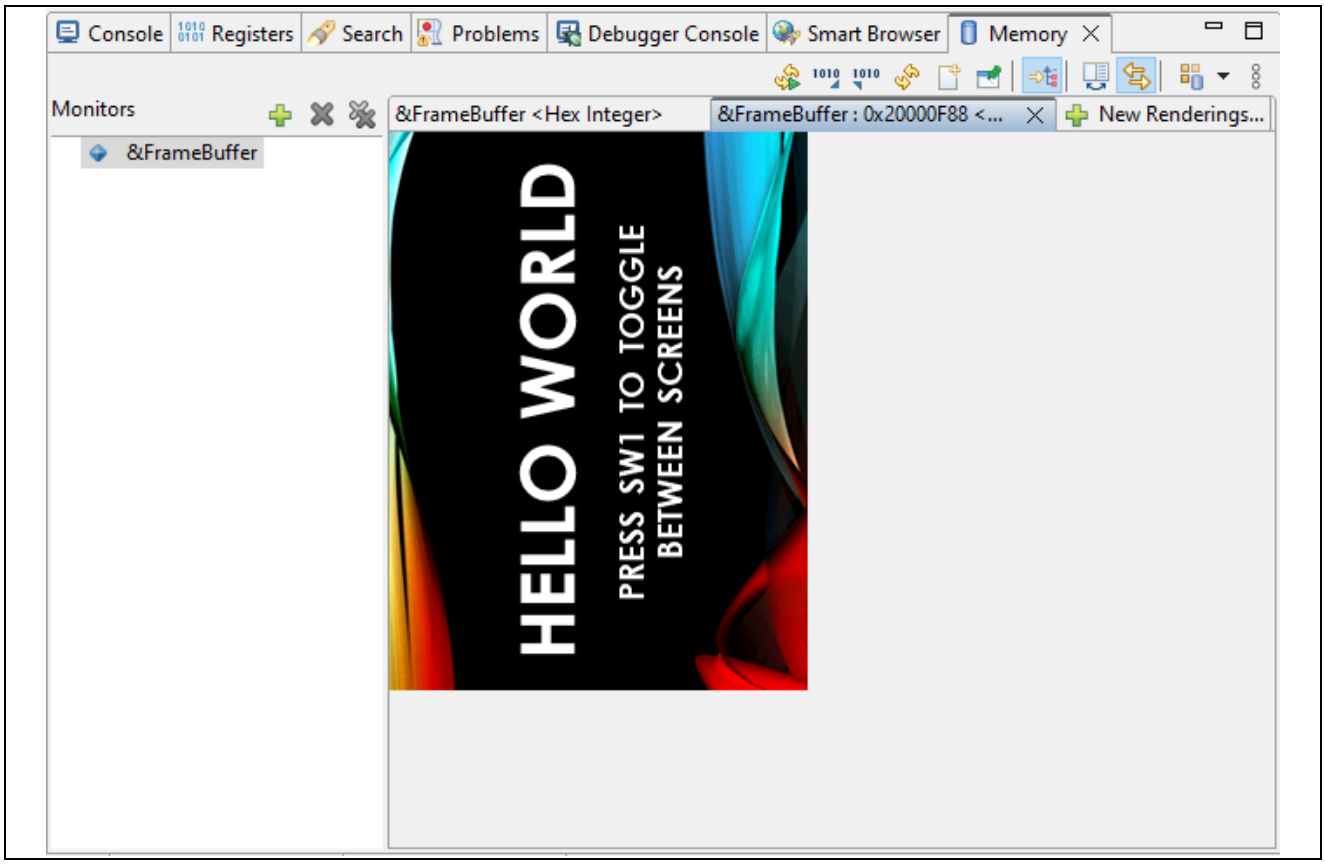


Figure 37. Raw image rendering of the FrameBuffer pixel array in RAM

6. Understanding the Application Project's Key Mechanisms

While some configurations and settings in the Hello World project are specific to the specifications of the NHD LCD controller, the overall design process and graphics framework setup is the same method for any other LCD controller one might wish to drive with an RA MCU.

The parts explaining how to configure the ECBIU module and how to write data over the external memory bus to the LCD can be generalized and used as a guide for designing other applications that drive external LCDs, memory devices, or other peripherals.

This section begins by reviewing the Hello World project's e^2 studio configurations and settings, while analyzing the design choices made for the target GUI application. Then this section explains how the custom code pieces are critical for a smooth application at the system level. Namely it details the operation of the ECBIU drivers supported by the BSP, the LCD drivers, and the middleware layer between the GUIX Studio Library APIs and the LCD drivers.

Note: This section is specific to the software rotation version of the project titled *Hello_World_GUIX_ECBIU_EK_RA6M4*. Please refer to Section 7 for details on how the DMAC rotation version of the project titled *<project>__dmac_rotation* operates and differs from the software rotation version of the project.

6.1 e^2 studio Project Configurations

The configuration.xml file contains the Hello World e^2 studio project's information regarding the configurations of the hardware and software modules, pins, clocks, interrupts, RTOS events, and components on the EK-RA6M4. This file is edited through the FSP Configuration View perspective in e^2 studio. Once it has been configured, clicking "Generate Project Content" will create the FSP and BSP source code content specified.

The important pin, clock and FSP stack settings for the Hello World graphics application are explained below.

6.1.1 Pin Settings

The pin configuration tab is where to assign the roles of the available I/O pins on your project's target RA MCU for communication with the desired system's peripheral. For the Hello World project, the signal

connections to the RA6M4 include the input from the hardware button SW1 and input/output to and from the external memory bus.

The pin P005, corresponding to the hardware button SW1, is configured to IRQ mode and is routed internally to channel 10 on the ICU module.

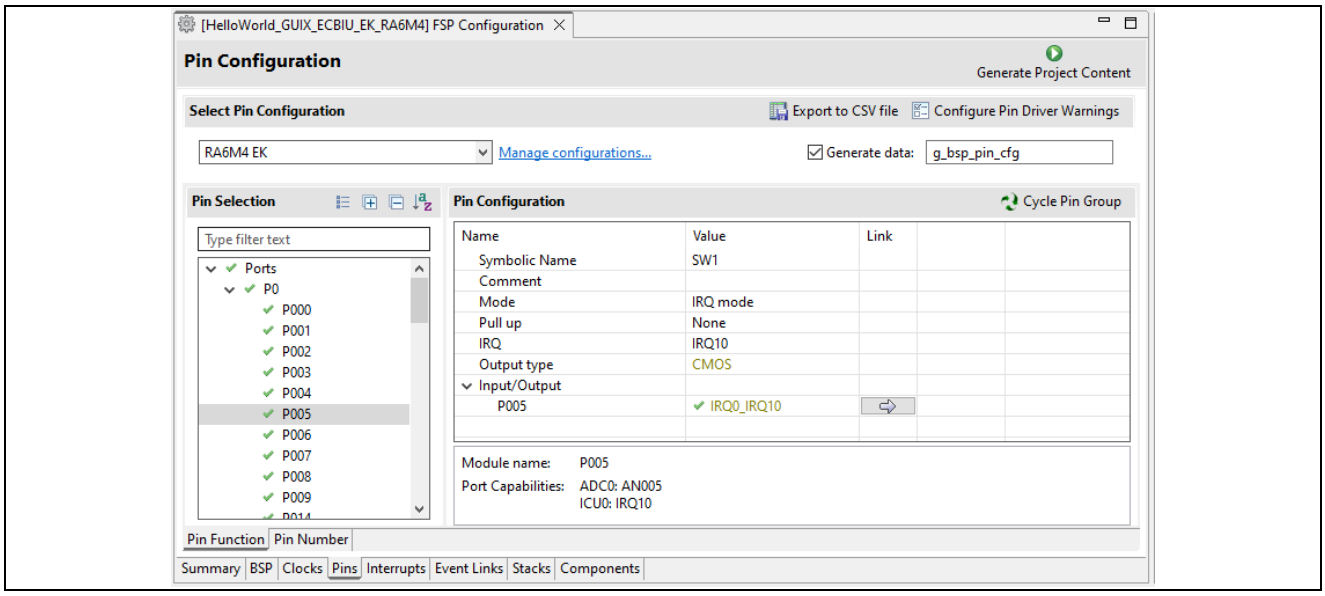


Figure 38. Pin P005 is configured as IRQ input from SW1

The ECBIU pins on the EK-RA6M4 need to be configured to match the pin assignments listed earlier in Table 9 in Section 5.2.1 Connecting the MCU to the LCD.

From the Pins tab, navigate to the **Peripherals > System:BUS > BUS_ASYNCH0** to open up the pin configuration for the external memory bus. Notice the project's Operation Mode is set to "Custom", and the pins activated match the NHD LCD pin assignments table.

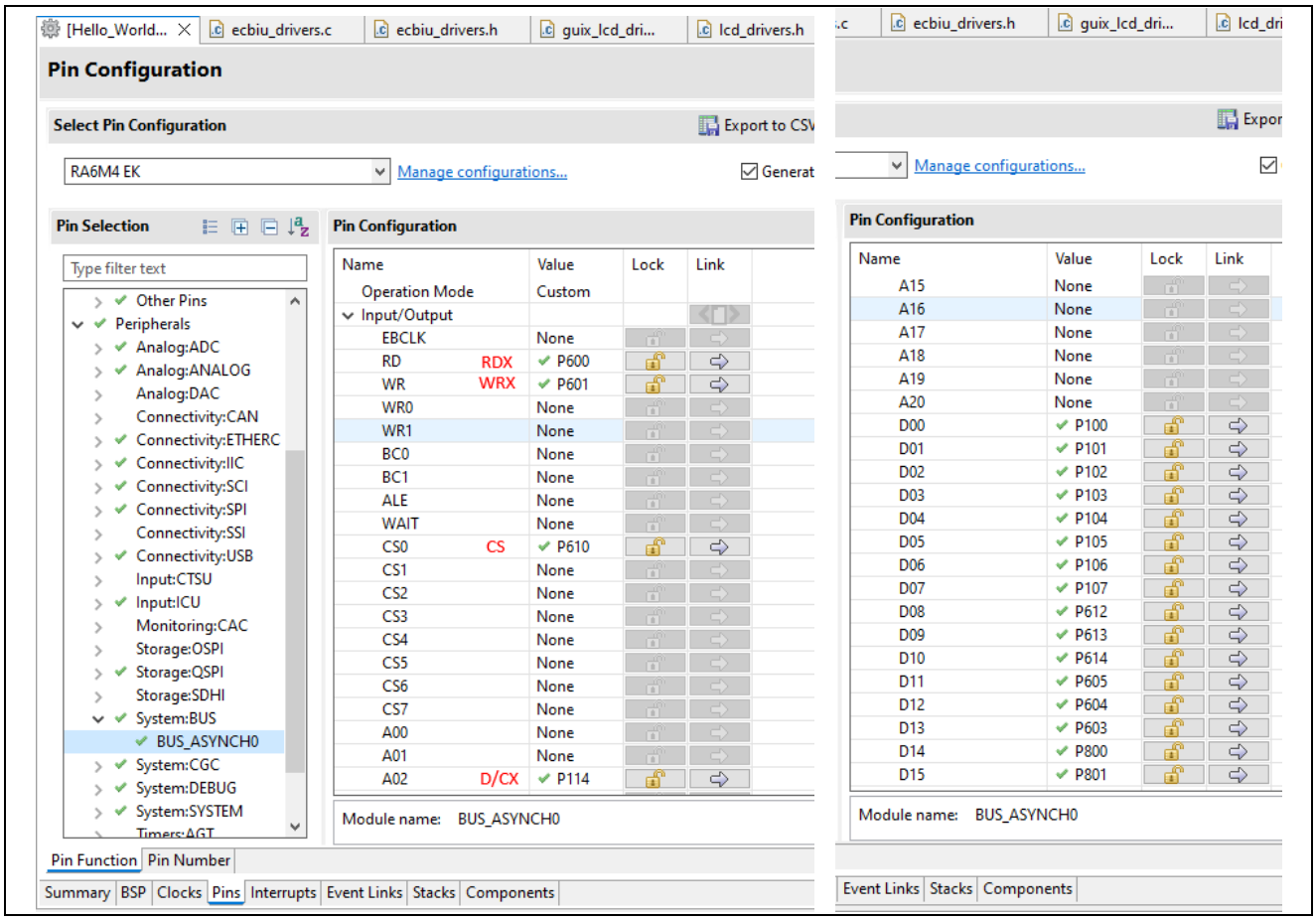


Figure 39. Pin configurations of the ECBIU in the Hello World application

Note: Some MCU pins are assigned multiple functions but each pin can only be configured to one function at a time. If any pins you activate for a particular peripheral are already assigned a function for another peripheral, a red X warning will show next to the pin to indicate the conflict. Be sure to disable the other pin's assigned functionality so that you can set it as desired. When there are no pin assignment conflicts, a green check will show next to the pin.

6.1.2 Clock Settings

The Clocks configuration tab sets up the internal and external clocking on the MCU for the target application. Clock sources and frequency divider settings can be selected for each of the clocks on the MCU. For full details on the Clock Generation Circuit (CGC), please refer to the RA6M4 Group's Hardware Users' Manual.

The CSC (CS area controller) of the external memory bus operates in synchronization with the BCLK. For the EK-RA6M4, the BCLK has a maximum frequency of 100MHz, and its source signal can be divided by 2,4,8,16, and 32 if slower speeds are desired. The PLL clock is selected as the source to the BCLK signal in this application.

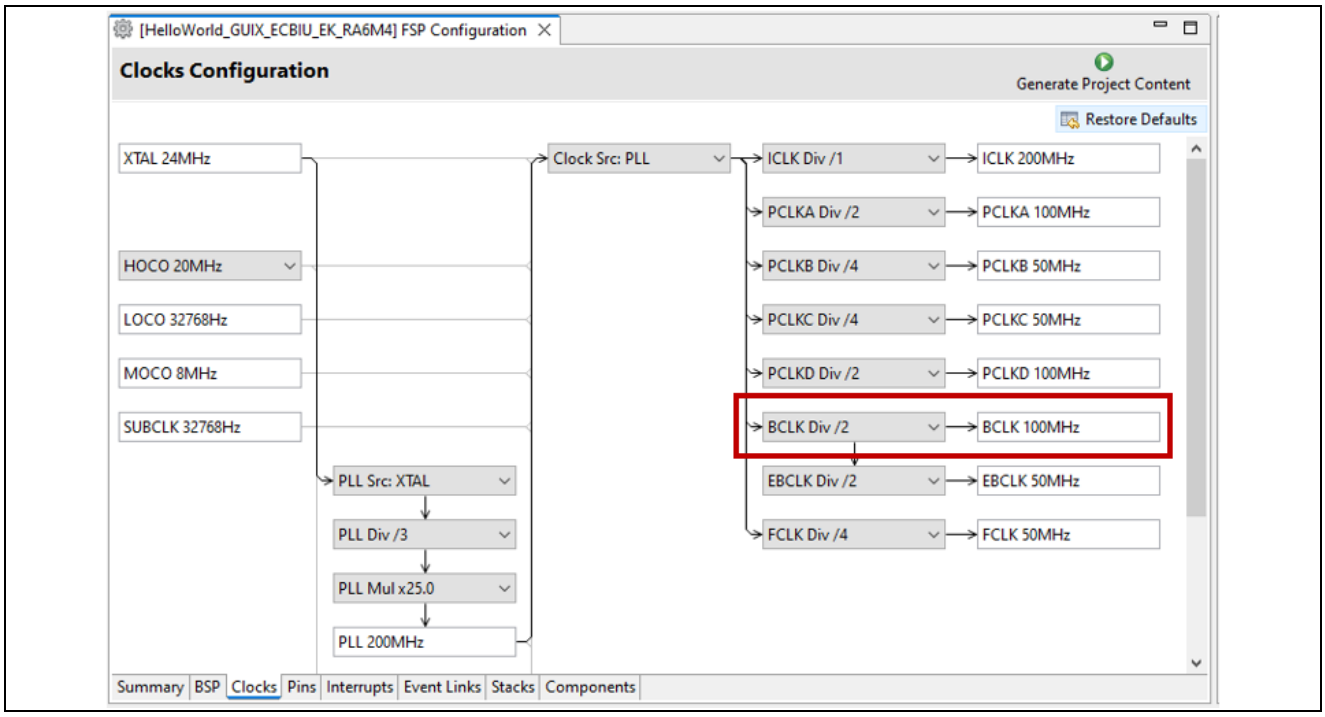


Figure 40. BCLK is configured with a 100MHz frequency

Operation cycles, such as wait cycles specified in the CSC register, are counted on BCLK. With BCLK at 100 MHz, one clock cycle gives a 10 ns period. This is optimal for use with the NHD LCD, whose timing diagrams specify signal wait times between 10 and 160 ns.

6.1.3 Stack Settings

The Stacks configuration tab is where you can add and manage your program's threads, modules, and RTOS object resources. This section explains the necessary resources and the settings required for the graphics application to run smoothly.

(1) r_ioport I/O Port

The I/O Port stack is automatically included in all projects, and it provides access to the drivers for all I/O ports on the MCU. The I/O port pins can operate as general I/O port pins, I/O pins for peripheral modules, interrupt input pins, analog I/O, port group function for the ELC, or bus control pins.

In the Hello World application, the **r_ioport** drivers are used to manually set the level of the LCD Reset signal during LCD startup.

(2) r_icu External IRQ

The External IRQ module is configured to generate an interrupt request when a user presses the button SW1 on the EK-RA6M4. The `button_cb` interrupt callback will be triggered for each SW1 toggle.

The following table explains the project-specific settings for the **r_icu** module which differ from the FSP default:

Table 10. Hello World configurations for the ICU module

Module Property	Default Value	Project Value	Reason
Channel	0	10	Channel 10 corresponds to P005 which is the button SW1.
Callback	NULL	button_cb	Specify name of the callback function.

(3) RTOS Objects

The Objects section of the Stacks tab allows you to configure the RTOS object resources for the project.

The semaphore `g_sempahore_button` is used in the System Thread to synchronize the SW1 button toggling with the software commands for GUIX to transition to the next screen.

6.2 Controlling the NHD LCD

This section begins by providing an in-depth analysis of setting the external bus ECBIU configurations to match the specifications of the NHD LCD controller. This section also explains how to operate the LCD in software, by writing LCD commands and pixel data to the display over the external bus.

The process for determining the correct settings for the ECBIU registers is explained in context of the NHD LCD hardware specifications, but it can be generalized for applications that drive other external peripherals or memory devices. In a similar way, the steps for operating the NHD LCD controller through software commands can be generalized to other LCD controllers.

6.2.1 Custom ECBIU Drivers

A module supported by the FSP has HAL module drivers that provide convenient API functions to access and operate MCU peripherals. Module properties are defined typically with the configuration.xml file, which can be edited using the FSP Configuration view for an e^2 studio project. The configuration GUI eliminates the tedious and error prone process of setting peripheral control registers. When configuration is complete, the generator automatically creates the code needed to implement the associated API functions.

Currently the external memory bus, ECBIU, does not have FSP support in the Stacks tab of the FSP configuration, and it lacks API drivers. There is support for the ECBIU in the form of the peripheral register structures defined in Board Support Package (BSP) files. Developers should create custom drivers for the ECBIU by directly assigning appropriate values to the register pointers. To understand how to operate peripherals for a specific use, developers should consult the MCU's Hardware Users' Manual.

In the RA6M4 Group Hardware Users' Manual, Chapter 14 titled Buses describes the functionality and valid operation conditions of the ECBIU and its registers. After pressing the Generate Project Content button in the FSP Configuration, the associated BSP, FSP, and HAL driver code files are generated based on the specifications in the configuration.xml file. The register structure for the ECBIU peripheral on the EK-RA6M4 is defined in the BSP file `R7FA6M4AF.h` as the `R_BUS_Type`.

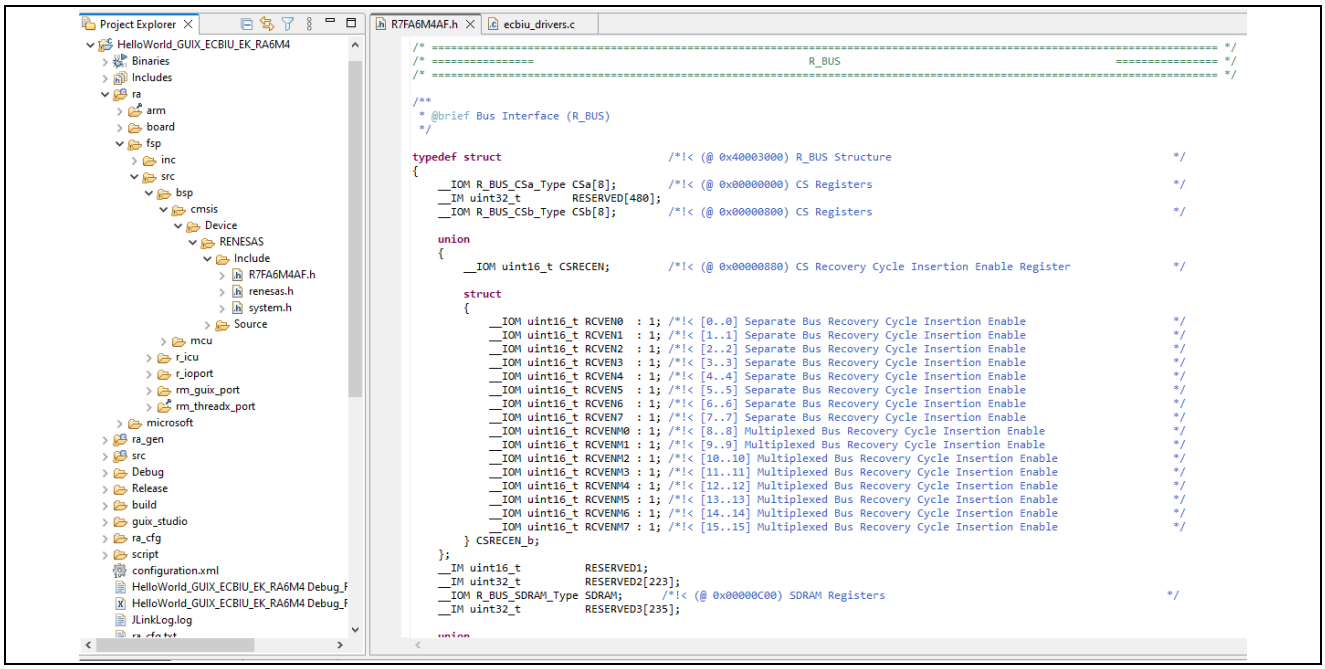


Figure 41. The BSP file R7FA6M4AF.h contains register definitions for all modules on the EK-RA6M4

For the Hello World application, a custom ECBIU configuration driver was created for setting the `R_BUS_Type` register values so the external memory bus operates within the specifications of the NHD LCD controller. The ***ecbiu_lcd_config()*** driver is defined in the `ecbiu_drivers.c` project file.

ecbiu_drivers.c

```
 * ecbiu_drivers.c
#include <ecbiu_drivers.h>

/* Configure the external bus for 16 bit mode. */
void ecbiu_lcd_config(void)
{
    R_BUS->CSb[0].CR_b.BSIZE = 0;       // 16 bit bus
    R_BUS->CSb[0].CR_b.EMODE = 0;       // Little Endian
    R_BUS->CSb[0].CR_b.MPXEN = 0;       // Separate bus interface
    R_BUS->CSa[0].MOD_b.WRMOD = 1;      // Single-write strobe mode

    R_BUS->CSb[0].REC_b.RRCV = 3;       // Read recovery cycles: 3
    R_BUS->CSb[0].REC_b.WRCV = 3;       // Write recovery cycles: 3

    R_BUS->CSRECEN_b.RCVEN0 = 1;        // Recovery cycles specified by RRCV are inserted between cycles: read to read, in same area
    R_BUS->CSRECEN_b.RCVEN2 = 1;        // Recovery cycles specified by RRCV are inserted between cycles: read to write, in same area
    R_BUS->CSRECEN_b.RCVEN4 = 1;        // Recovery cycles specified by WRCV are inserted between cycles: write to read, in same area
    R_BUS->CSRECEN_b.RCVEN6 = 1;        // Recovery cycles specified by WRCV are inserted between cycles: write to write, in same area

    R_BUS->CSa[0].WCR1_b.CSWWAIT = 1;   // See timing diagram for value explanation
    R_BUS->CSa[0].WCR1_b.CSRWAIT = 5;
    R_BUS->CSa[0].WCR2_b.CSROFF = 7;
    R_BUS->CSa[0].WCR2_b.CSWOFF = 2;
    R_BUS->CSa[0].WCR2_b.WDOFF = 1;
}
```

Figure 42. The ecbiu_lcd_config() function sets up the ECBIU for communication with the NHD LCD

The first 4 lines of the custom function configures the external memory bus with 16-bit data width, little-endian alignment, no address/data multiplexing, and single-write strobe mode (a required setting for a 16 bit width bus). The rest of the lines in the ***ecbiu_lcd_config()*** function adjust the timing for the bus signals of the external memory bus so they meet the signal timing requirements specified by NHD LCD, and are further explained in the next section.

6.2.2 Matching ECBIU Timing to LCD Specifications

The timing sequence of the ECBIU signals is highly configurable and signals can be extended or adjusted at the clock cycle level by setting the appropriate R_BUS register. The signal timing flexibility means the ECBIU can be configured to operate many peripherals that have different read and write timing sequences.

The NHD LCD controller interfaces to the MCU with the Intel 8080-II series parallel bus communication protocol. For a refresher of the timing sequence in Section 4.3.1, Figure 43 and Table 11Table 11. Timing specifications for each signal and symbol in diagram again illustrate the timing characteristics of the signals in the 8080-II specification.

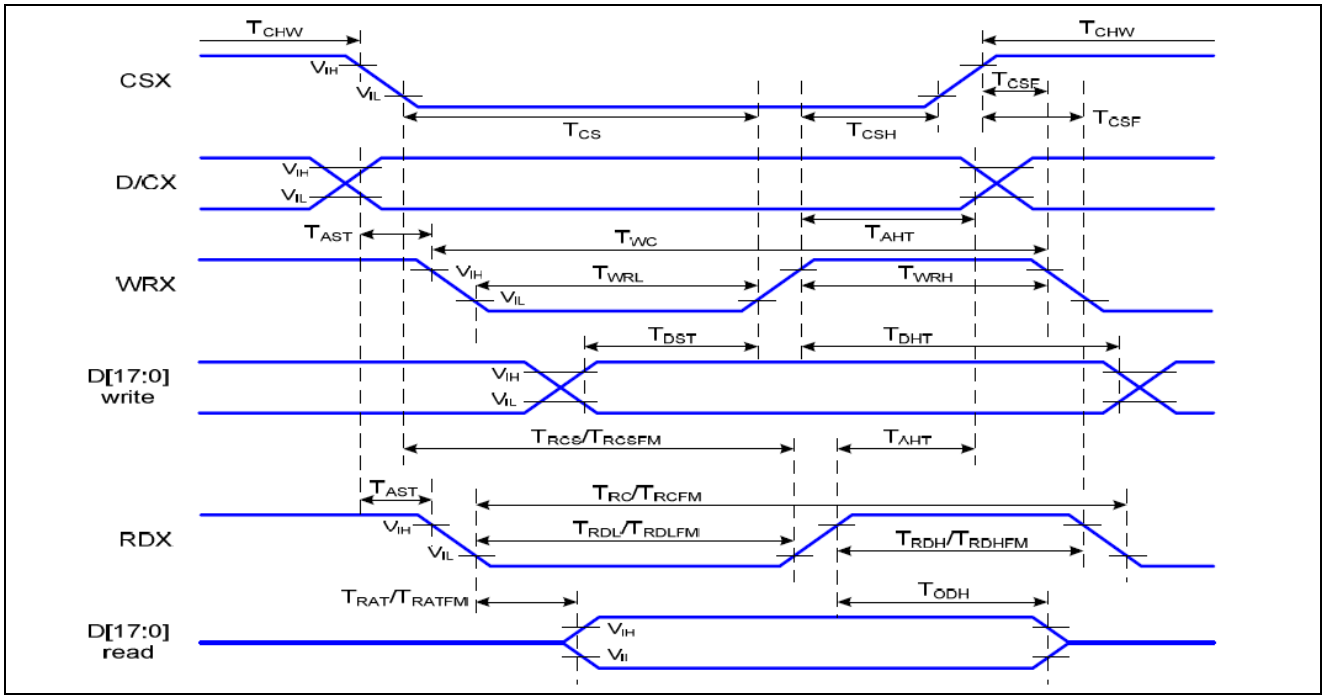


Figure 43. Parallel interface timing characteristics (8080-II Series MCU interface)

Table 11. Timing specifications for each signal and symbol in diagram

Signal	Symbol	Parameter	Min	Max	Unit	Description
D/CX	T_{AST}	Address setup time	0		ns	-
	T_{AHT}	Address hold time (Write/Read)	10		ns	
CSX	T_{CHW}	Chip select “H” pulse width	0		ns	-
	T_{CS}	Chip select setup time (Write)	15		ns	
	T_{RCS}	Chip select setup time (Read ID)	45		ns	
	T_{RCSFM}	Chip select setup time (Read FM)	355		ns	
	T_{CSF}	Chip select wait time (Write/Read)	10		ns	
	T_{CSH}	Chip select hold time	10		ns	
WRX	T_{WC}	Write cycle	66		ns	
	T_{WRH}	Control pulse “H” duration	15		ns	
	T_{WRL}	Control pulse “L” duration	15		ns	
RDX (ID)	T_{RC}	Read cycle (ID)	160		ns	When read ID data
	T_{RDH}	Control pulse “H” duration (ID)	90		ns	
	T_{RDL}	Control pulse “L” duration (ID)	45		ns	
RDX (FM)	T_{RCFM}	Read cycle (FM)	450		ns	When read from frame memory
	T_{RDHFM}	Control pulse “H” duration (FM)	90		ns	
	T_{RDLFM}	Control pulse “L” duration (FM)	355		ns	
D[17:0]	T_{DST}	Data setup time	10		ns	For CL=30pF
	T_{DHT}	Data hold time	10		ns	
	T_{RAT}	Read access time (ID)		40	ns	
	T_{RATFM}	Read access time (FM)		340	ns	
	T_{ODH}	Output disable time	20	80	ns	

The minimum timing requirements can be met by extending the corresponding signal wait registers on the EK-RA6M4. The process for matching the read and write cycles for the EK-RA6M4’s external memory bus to the read and write cycle requirements for the NHD LCD controller are explained further.

For a refresher of Section 2.2 , Figure 44 again illustrates a normal read and write cycle timing diagram of the ECBIU bus signals.

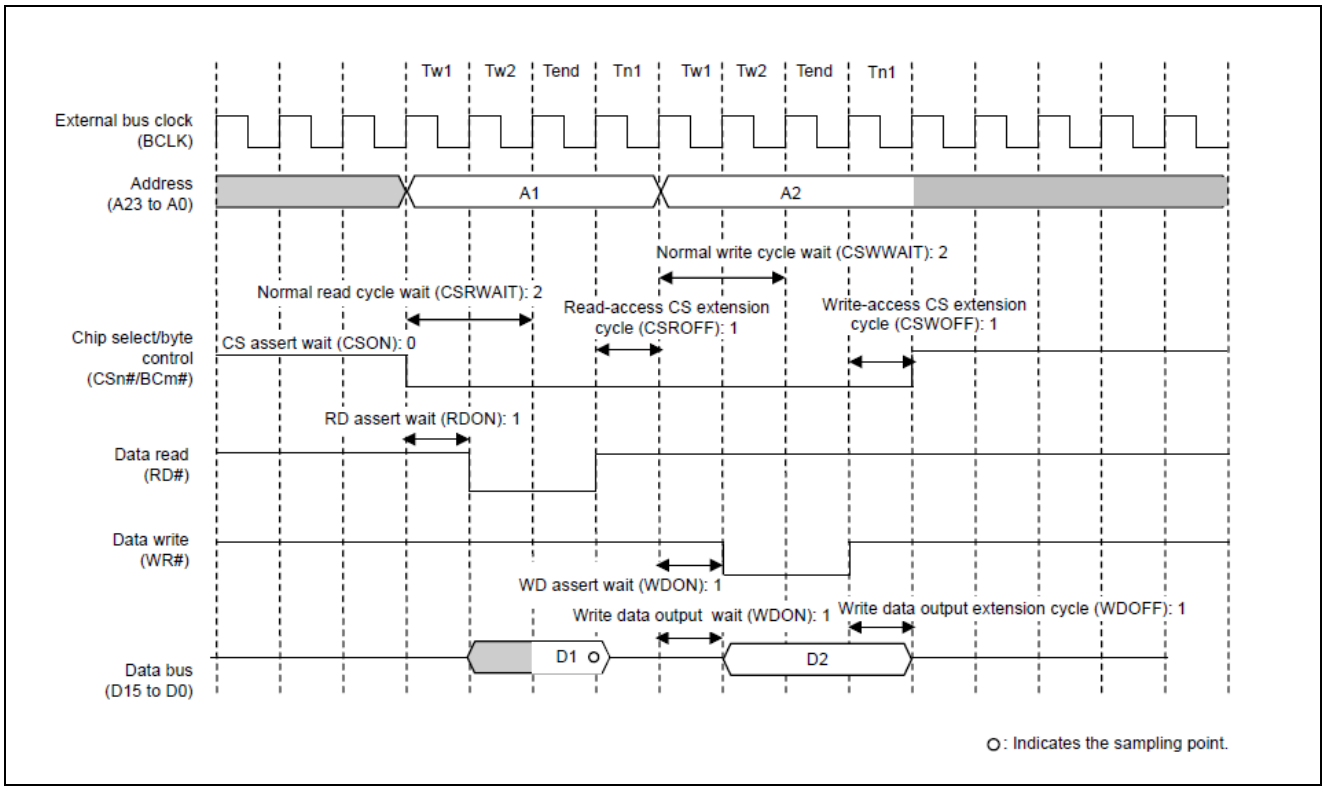


Figure 44. Normal access operation for a read and write sequence

(1) Read Cycle

During a read cycle, the MCU reads information from the LCD controller using the Intel 8080-II series parallel bus communication protocol. The LCD sends valid data bits D[15:0] over the ECBIU when the RDX signal has a falling edge. Then, the MCU reads data bits D[15:0] when the RDX signal has a rising edge.

The listed ECBIU registers specifying the signal timings of the read cycle were set to satisfy the following NHD LCD timing conditions:

- CSRWAIT = 5
  - TRCS (Chip select setup time) >= 45 ns
  - TRDL (RX low pulse duration) >= 45 ns
  - TRAT >= 40 ns
- CSROFF = 7
  - TRDH (RX high duration) >= 90 ns (CSROFF + TREC)
- RDON = 0
  - TRDH (RX high duration) >= 90 ns
  - TCSF (CX high duration until RX signal goes low again on read) >= 10 ns
- CSON = 0
  - TCHW >= 0 (No CSX assert wait needed)
- RRCV, WRCV = 2; RCVEN0, RCVEN4 = 1
  - Write-to-read and read-to-read recovery cycles
- CSRWAIT + CSROFF + 1 + REC >= TRC = 160 ns

Since BCLK = 100MHz, adding 1 clock cycle to any signal will increase the signal's period by 10 ns.

(2) Write Cycle

The normal write cycle for the ECBIU is illustrated in the figure below:

IMG: normal write cycle for the ECBIU

During the write cycle, the MCU sends either data or command information to the LCD controller. The MCU controls data bits D[15:0] on the ECBIU when the WRX signal has a falling edge. Then, the LCD reads data bits D[15:0] when the WRX signal has a rising edge. The D/CX control signal for the LCD controller is connected to address line A02 on the external memory bus. The D/CX signal (connected to A02 on MCU) tells the LCD controller whether the data bits D[15:0] are pixel data or LCD controller commands.

The listed ECBIU registers specifying the signal timings of the write cycle were set to satisfy the following NHD LCD timing conditions:

- CSWWAIT = 1
  - TCS (CSX low until WRX goes high on write) >= 15 ns
  - TWRL (WRX low pulse duration) >= 15 ns
- CSWOFF = 2
  - TAHT (WRX high until DCX is not valid) >= 10 ns
  - TWRH (WRX high pulse duration) >= 15 ns
  - TDST (data valid on bus until WRX goes high) >= 10 ns
- WRON = 0
  - TCSF (CSx high until WRX goes low again on write) >= 10 ns
- CSON = 0
  - TCHW (CSX high pulse width can be 0 ns
- WDOFF = 1
  - TDHT (WRX high until data is no longer valid) >= 10 ns
- CSWWAIT + CSWOFF + 1 >= TWC = 66 ns
- WRCV + Tend + CSWWAIT + RRCV >= TWC = 66 ns

Since BCLK = 100MHz, adding 1 clock cycle to any signal will increase the signal's period by 10 ns.

6.2.3 Sending LCD Data/Commands on the ECBIU

During a write cycle, the RA6M4 sends either pixel data or LCD controller commands over the data bits D[15:0] on the external address space. A high D/CX signal indicates to the LCD that D[15:0] is sending pixel data, and a low D/CX signal indicates to the LCD that the D[7:0] is sending a command.

The LCD's D/CX is tied to the address line 2 (A02) on the ECBIU. The external address space on EK-RA6M4 begins at address 0x8000_0000. The rest of the address lines on the external memory bus are disabled in the Hello World application.

Therefore, writing a 16-bit integer to any external memory space address where the A02 bit is '1' will pull the D/CX line high and indicate a data write to the LCD. Writing an 8-bit integer to any address where the A02 bit is '0' will pull the D/CX line low and indicate a command to the LCD controller. In the file `lcd_drivers.h`, defined constants `LCD_CMD` points to address 0x8000_0000 and `LCD_DATA` points to address 0x8000_0004.

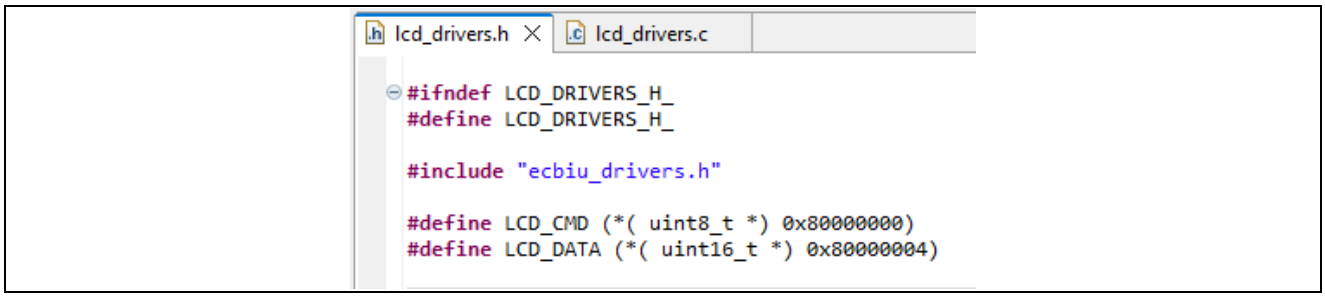


Figure 45. Address definitions for LCD pixel data and commands

6.2.4 LCD Drivers

The file `lcd_drivers.c` contains user-defined function drivers to power on and initialize the LCD controller, fill the LCD controller with a solid color, and turn on the LCD controller's display.

For each driver, the values written to the address pointers `LCD_CMD` and `LCD_DATA` are specific to the NHD LCD controller characteristics and are defined in the ST7789Vi protocol specifications. Other LCD controllers will require their own version of the drivers which can be made by referencing the respective LCD's hardware communication protocol. Typically, LCD drivers are a sequence of the right commands and data values sent out to the display controller to perform a desired function.

The LCD initialization function ***LCD_Init_ST7789Vi()*** performs the following steps to get the LCD turned on and configured correctly:

- Toggle the RESET pin on the LCD
- Tell LCD to exit sleep mode
- Set the memory data access control
- Set the color mode to use 16 bits per pixel, 65k RGB
- Turn on display inversion
- Perform porch setting
- Set gate control
- Set VCOM
- Set LCM control
- VDV and VRH command enable
- VDV and VRH setting
- Set frame rate control to normal mode
- Set positive and negative voltage gamma control
- Set X address from 0-239
- Set Y address from 0-319

The LCD fill function LCD_Fill function performs the following steps to fill the display a solid color:

- Send the LCD command to write memory
- Send a 16-bit 565RGB pixel, for every pixel in the framebuffer (240x320 times)
- Send the LCD command end data transfer

To view other commands and possible settings for the NHD LCD controller, visit the specifications of the ST7789V controller are available for download at: https://support.newhavendisplay.com/hc/en-us/articles/4414907838487-ST7789V.

6.2.5 Backlight Control

The backlight on the LCD consists of 5 white LEDs that require powering by a 3.1 voltage signal with a load of 100mA. Typically, the power source will be a backlight controller, but for prototyping purposes a signal generator with the right load will also work. Adjusting the pulse width modulation on the voltage signal will control the relative brightness of the LCD backlight.

6.3 Middleware Layer Between GUIX and Hardware Drivers

The GUIX Studio Library provides APIs that your application threads can call. The library includes routines that initialize the GUIX system thread, start the GUIX thread's event processing, create and control widgets in the GUI, and send events to the GUIX system thread which in turn will trigger defined actions. Once the GUIX system thread has been started, the thread should run continuously and process any events that are sent to the GUIX queue.

Additionally, some GUIX APIs allow you to provide pointers to any custom display drivers, for setting up the graphics framework hardware on the MCU and for operating the hardware to send pixel data from the MCU

to the LCD. The GUIX system thread will call the custom display drivers when appropriate, using the function pointers provided. If you are writing custom drivers to provide to the GUIX thread you will need to include the gx_display.h header file in your custom driver source.

For the Hello World application, the middleware layer operates between the GUIX system thread and the custom LCD and ECBIU display drivers. The middleware functions are defined in the guix_lcd_drivers.c file and called from the System Thread in the ***guix_init()*** routine. During the routine there are 2 custom display drivers that are passed to the GUIX thread: a graphics hardware initialization function ***ra_gd_setup()*** called during GUIX's startup routine and a toggle framebuffer function ***guix_buffer_toggle_565rgb()*** that is triggered after a GUIX draw routine changes the pixel values stored in the `FrameBuffer` variable. Figure 46 shows a block diagram of the routines comprising the middleware layer between the GUIX system and the hardware display drivers.

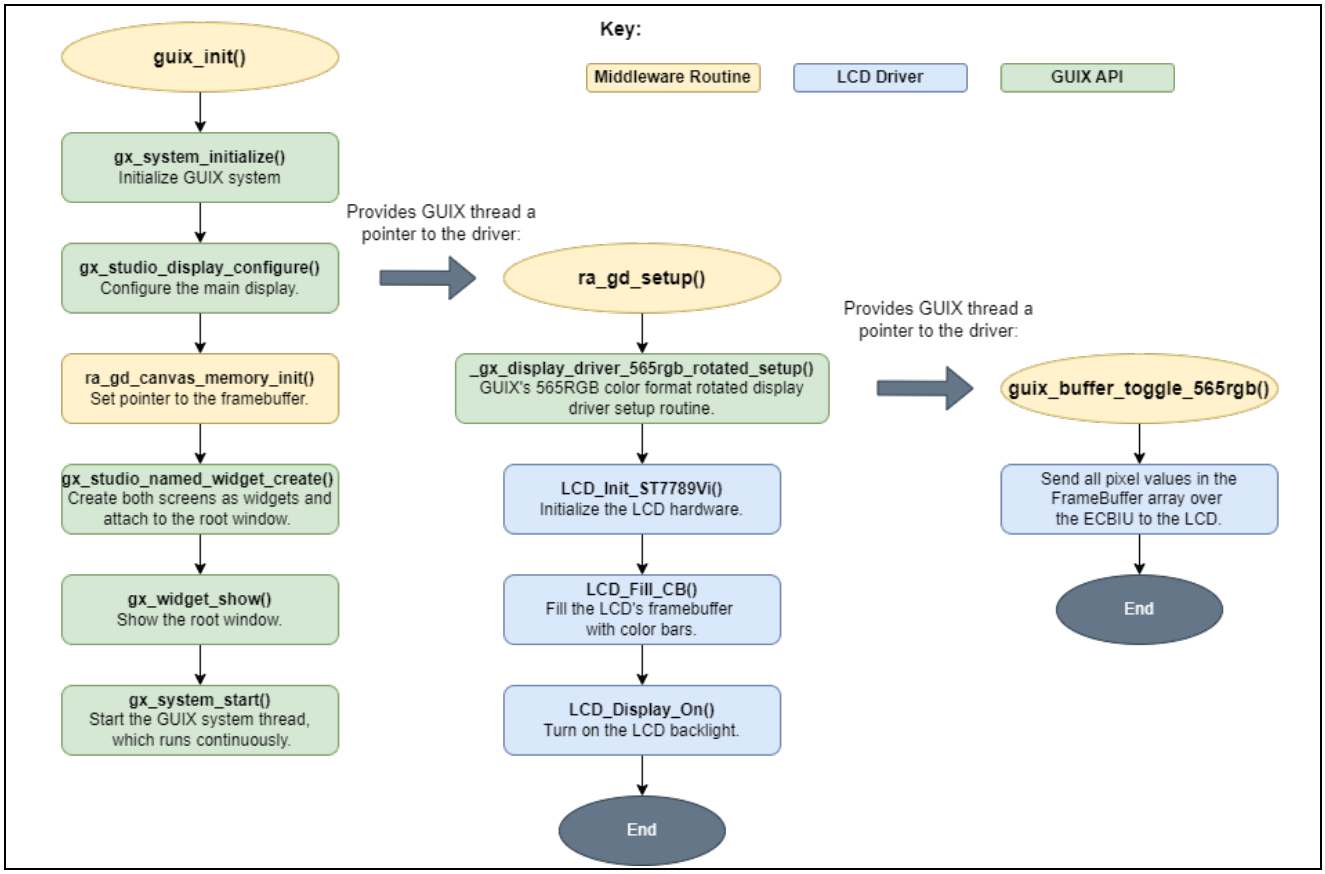


Figure 46. Block diagram of the middleware routines in the Hello World application

Continue reading to understand the operation of the functions that comprise the middleware layer between the GUIX system thread and the hardware display drivers. Refer to the Azure RTOS GUIX User Guide for a complete definition of any GUIX APIs mentioned.

(1) guix_init: Initialize displays and GUIX system

This routine is called from the System Thread and is responsible for initializing the GUIX system, providing the GUIX thread pointers to the custom display drivers, setting up the GUI displays and framebuffer, and starting the GUIX system thread.

The key information for each of the outlined steps of the ***guix_init()*** routine is explained below:

1. ***gx_system_initialize()*** must be called before any other GUIX service is called. This function creates the GUIX event queue, initializes the GUIX timer facility, creates the main GUIX system thread, and initializes various data structures maintained by GUIX during the execution of the application.
2. ***gx_studio_display_configure()*** initializes the main display. The function assigns a pointer to the display diver initialization function ***ra_gd_setup()*** which is stored in the GUIX control block. It also

creates a canvas to fit the display, creates a root window for the canvas, and installs languages and themes.

3. ***ra_gd_canvas_memory_init()*** is a custom function for setting the GUIX canvas memory pointer to the FrameBuffer variable, so that it is accessible to the LCD drivers.
4. ***gx_studio_named_widget_create()*** is used to create top-level screens using the screen name specified within the GUIX Studio application as the widget definition identifier. Used to create a widget for the Hello World Screen and attach it to the root window
5. ***gx_studio_named_widget_create()*** is used again to create a widget for the Renesas Background Screen and attach it to the root window
6. ***gx_widget_show()*** shows the root widget.
7. ***gx_system_start()*** starts GUIX processing. Under normal circumstances this function never returns, but instead continues to process threads GUIX event queue. When the GUIX event queue is empty, this service suspends the calling thread until new events arrive in the GUIX event queue.

(2) ra_gd_setup: Setup graphics hardware

The ***ra_gd_setup()*** function is provided to the GUIX system thread using the API ***gx_studio_display_configure()*** so it can be called by GUIX for display hardware setup. This custom routine sets the GUIX display settings and initializes the LCD hardware.

1. ***_gx_display_driver_565rgb_rotated_setup()*** is the GUIX API driver setup routine that specifies that the framebuffer should be drawn from the pixelmaps using the 565RGB pixel color format and rotated 90^0 clockwise. This function also provides the GUIX thread control block a pointer to the ***guix_buffer_toggle_565rgb()*** low-level LCD framebuffer toggling function.
2. Calls to the LCD driver functions to initialize the LCD, fill the display with color bars, and turn on the display backlights.

(3) guix_buffer_toggle_565rgb: Toggle framebuffer on LCD

The ***guix_buffer_toggle_565rgb()*** is a custom routine that's called by the GUIX thread after a new frame buffer has been drawn by GUIX. The function sends all pixels stored in the `FrameBuffer` variable over the external bus to the LCD.

7. Improving Graphics Performance with DMAC Rotation

This section explains the Direct Memory Access (DMA) rotation version of the Hello World application project, which shares the project title and folder name of *Hello_World_GUIX_ECBIU_EK_RA6M4__dmac_rotation.*

In the software rotation version of the project, the GUIX drivers are responsible for rotating the landscape GUI design 90^0 clockwise so it can be sent to a portrait LCD display. GUIX's rotation is a somewhat intensive software process that occurs during drawing, when the pixelmap data is drawn into the framebuffer. In this project version, GUIX performs the rotation behind the scenes and the ECBIU is sending pixel data to the LCD from a portrait framebuffer array. A visualization is shown in Figure 47 below:

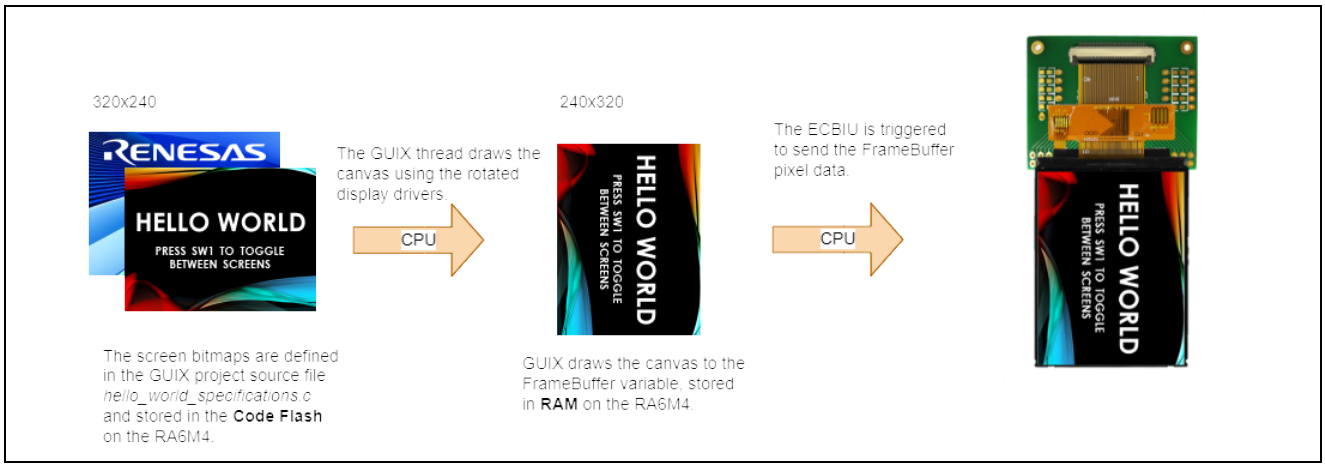


Figure 47. Tracing the display in the software rotation project

In the DMAC rotation version of the project, the DMA Controller is responsible for rotating the landscape framebuffer in RAM to the portrait LCD. The clockwise rotation occurs during the DMAC transfer, as a result of changing the order that the framebuffer's pixel data is sent out of RAM onto the external address space. The DMAC is configured with logic that flips the array's columns and rows, resulting in a rotated design. Using DMAC to transfer a column of the frame buffer data at a time offloads the CPU. This allows other tasks to be completed as the transfer occurs, which reduces overhead. A visualization is shown in Figure 48 below:

Figure 48. Tracing the display in the DMAC rotation project

This section explains how to configure the DMAC module in the e² studio project, set up the GUIX project resolution settings properly, and integrate the DMAC rotation into the LCD low-level and middleware display drivers.

7.1 Configuring the DMAC Module for XY Conversion

The DMA Controller(DMAC) is responsible for transferring the FrameBuffer variable's pixel data from RAM onto the external address space. The DMAC will transfer the framebuffer over the external bus to the LCD, without needing the CPU. The DMAC rotation process is explained further in the RA6M4 Hardware User's Manual in the DMAC chapter under section 16.3.4.2, Example of XY Conversion Using Offset Addition.

To rotate the `FrameBuffer` array 90^0 clockwise, the DMAC module is configured with logic to flip the rows and columns as it transfers the pixel data onto the ECBIU and out the LCD. The DMAC transfers a column of pixel data at a time using offset addition, and so a transfer must be triggered from software for each column. Figure 49 illustrates how flipping columns and rows results in image rotation and Table 12 lists the project-specific settings for the r_dmac module that enables the XY conversion in the Hello World application.

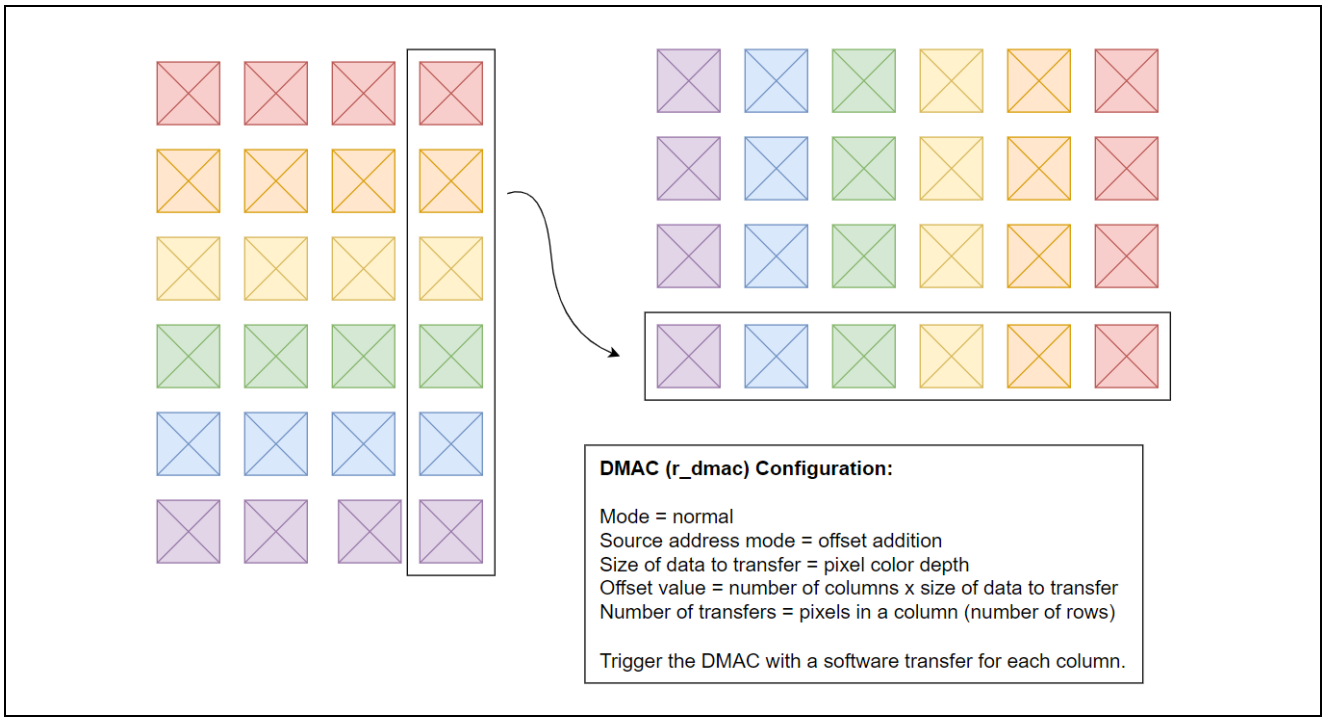


Figure 49. Performing XY conversion using DMAC offset addition

Table 12. Settings of the r_dmac module to enable rotation in the Hello World application

Module Property	Default Value	Project Value	Reason
Mode	Normal	Normal	One data transfer will be initiated for each trigger of the DMAC.
Transfer Size	2 Bytes	2 Bytes	Each pixel data is stored using the 16-bit 565RGB color format.
Destination Address Mode	Fixed	Fixed	All transfers are sent to the defined address pointer LCD_DATA on the external address space.
Source Address Mode	Fixed	Incremented	The DMAC source will increment through the pixel data stored in the FrameBuffer array.
Number of Transfers	1	240	Each DMAC trigger transfers a column. This is the number of pixels in a column.
Callback	NULL	dma_callback	Specify name of the callback function.
Transfer End Interrupt Priority	Disabled	Priority 1	Callback is a high-priority task.

Additionally, in the Stacks tab there is an additional RTOS resource added to the DMAC rotation Hello World application. The following image shows the definition of the semaphore object `g_semaphore_col_done`, which controls the timing of each column transferred by the DMAC.

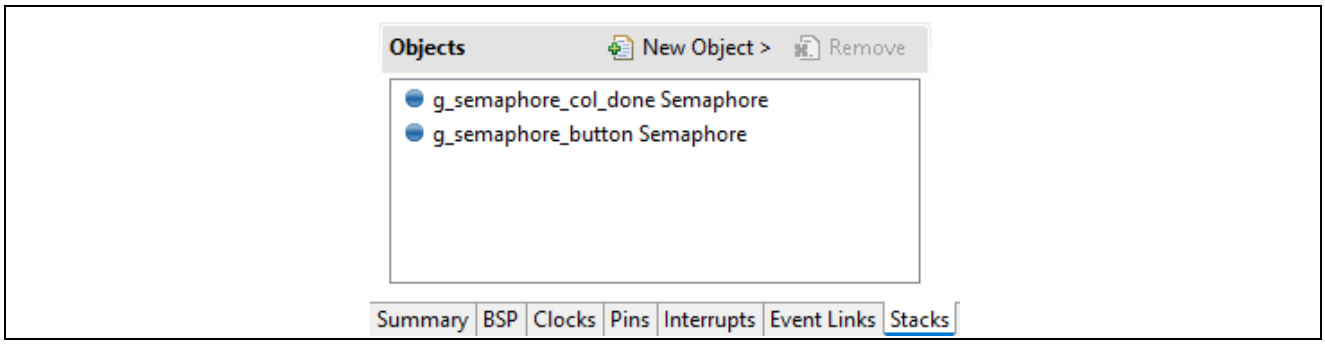


Figure 50. The g_semaphore_col_done is added in the Stacks tab for DMAC transfer synchronization

7.2 Adding DMAC support to the Middleware Layer

The middleware layer drivers that operate between the GUIX system and the hardware level drivers have updated functionality to support DMAC transfer. See Figure 51 for greater details on the implementation of the middleware routines:

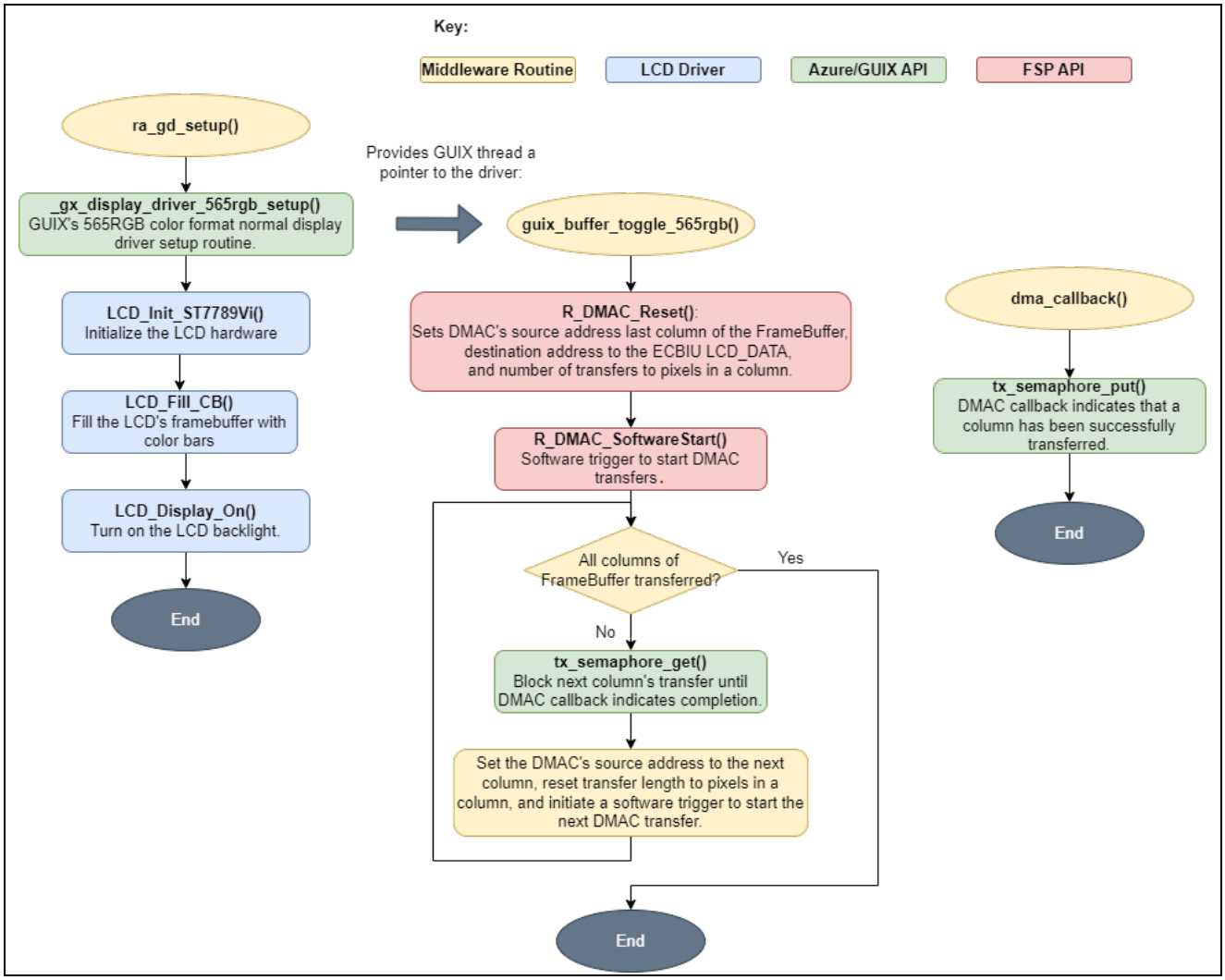


Figure 51. Block diagram of the middleware routines in the DMAC rotation version

The main differences in the middleware layer on the DMAC rotation version of the Hello World application are the following:

- ***ra_gd_setup()*** uses the normal GUIX display driver setup routine ***_gx_display_driver_565rgb_setup()*** instead of the rotated version

- ***guix_buffer_toggle_rgb()*** is implemented to setup the DMAC to transfer each column of the FrameBuffer variable on the ECBIU to the LCD_DATA address, and to work with the ***dma_callback()*** function to initiate the software trigger for each column's transfer by the DMAC.
- ***dma_callback()*** uses the `g_semaphore_col_done` RTOS object to indicate to the toggling function that the next column is ready for transfer.

The approach used in Hello World for the toggle function is to use DMAC to transfer the framebuffer pixels one column at a time, until all columns are written to the external memory space and onto the display. This approach is used because it's not possible to configure the r_dmac module to write the whole framebuffer at once since the total number of transfers required is much larger than the maximum value allowed in the r_dmac configurations.

The `FrameBuffer` variable is an array of 16-bit data with total length of 240x320. The data size corresponds to the NHD display's pixel format, and the array length corresponds to the display's x-resolution by y-resolution.

The r_dmac module is configured to transfer 2 bytes (one 16-bit pixel) 240 times when a software trigger occurs. The source address is configured to increment so that the DMAC traverses down the `FrameBuffer` column during the transfer. After DMAC completes the transfer of a column, the CPU receives an interrupt. After the ***dma_callback()*** returns, the next column is selected as the DMAC source address, and another software trigger restarts the DMAC transfer. The process repeats until all columns have been transferred.

IMG: pic of callback function

Without DMAC, the CPU is typically fully occupied during the entire operation of any I/O read or write cycles, and thus no other instructions or tasks can be executed during this time. With DMA, the CPU initiates the transfer start but can then perform other operations while the transfer is in progress. DMAC also typically achieves faster read/write data transfers than the CPU can achieve.

For the Hello World application's use case using DMAC to rotate the design improved the framerate by 10%. This impact will increase as the resolution of the display increases.

DMAC is especially critical to graphical systems with higher resolutions because they require transferring larger framebuffer sizes, and/or to graphical systems with faster framerate requirements. In both scenarios, framebuffer data must be transferred from the RAM to the LCD at higher rates. DMAC is useful any time that the CPU cannot keep up with the rate of data transfer, or when the CPU needs to perform work while waiting for a relatively large (and thus slower) I/O data transfer. Overall, systems with DMAC can transfer data to and from devices with much less CPU overhead than a system without.

8. References

List of the GUIX Reference documents and links.

- GUIX Studio User's Guide: https://rtos.com/solutions/guix-studio/embedded-ui-design-tool/

List of the SSP and Application Reference documents and links.

- GUIX Hello World Application Project on PE-HMI1: https://www.renesas.com/en-us/software/D6001630.html

Website and Support

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

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

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Oct.01.24	—	First release document

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.