Edge Vision SoC User Guide

UG-EVSOC-v1.1
December 2020
www.efinixinc.com

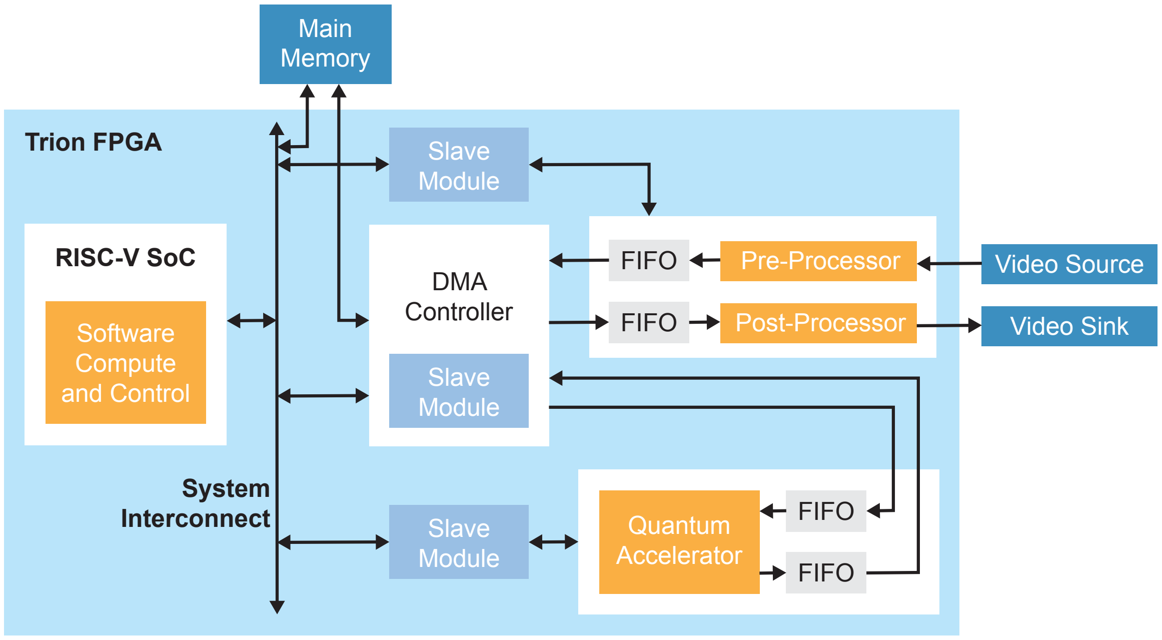
Contents

Introduction

Artificial intelligence (AI) and vision at the edge is an emerging domain that spans a wide range of applications such as automotive, industrial vision, retail, and robotics. The Efinix® Edge Vision SoC framework, which uses a soft RISC-V SoC, facilitates faster time-to-market for edge vision applications. The Edge Vision SoC framework supports crucial domain-specific embedded software functions, hardware accelerator modules, memory and I/O related interfaces, peripherals, and controllers.

The Edge Vision SoC framework uses Quantum accelerators to facilitate hardware/software partitioning and achieve the desired performance. Within this framework, Efinix provides example designs for specific functions, such as video processing, AI object detection, machine learning, multi-camera fusion, etc.

Figure 1: Edge Vision SoC Framework



This user guide describes how to use the Image Signal Processing example design, which is the first design available in the Edge Vision SoC framework.

The Image Signal Processing example design illustrates a use case for the Edge Vision SoC framework, specifically, hardware/software co-design for video processing. Additionally, the design shows how you can control the FPGA hardware using software, that is, you can enable different hardware acceleration functions by changing firmware in the RISC-V processor.

This example presents these concepts in the context of video filtering functions; however, you can use the same design with your own hardware accelerator block instead of the provided filtering functions. The design helps you explore accelerating computationally intensive functions in hardware and using software to control that acceleration.

Chapter 1

Installation

Contents:

Install the Efinity® Software

If you have not already done so, download the Efinity® software from the Support Center and install it. For installation instructions, refer to the Efinity Software Installation User Guide.

Warning: Do not use spaces or non-English characters in the Efinity path.

Install the Edge Vision SoC

To install the edge vision SoC files:

1. Create a directory called **riscv** at the root level of your file system.
2. Unzip the files into the **riscv** directory.

The files are organized in this directory structure:

- **evsoc**
  - **soc_hw**—Hardware files.
    - **sim**—Testbench files.
    - **source**—RTL source code.
    - **efinity_project**—Example Efinity® projects illustrating different use cases.
  - **soc_sw**—Software files.
    - **bsp**—Board specific package.
    - **software**—Software examples.
    - **config**—Has the Eclipse project settings file and OpenOCD debug configuration settings files for Windows.
    - **config_linux**—Has the Eclipse project settings file and OpenOCD debug configuration settings files for Linux.
    - **cpu0.yaml**—CPU file for debugging with OpenOCD.

Install the RISC-V SDK

To install the SDK:

1. Download the file **riscv_sdk_windows-v<version>.zip** or **riscv_sdk_ubuntu-v<version>.zip** from the Support Center.
2. Create a directory for the SDK, such as **c:\riscv-sdk** (Windows) or **home/my_name/riscv-sdk** (Linux).
3. Unzip the file into the directory you created. The complete SDK is distributed as compressed files. You do not need to run an installer.

Windows directory structure:

- **SDK_Windows**
  - **eclipse**—Eclipse application.
  - **GNU MCU Eclipse**—Windows build tools.
  - **openocd**—OpenOCD debugger.
  - **riscv-xpack-toolchain_8.3.0-1.1_windows**—GCC compiler.
  - **run_eclipse.bat**—Batch file that sets variables and launches Eclipse.
  - **setup.bat**—Batch file to set variables for running OpenOCD on the command line to flash the binary.

Ubuntu directory structure:

- **SDK_Ubuntu<version>**
  - **eclipse**—Eclipse application.
  - **openocd**—OpenOCD debugger.
  - **riscv-xpack-toolchain_8.3.0-1.1_linux**—GCC compiler.
  - **run_eclipse.sh**—Shell file that sets variables and launches Eclipse.
  - **setup.sh**—Shell file to set variables for running OpenOCD on the command line to flash the binary.

Install the Java JRE

To install the JRE:

1. Download the 64-bit version of the JRE for your operating system from www.java.com/en/download/manual.jsp.
2. Follow the installation instructions on the Java web site to install the JRE.

Note: You need a 64-bit version of the Java JRE. If you use a 32-bit version, when you try to launch Eclipse you will get an error that Java quit with exit code 13.

Chapter 2

Set Up the Hardware

Contents:

Required Hardware

The example design uses the following hardware from the Trion® T120 BGA324 Development Kit:

- Trion® T120 BGA324 Development Board
- Raspberry Pi Camera Connector Daughter Card
- MIPI and LVDS Expansion Daughter Card
- Raspberry Pi v2 camera module
- 15-pin flat cable
- Micro-USB cable
- 12 V power adapter

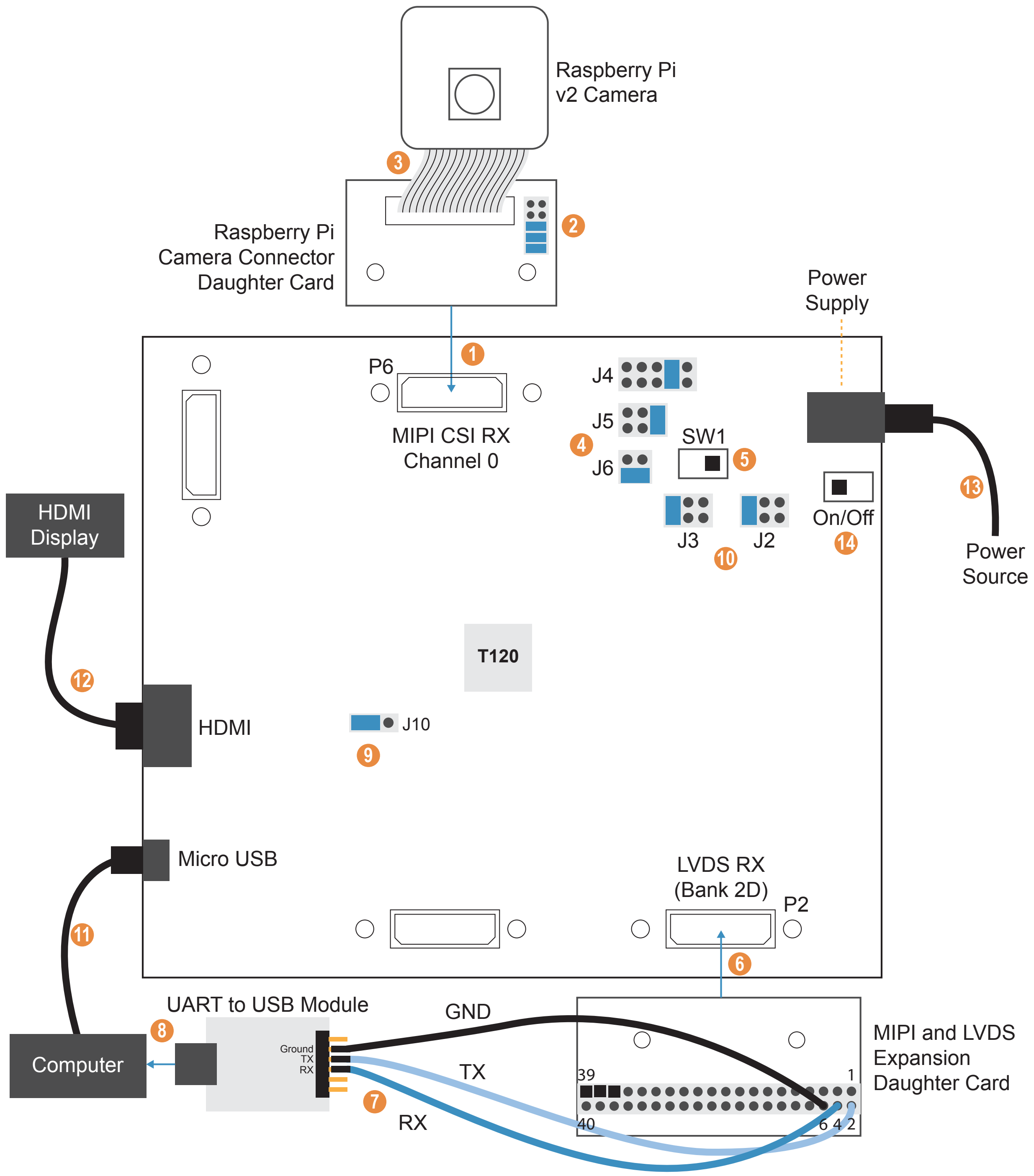
You also need the following hardware, which you provide yourself:

- USB-to-UART module
- 3 jumper wires
- 1080p monitor with HDMI connector
- HDMI cable
- Computer with Efinity® software installed

Set Up the Development Board

The following figure shows the hardware setup steps. If you have not already done so, attach standoffs to the board.

Figure 2: Hardware Setup



Important: Make sure that the Trion® T120 BGA324 Development Board is turned off before connecting any cards or cables.

Set Up the Camera

1. Connect the Raspberry Pi Camera Connector Daughter Card to the board at P6.
2. On the daughter card, connect the following pins with jumpers: 1 - 2, 3 - 4, and 5 - 6.
3. Connect the Raspberry Pi v2 camera module to the daughter card using the 15-pin flat cable.
4. Connect jumpers to set the power sequence:
   - *VSUP1 (J4)*—Connect pins 3 - 4
   - *VSUP2 (J5)*—Connect pins 1 - 2
   - *VSUP3 (J6)*—Connect pins 2 - 4
5. Slide SW1 to position 3, which enables the power up sequence circuit for the MIPI CSI-2 cameras.

Set Up the UART Module

6. Connect the MIPI and LVDS Expansion Daughter Card to the board at P2.
7. Connect jumper wires to the daughter card and UART module:
   - RX to pin 4
   - TX to pin 2
   - GND to pin 6
8. Connect the UART module to your computer.

Connect Cables and Jumpers

9. On header J10, connect pins 2 - 3 with a jumper (default) to use the on-board oscillator.
10. On J2, connect pins 5 - 6.
11. On J3, connect pins 5 - 6.
12. Connect a USB cable to the board and to your computer.
13. Connect an HDMI cable to the board and to your HDMI display.
14. Connect the 12 V power supply to the board connector and to a power source.
15. Turn on the board using the power switch.

Installing USB Drivers

Efinix development boards have FTDI (FT232H, FT2232H, or FT4232H) chips to communicate with the USB port. These chips have separate channels that the boards use for SPI, JTAG, and UART interfaces. If you install the driver for each interface, each interface appears as a unique FTDI device. If you install a composite driver, all of the separate interfaces appear as a composite device.

If you have not already done so, install a composite driver for the Trion® T120 BGA324 Development Board. When working with OpenOCD, you need to install the **libusbK** driver as described in the following section for Windows.

Note: If you already installed the **libusb-win32** driver and want to use OpenOCD, uninstall **libusb-win32** and install **libusbK** instead.

Installing Drivers on Windows

1. Download the Zadig software from zadig.akeo.ie. (You do not need to install it; simply run the downloaded executable.)
2. Run the Zadig software.

   Note: To ensure that the USB driver is persistent across user sessions, run the Zadig software as administrator.

3. Choose **Options > List All Devices**.
4. Turn off **Options > Ignore Hubs or Composite Parents**.
5. Select the Trion® T120 BGA324 Development Board.
6. Select **libusbK** (*version*) in the **Driver** drop-down list.
7. Click **Replace Driver**.
8. Close the Zadig software.

When you open the Device Manager in the Windows Control Panel, it displays the new USB device driver.

Installing Drivers on Linux

The following instructions explain how to install a USB driver for Linux operating systems.

1. Disconnect your board from your computer.
2. In a terminal, use these commands:

```
> sudo <installation directory>/bin/install_usb_driver.sh
> sudo udevadm control --reload-rules
```

Note: If your board was connected to your computer before you executed these commands, you need to disconnect and re-connect it.

Program the Trion® T120 BGA324 Development Board

The Trion® T120 BGA324 Development Board ships pre-loaded with an example design that sends color bars to an HDMI monitor. To use the Image Signal Processing design, you must program the design into the board.

1. Open the project (**edge_vision_soc.xml**) in the Efinity® software and review it.
2. Use the Efinity® Programmer and SPI active mode to download the bitstream file to your board. The example includes a bitstream file, **edge_vision_soc.hex**. You use SPI active mode because you need to reset the FPGA.
3. Press SW2 (CRESET) on the Trion® T120 BGA324 Development Board to reset the FPGA. This reset ensures that the DDR memory initialization happens before the user application runs.

Learn more: Instructions on how to use the Efinity® software is available in the Support Center.

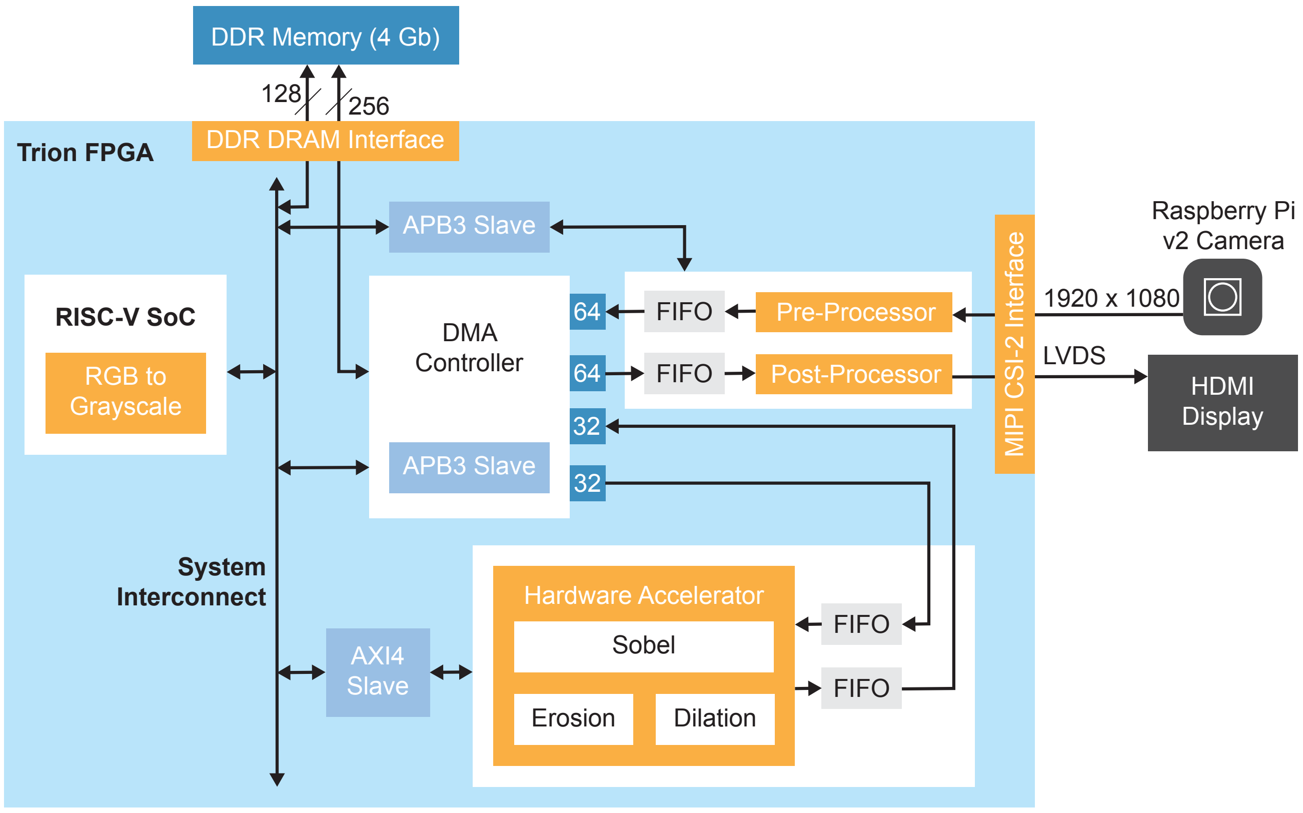
Chapter 3

About the RTL Design

Contents:

So you can get started easily with the Image Signal Processing example, Efinix provides an RTL design targeting the Trion® T120 BGA324 Development Board. In this design, the Hardware Accelerator demonstrates how you can use software (RISC-V firmware) to control the FPGA hardware (acceleration functions).

Figure 3: Image Signal Processing Example Block Diagram



Ruby Vision RISC-V SoC

The Ruby Vision RISC-V SoC performs overall system control and executes algorithms that are inherently sequential or require flexibility. The Ruby Vision SoC supports a highly flexible hardware/software co-design methodology, so you can choose whether to perform compute in the RISC-V processor or in hardware.

The Ruby Vision RISC-V SoC included in the design is a variation of the standard one provided in the Efinix Support Center. Refer to the Ruby Vision Vision SoC Data Sheet for the specifications.

DMA Controller

The DMA controller facilitates communication between the DDR memory and other building blocks in the design. It stores frames of data into the DDR DRAM memory, sends and receives data to/from the hardware acceration block, and sends data to the post-processing engine. The DMA controller has dedicated, fixed-direction channels for each port.

DMA controller has modes: direct mode and scatter and gather (SG) mode. In both modes, to determine whether the triggered DMA transfer is completed, the RISC-V processor uses polling (default) or interrupt.

- *Polling*—The RISC-V processor checks the busy signal until it is de-asserted.
- *Interrupt*—The RISC-V can perform other operations while the DMA transfer is in progress; it receives an interrupt when the transfer is completed.

In the RTL design, the DMA controller connects to an external 256-bit DDR DRAM port. A wrapper (**dma2ddr_wrapper.v**) is created on top of the dma controller (**dma_socRuby.v**). The DMA files are located in the **soc_hw/source/dma** directory.

The **dma2ddr_wrapper.v** file has the following top-level ports (**soc_hw/ T120F324_devkit/edge_vision_soc.v**):

Table 1: DDR Controller Wrapper Ports

Port	Description
dma_clk	DMA controller clock.
dma_reset	DMA controller reset.
ctrl_*<name>*	APB3 ports. These ports connect to the Ruby Vision SoC's APB3 ports, and facilitate communication between the RISC-V processor and DMA controller.
io_ddr_*<name>*	DDR ports. These ports connect to the external DDR module. You set up these ports in the Interface Designer using the DDR block.
dat64_i_ch0_*<name>*	64-bit DMA channel 0 to main memory.
dat64_o_ch1_*<name>*	64-bit DMA channel 1 from main memory.
dat32_i_ch2_*<name>*	32-bit DMA channel 2 to main memory.
dat32_o_ch3_*<name>*	32-bit DMA channel 3 from main memory.

You use the channel number in the firmware to set up and trigger operations on a particular DMA channel.

Each DMA channel has a clock and reset, AXI streaming signals. The `descriptorUpdated` signal is reserved for DMA operation in SG mode (beyond the scope of this user guide).

Note: By default the DMA Controller operates in direct mode. You can modify the firmware to run it in interrupt mode by following the instructions in **Using the DMA Controller in Interrupt Mode**.

Video Capture and Pre-Processing

The Image Signal Processing design uses a Raspberry Pi v2 camera module as the input sensor to the system. The video comes from the camera to the MIPI CSI-2 RX interface at 1920 x 1080 resolution, RAW10 format (10 bits per raw pixel), 4 pixels per clock, with a `mipi_pclk` of 75 MHz. The frame rate varies depending on the camera driver setting.

The Pre-Preprocessor receives the video from the camera and performs these functions:

- Adjusts the RGB gain according to the settings you choose in the RISC-V firmware.
- Crops the video for the HDMI display.
- Changes the number of pixels per clock from 4 to 2.
- Converts RAW to RGB.
- Optionally converts the RGB pixel data to grayscale pixel data.

When it has finished these functions, the Pre-Preprocessor stores the data into the DDR DRAM memory via the DMA Controller.

Firmware running on the RISC-V processor specifies whether the RGB or grayscale data is stored in the DDR memory. So you can choose to implement the conversion to grayscale in hardware, or to perform that function in the RISC-V processor instead.

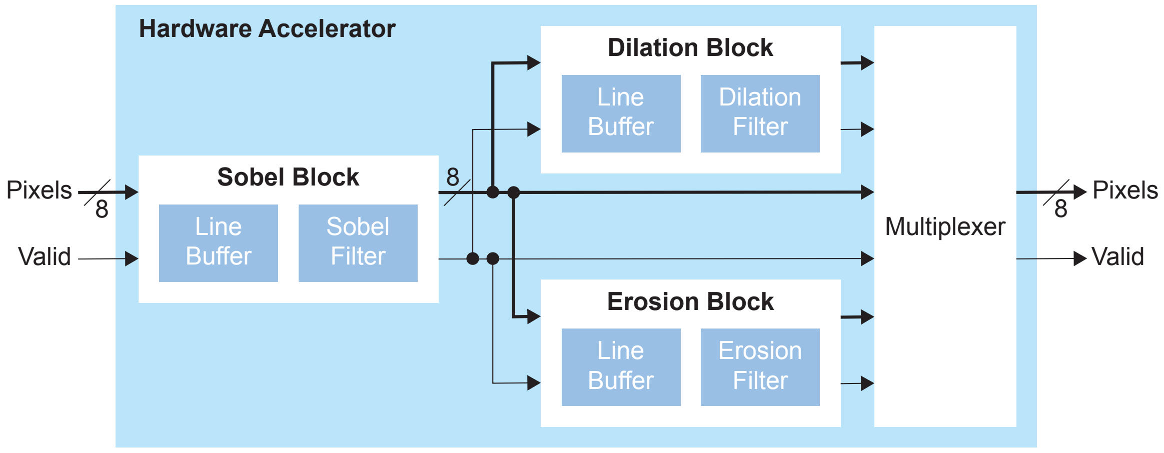
Hardware Accelerator

The design provides several standard functions in the Hardware Accelerator. (These functions are also available as software code.) By default, the design illustrates hardware/software co-design: the RISC-V processor executes the RGB to grayscale conversion as embedded software while the Hardware Accelerator performs the Sobel edge detection, binary erosion, and binary dilation.

- *Sobel filter*—Performs edge detection.
- *Binary erosion*—Removes line detail by ANDing all windowed pixels.
- *Binary dilation*—Strengthens line detail by ORing all windowed pixels.

The Hardware Accelerator modules are implemented in a pipelined, streaming architecture, and operates in three modes: Sobel, Sobel with dilation, or Sobel with erosion. You specify the mode in the firmware running on the RISC-V processor.

Figure 4: Hardware Acceration Block Diagram



In the RTL design, a standard wrapper encompasses the acceleration functions, input and output FIFOs, as well as debug and control registers that interface with an AXI4 slave module that connects to the RISC-V processor. The FIFOs and control logic fetch input data from the DDR DRAM and store output data to DDR DRAM through the DMA Controller. This wrapper is designed to provide a standard interface so you can drop in your own acceleration functions easily.

Post-Processing and Display

The Post-Processor generates video signals (HSYNC, VSYNC, and DE), and pixel data for the LVDS interface. It creates a TX video signal that conforms to the VESA specifications (operating frequency, blanking, etc.,) for a 640 x 480 or 1280 x 720, 60 Hz HDMI flat panel display. It operates using a `tx_slowclk` of 12.37 MHz for 640 x 480 resolution. 37.12 MHz for 1280 x 720 resolution.. The Post-Processor fetches the frame data for the HDMI display from the DDR memory.

Working with the Example

Working with the Image Signal Processing example design involves these steps:

1. Set up the hardware and download a bitstream file to the Trion® T120 BGA324 Development Board.
2. Set up an Eclipse project and configure the OpenOCD debugger.
3. Download the firmware binary to the RISC-V processor running in the Trion T120 FPGA.

Out of the box, the design captures video from the Raspberry Pi camera and sends it to the HDMI display. However, you can modify the software to enable the filtering functions as described in Customizing the Firmware on page 29.

Simulate

The Image Signal Processing example includes simulation files for use with ModelSim simulators. The testbench includes system-level SoC simulation but does not cover the MIPI RX input or LVDS output interfaces. The input to the simulation testbench is based on off-line RGB frame data. The **file image_64_48.vh** is 64 x 48 resolution, and `image_640_480.vh` is 640 x 480 resolution. The smaller resolution file is useful for speeding up the overall simulation time. The pixel data format is 32 bits per pixel {8'd0, 8-bit BLUE, 8-bit GREEN, 8-bit RED}. You can select the image data header file in `tb_soc.v`. You can also update the corresponding `FRAME_WIDTH` and `FRAME_HEIGHT` parameters.

The simulation is based on a simple DDR DRAM model; the 256- and 128-bit ports connect to two independent simple DDR models. Therefore, data written to one port is not synchronized with the other. Hardware/software codesign simulation is only feasible when the DMA Controller shares the same 128-bit DDR port as the RISC-V processor (connected to the DDR master port of the RISC-V system interconnect). In the firmware created for simulation purposes (**evsoc_ispExample_sim**), you can observe that the RGB to grayscale conversion is performed in the camera Pre-Processor block; RISC-V procecssor does not perform and the algorithmic computations.

Output checking is done on incoming data fetched by the DMA Controller into the Post-Processor; simulation does not check the LVDS format. The intermediate grayscale and display output frame data is dumped to text files for verification.

To simulate:

1. Open a Command Prompt (Windows) or terminal (Linux).
2. Change to the **soc_hw/sim** directory.
3. Execute the command `./run.bat` (Windows) or `./run.sh` (Linux).

By default, the **run.bat** and **run.sh** scripts simulate the **evsoc_ispExample_sim** application. To simulate a different application, change the value of the `MyApp` variable and copy the application's binary file into the **sim** directory.

For vision-based systems, you can examine the output of individual processing blocks through visualization. With the provided simulation testbench, intermediate pixel data is dumped to text files, and you can convert the pixel data into a standard image format to verify the result using an image viewer. Alternatively, you can perform a pixel-by-pixel comparison if you have an equivalent software implementation of all of the algorithmic blocks. You can do the comparison with your preferred method, such as MATLAB, Python, C or C++, which serves as the golden reference model.

Chapter 4

About the Software

Contents:

The Image Signal Processing design has software code in the **soc_sw** > **software** directory.

Table 2: Software Directory Structure

Directory	Description
common	Provides linking for the makefiles.
driver	This directory contains the system drivers for the RISC-V peripherals (I^2C, UART, SPI, etc.). Refer to API Reference on page 42 for details.
evsoc_ispExample	Contains the software code for normal operation.
evsoc_ispExample_demo	This design is similar to evsoc_ispExample except you can control the hardware acceleration function using UART commands in a terminal. With this method, you can view the different acceleration options without recompiling the software code.
evsoc_ispExample_demo2	This design is similar to evsoc_ispExample except you can control the hardware acceleration function using switches on the development board.
evsoc_ispExample_sim	Contains the software code for simulation.
evsoc_ispExample_timestamp	This example is similar evsoc_ispExample except it adds a frame counter and a timestamp to count the processing frame rates of the different scenarios.

The **main.c** file in the `src` subdirectory contains several functional blocks. You can change the software operation by commenting and uncommenting certain sections (described later in this document). The camera capture, RISC-V processing, and Hardware Accelerator sections are within a while loop, which executes iteratively to process the video stream.

Table 3: main.c Functional Blocks

Code Section	Function
DMA-Related Functions	Provides interrupt functions for the DMA Controller. This section only applies to the evsoc_ispExample_demo and evsoc_ispExample_demo2 examples.
Demo-Related Functions	Includes code to run the example via the UART or switches on the board. This section only applies to the evsoc_ispExample_demo and evsoc_ispExample_demo2 examples.
Setup PICAM & HDMI Display	Initializes the Raspberry Pi camera. Initializes the HDMI display. Sets the RGB gain values for the camera pre-processing block.

Code Section	Function
Setup DMA	Initializes the DMA controller. Sets the DMA priority for available the DMA channels (0 has highest priority).
Trigger Display	Initializes the test image display content (vertical colour bars a red dot in each corner) for display verification purposes. Indicates the memory source address for display data retrieval. Triggers the display MM2S DMA in self-restart mode (only once). Delays for 5 seconds. You will see a test image on the display for 5 seconds before entering video streaming mode.
Camera Capture	Selects the RGB or grayscale pixel output from the camera block. Indicates the memory destination address to store the camera data. Triggers the camera S2MM DMA and indicates when S2MM DMA initialization completes. Triggers capture/storage of one frame in the camera block.
RISC-V Processing	This code is commented out by default. Executes the filtering software functions. The code references the library **isp.h**, which contains the rgb2grayscale, sobel_edge_detection, binary_erosion, and binary dilation functions.
HW Accelerator	This code is commented out by default. Sets the threshold value for Sobel edge detection. Selects the hardware accelerator mode (Sobel only, Sobel plus dilation, or Sobel plus erosion). Indicates the memory source address for input data retrieval (MM2S). Selects the DMA transfer length based on the selected hardware accelerator mode. Triggers the hardware accelerator MM2S DMA. Indicates the memory destination address for storing the output data (S2MM). Triggers the hardware accelerator S2MM DMA and indicates when S2MM DMA initialization comples.
Check AXI4 Slave Status (HW Accelerator)	This code is commented out by default. Use these code snippets in other code segments to print the intermediate status of debug registers, for example, within the hardware accelerator section.
Check APB3 Slave Status (Camera & Display)	This code is commented out by default. Use these code snippets in other code segments to print the intermediate status of debug registers, for example, within the Trigger Display or Camera capture section.

About the Board Specific Package

The board specific package (BSP) defines the address map and aligns with the Ruby SoC hardware address map. The BSP files are located in the **bsp/efinix/EfxRubySoc** subdirectory.

Table 4: BSP Files

File or Directory	Description
app	Files used by the example software and bootloader.
include\soc.mk	Supported instruction set.
include\soc.h	Defines the system frequency and address map.
linker\default.ld	Linker script for the main memory address and size.
linker\bootloader.ld	Linker script for the bootloader address and size.
openocd	OpenOCD configuration files.

Note: If your binary is larger or smaller than the default size (124 KB), change the `LENGTH` parameter in **default.ld** to adjust for the different size and programming time.

```
MEMORY
{
  ram   (wxai!r) : ORIGIN = 0x00001000, LENGTH = 124K
}
```

Ruby Vision SoC Address Map

The following table shows the address map for the RISC-V processor.

Note: Because the address range might be updated, Efinix recommends that you always refer to the parameter name when referencing an address in firmware, not by the actual address. The parameter names are defined in **soc.h**.

Table 5: Default Address Map, Interrupt ID, and Cached Channels

The AXI user slave channel is in a cacheless region (I/O) for compatibility with AXI-Lite.

Device	Parameter	Size	Interrupt ID	Region
Off-chip DRAM	SYSTEM_DDR_BMB	3.5 GBytes	-	Cache
GPIO	SYSTEM_GPIO_0_IO_APB	4K	[0]: 12 [1]: 13	I/O
I^{2}C 0	SYSTEM_I2C_0_IO_APB	4K	8	I/O
I^{2}C 1	SYSTEM_I2C_1_IO_APB	4K	9	I/O
I^{2}C 2	SYSTEM_I2C_2_IO_APB	4K	10	I/O
Machine timer	SYSTEM_MACHINE_TIMER_APB	4K	31	I/O
PLIC	SYSTEM_PLIC_APB	4K	-	I/O
SPI master 0	SYSTEM_SPI_0_IO_APB	4K	4	I/O
SPI master 1	SYSTEM_SPI_1_IO_APB	4K	5	I/O
SPI master 2	SYSTEM_SPI_2_IO_APB	4K	6	I/O
UART 0	SYSTEM_UART_0_IO_APB	4K	1	I/O
UART 1	SYSTEM_UART_1_IO_APB	4K	2	I/O
User peripheral 0	IO_APB_SLAVE_0_APB	64K	-	I/O
User peripheral 1	IO_APB_SLAVE_1_APB	64K	-	I/O
User peripheral 2	IO_APB_SLAVE_2_APB	64K	-	I/O
User peripheral 3	IO_APB_SLAVE_3_APB	64K	-	I/O
On-chip BRAM	SYSTEM_RAM_A_BMB	4 KB	-	Cache
AXI user slave	SYSTEM_AXI_A_BMB	16 MB	-	I/O
External interrupt	-	-	25	I/O

Note: The RISC-V GCC compiler does not support user address spaces starting at 0x0000_0000.

Chapter 5

Using Eclipse and OpenOCD

Contents:

You use Eclipse to manage your software projects and OpenOCD for debugging. The following sections explain how to set up your Eclipse environment, create a project and build it. Additionally, this section walks you through how to set up the OpenOCD debugger for use with the Ruby Vision SoC.

Note: The instructions in this user guide assume you are using the RISC-V SDK v1.1 or higher.

Launch Eclipse

The RISC-V SDK includes the **run_eclipse.bat** file (Windows) or **run_eclipse.sh** file (Linux) that adds executables to your path, sets up envonment variables for the Ruby Vision BSP, and launches Eclipse. Always use this executable to launch Eclipse; do not launch Eclipse directly.

When you first start working with the Ruby Vision SoC, you need to configure your Eclipse workspace and environment. Setting up a global development environment for your workspace means you can store all of your Ruby software code in the same place and you can set global environment variables that apply to all software projects in your workspace.

You should use a unique workspace for your Ruby Vision SoC projects. Efinix recommends using the **evsoc** directory as the workspace directory.

Note: Efinix provides several RISC-V SoCs. Using the top-level SoC directory as your workspace means that you can explore more than one SoC by simply switching workspaces.

Follow these steps to launch Eclipse and set up your workspace:

1. Launch Eclipse using the **run_eclipse.bat** file (Windows) or **run_eclipse.sh** file.
2. The launch script prompts you to select your SoC. Type 1 for Ruby and press enter.
3. If this is the first time you are running Eclipse, create a new workspace that points to the **evsoc** directory. Otherwise, choose **File > Switch Workspace > Other** to choose an existing workspace directory and click **Launch**.

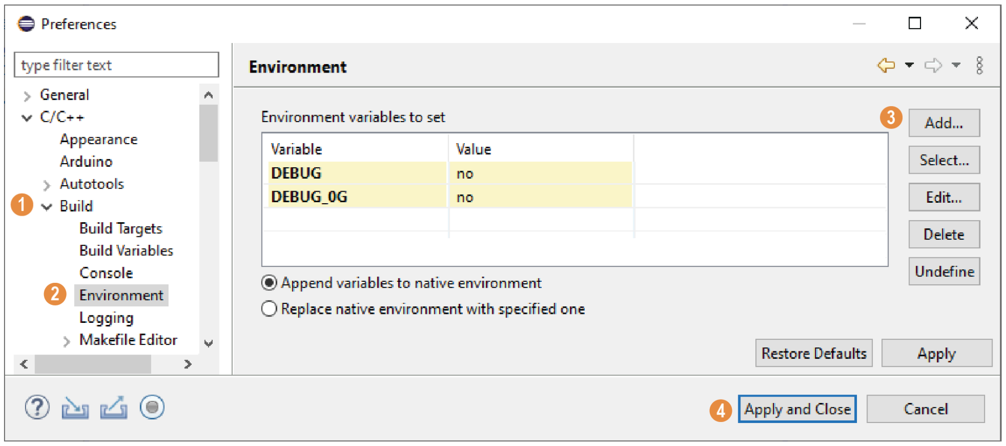
Set Up the Eclipse Workspace

When you first start working with the Ruby Vision SoC, you need to configure your Eclipse workspace and environment. Setting up a global development environment for your workspace means you can store all of your Ruby software code in the same place and you can set global environment variables that apply to all software projects in your workspace.

You should use a unique workspace for your Ruby Vision SoC projects. Efinix recommends using the **evsoc** directory as the workspace directory.

Note: Efinix provides several RISC-V SoCs. Using the top-level SoC directory as your workspace means that you can explore more than one SoC by simply switching workspaces.

1. Launch Eclipse.
2. If this is the first time you are running Eclipse, create a new workspace. Otherwise, choose **File > Switch Workspace > Other** to choose a new workspace directory and click **Launch**.
3. Create your workspace environment variables; these variables apply to all projects in your workspace. Choose **Window > Preferences** to open the **Preferences** window.



Environment variables to set

Variable	Value
DEBUG	no
DEBUG_0G	no

[✓ Append variables to native environment]
Replace native environment with specified one

a. In the left navigation menu, expand **C/C++ > Build**.
b. Click **C/C++ > Build > Environment**.
c. Click **Add** and add the following environment variables:

Variable	Value	Description
DEBUG	no	Enables or disables debug mode. no: Debugging is turned off yes: Debugging is enabled
DEBUG_OG	no	Enables or disables optimization during debugging. Use an uppercase letter O not a zero.

d. Click **Apply and Close**.

Create a New Project

In this step you create a new project from the **evsoc_ispExample** code example.

1. Launch Eclipse.
2. Select the Ruby Vision workspace if it is not open by default.
3. Make sure you are in the C/C++ perspective.

Import the **evsoc_ispExample** example:

4. Choose **File** > **New** > **Makefile Project with Existing Code**.
5. Click **Browse** next to **Existing Code Location**.
6. Browse to the **software/evsoc_ispExample** directory and click **Select Folder**.
7. Select **<none>** in the **Toolchain for Indexer Settings** box.
8. Click **Finish**.

Import Project Settings (Optional)

Efinix provides a C/C++ project settings file that defines the include paths and symbols for the C code. Importing these settings into your project lets you explore and jump through the code easily.

Note: You are not required to import the project settings to build. These settings simply make it easier for you to write and debug code.

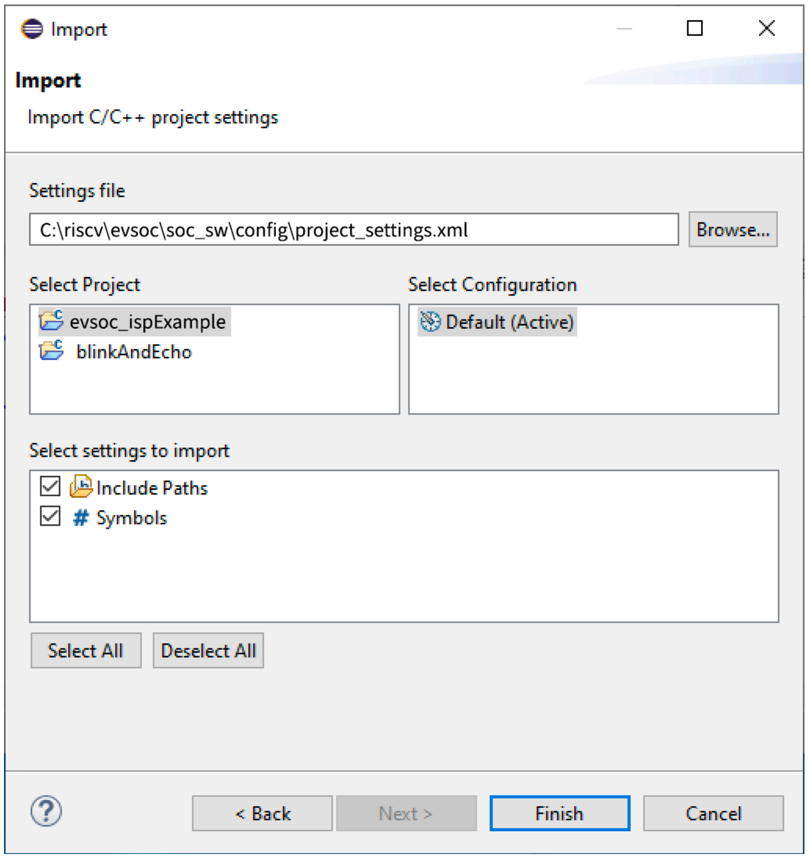
To import the settings:

1. Choose **File** > **Import** to open the **Import** wizard.
2. Expand **C/C++**.
3. Choose **C/C++** > **C/C++ Project Settings**.
4. Click **Next**.
5. Click **Browse** next to the **Settings file** box.
6. Browse to one of the following files and click **Open**:

Option	Description
Windows	evsoc\soc_sw\config\project_settings.xml
Linux	evsoc/soc_sw/config_linux/project_settings_linux.xml

7. In the **Select Project** box, select the project name(s) for which you want to import the settings.

8. Click **Finish**.


Import

Import

Import C/C++ project settings

Settings file

C:\riscv\evsoc\soc_sw\config\project_settings.xml

Select Project: evsoc_ispExample (selected), blinkAndEcho

Select Configuration: Default (Active)

Select settings to import:

☑ Include Paths
☑ Symbols

Eclipse creates a new folder in your project named **Includes**, which contains all of the files the project uses.

After you import the settings, clean your project (**Project > Clean**) and then build (**Project > Build Project**). The build process indexes all of the files so they are linked in your project.

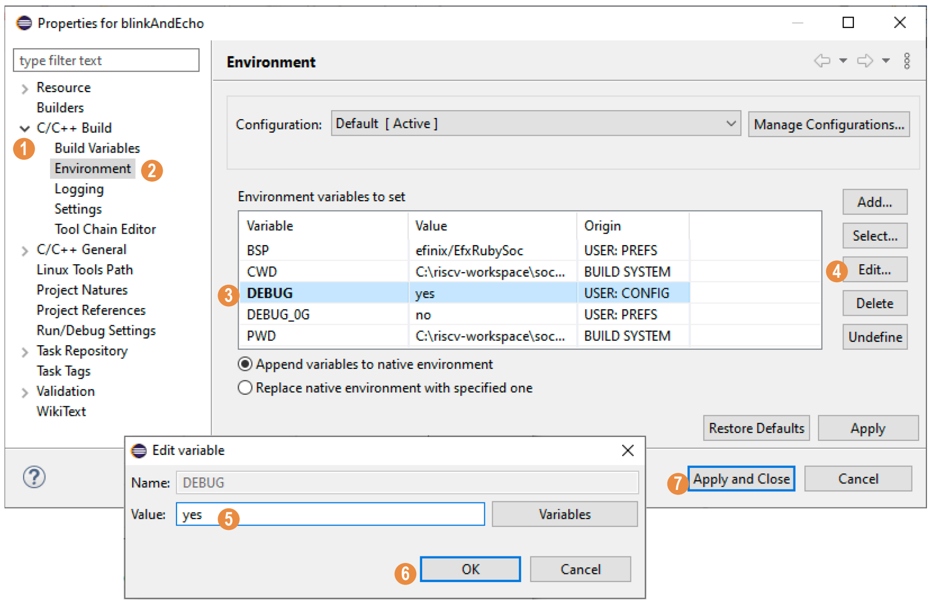
Enable Debugging

When you set up your workspace, you defined an environment variable for debugging with a default value of **no**.

- To run the program for normal operation, keep **DEBUG** set to **no**.
- To debug with the OpenOCD debugger, set **DEBUG** to **yes**.

In debug mode, the program suspends operation after loading so that you can set breakpoints or perform debug tasks.

To change the debug settings for your project, right-click the project name **evsoc_ispExample** in the Project Explorer and choose **Properties** from the pop-up menu.



Configuration: Default [Active]

Environment variables to set

Variable	Value	Origin
BSP	efinix/EfxRubySoc	USER: PREFS
CWD	C:\riscv-workspace\soc...	BUILD SYSTEM
DEBUG	yes	USER: CONFIG
DEBUG_OG	no	USER: PREFS
PWD	C:\riscv-workspace\soc...	BUILD SYSTEM

(●) Append variables to native environment
( ) Replace native environment with specified one

Name: DEBUG
Value: yes

1. Expand **C/C++ Build**.
2. Click **C/C++ Build > Environment**.
3. Click the **Debug** variable.
4. Click **Edit**.
5. Change the **Value** to `yes`.
6. Click **OK**.
7. Click **Apply and Close**.

Important: When you change the debug value for a project you previously built, you must clean the project (**Project** > **Clean**) before building again. Otherwise, Eclipse gives a message in the Console that there is `Nothing to be done for 'all'`

Build

Choose **Project > Build Project** or click the Build Project toolbar button.

The **makefile** builds the project and generates these files in the **build** directory:

- **evsoc_ispExample.asm**—Assembly language file.
- **evsoc_ispExample.bin**—Download this file to the flash device on your board using OpenOCD. When you turn the board on, the SoC loads the application into the RISC-V processor and executes it.
- **evsoc_ispExample.elf**—Use this file when debugging with the OpenOCD debugger.
- **evsoc_ispExample.hex**—Hex file.
- **evsoc_ispExample.map**—Contains the SoC address map.

Debug with the OpenOCD Debugger

With the Trion® T120 BGA324 Development Board programmed and the software built, you are ready to configure the OpenOCD debugger and perform debugging. These instructions use the **evsoc_ispExample** example to explain the steps required.

Important: If you want to use the OpenOCD Debugger at the same time as the Efinity® Debugger, you cannot use the same USB connection to the board because they conflict causing one of the applications to crash. Refer to Efinity Debugger Crashes when using OpenOCD on page 40 for information on using two USB cables to operate both debuggers simultaneously.

Import the Debug Configuration

To simplify the debugging steps, the Ruby Vision SoC includes a debug configuration that you import.

1. Connect your board to your computer using a JTAG cable. If you are using an Efinix development board, you can use a USB cable instead.
2. Launch Eclipse by running the **run_eclipse.bat** file (Windows) or **run_eclipse.sh** (Linux).
3. Select a workspace (if you have not set one as a default).
4. Open the **evsoc_ispExample** project or select it under **C/C++ Projects**.
5. Right-click the **evsoc_ispExample** project name and choose **Import**.
6. In the Import dialog box, choose **Run/Debug > Launch Configurations**.
7. Click **Next**. The Import Launch Configurations dialog box opens.
8. Browse to the following directory and click **OK**:

Option	Description
Windows	evsoc\soc_sw\config
Linux	evsoc/soc_sw/config_linux

9. Check the box next to **config** (Windows) or **config_linux** (Linux).
10. Click **Finish**.
11. Right-click the **evsoc_ispExample** project name and choose **Debug As > Debug Configurations**.
12. Choose **GDB OpenOCD Debugging > default**.
13. Enter `evsoc_ispExample` in the **Project** box.
14. Enter `build\evsoc_ispExample.elf` in the **C/C++ Application** box.
15. *Windows only:* you need to change the path to the **cpu0.yaml** file:
    a. Click the **Debugger** tab.
    b. In the **Config options** box, change `${workspace_loc}` to the full path to the **soc_sw** directory.

    Note: For the **cpu0.yaml** path, make sure to use \\ as the directory separator because the first slash escapes the second one. For example, use:
    c:\\\\riscv\\\\evsoc\\\\soc_sw\\\\cpu0.yaml
16. Click **Debug**.

Note: If Eclipse prompts you to switch to the Debug Perspective, click **Switch**.

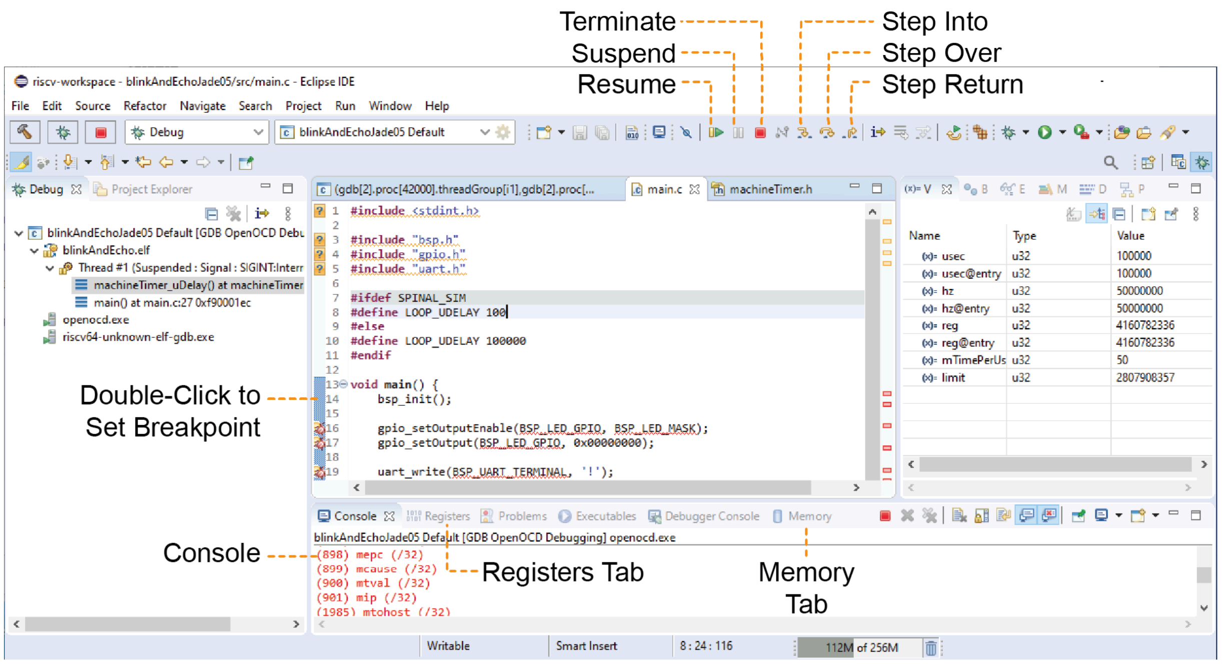
Debug

After you click **Debug** in the Debug Configuration window, the OpenOCD server starts, connects to the target, starts the gdb client, downloads the application, and starts the debugging session. Messages and a list of VexRiscv registers display in the **Console**. The **main.c** file opens so you can debug each step.

1. Click the **Resume** button or press F8 to resume code operation. LEDs D9 and D10 blink alternately.
2. Click **Step Over** (F6) to do a single step over one source instruction.
3. Click **Step Into** (F5) to do a single step into the next function called.
4. Click **Step Return** (F7) to do a single step out of the current function.
5. Double-click in the bar to the left of the source code to set a breakpoint. Double-click a breakpoint to remove it.
6. Click the **Registers** tab to inspect the processor's registers.
7. Click the **Memory** tab to inspect the memory contents.
8. Click the **Suspend** button to stop the code operation.
9. When you finish debugging, click **Terminate** to disconnect the OpenOCD debugger.

The evsoc_ispExample example blinks the LEDs and prints messages on a UART terminal. Refer to Using a UART Module for steps on setting it up.

Figure 5: Perform Debugging



Terminate
Suspend
Resume
Step Into
Step Over
Step Return
Double-Click to Set Breakpoint
Console
Registers Tab
Memory Tab
riscv-workspace - blinkAndEchoJade05/src/main.c - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
blinkAndEchoJade05 Default
Debug
Project Explorer
(gdb[2].proc[42000].threadGroup[i1],gdb[2].proc[...
main.c
machineTimer.h
Console
Registers
Problems
Executables
Debugger Console
Memory
Writable
Smart Insert
8 : 24 : 116
112M of 256M

```
blinkAndEchoJade05 Default [GDB OpenOCD Debu
  blinkAndEcho.elf
    Thread #1 (Suspended : Signal : SIGINT:Interr
      machineTimer_uDelay() at machineTimer
      main() at main.c:27 0xf90001ec
  openocd.exe
  riscv64-unknown-elf-gdb.exe
```

```
#include <stdint.h>

#include "bsp.h"
#include "gpio.h"
#include "uart.h"

#ifdef SPINAL_SIM
#define LOOP_UDELAY 100
#else
#define LOOP_UDELAY 100000
#endif

void main() {
    bsp_init();

    gpio_setOutputEnable(BSP_LED_GPIO, BSP_LED_MASK);
    gpio_setOutput(BSP_LED_GPIO, 0x00000000);

    uart_write(BSP_UART_TERMINAL, '!');
```

Name	Type	Value
usec	u32	100000
usec@entry	u32	100000
hz	u32	50000000
hz@entry	u32	50000000
reg	u32	4160782336
reg@entry	u32	4160782336
mTimePerUs	u32	50
limit	u32	2807908357

```
blinkAndEchoJade05 Default [GDB OpenOCD Debugging] openocd.exe
(898) mepc (/32)
(899) mcause (/32)
(900) mtval (/32)
(901) mip (/32)
(1985) mtohost (/32)
```

Learn more: For more information on debugging with Eclipse, refer to Running and debugging projects in the Eclipse documentation.

Using a UART Terminal

When working with the Image Signal Processing design in Eclipse, it is helpful to have a Telnet terminal open to send commands and view messages via the UART interface. The following sections explain how to enable Telnet in Windows and how to open a UART terminal.

Enable Telnet on Windows

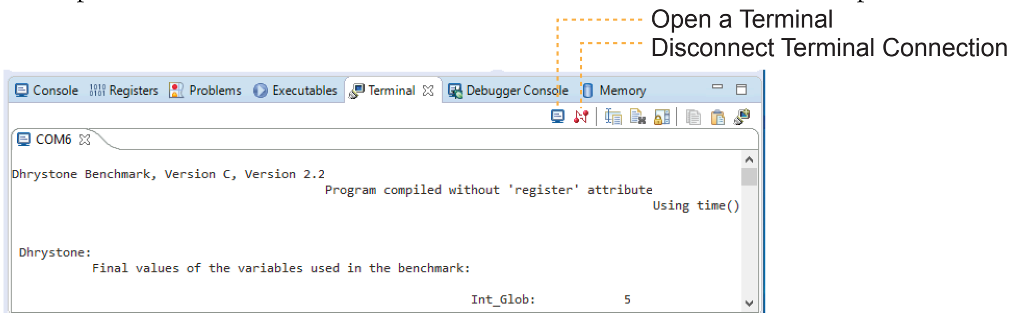
Windows does not have telnet turned on by default. Follow these instructions to enable it:

1. Type `telnet` in the Windows search box.
2. Click **Turn Windows features on or off (Control panel)**. The **Windows Features** dialog box opens.
3. Scroll down to **Telnet Client** and click the checkbox.
4. Click **OK**. Windows enables telnet.
5. Click **Close**.

Open a Terminal

You can use any terminal program, such as Putty, termite, or the built-in Eclipse terminal, to connect to the UART. These instructions explain how to use the Eclipse terminal; the others are similar.

1. In Eclipse, choose **Window** > **Show View** > **Terminal**. The Terminal tab opens.



```
Dhrystone Benchmark, Version C, Version 2.2
                                        Program compiled without 'register' attribute
                                                                          Using time()


 Dhrystone:
          Final values of the variables used in the benchmark:

                                                               Int_Glob:          5
```

2. Click the Open a Terminal button.
3. In the **Launch Terminal** dialog box, enter these settings:

Option	Setting
Choose terminal	Serial Terminal
Serial port	COM*n* (Windows) or ttyUSB*n* (Linux) where *n* is the port number for your UART module.
Baud rate	115200
Data size	8
Parity	None
Stop bits	1
Encoding	Default (ISO-8859-1)

4. Click **OK**. The terminal opens a connection to the UART.
5. Run your application. Messages are printed in the terminal.

6. When you are finished using the application, click the Disconnect Terminal Connection button.

View the Results

With the board programmed and the firmware downloaded to the RISC-V processor, you should see RGB video displayed on the HDMI display.

This scenario requires a frame buffer because the frame rate of incoming camera data from the MIPI interface varies depending on the camera driver setting while the output to the HDMI display is at 60 frames/s for compliance with the VESA display standard specifications. The frame buffer is implemented using main memory, and it stores frame data to memory and then retrieves it from memory for subsequent processes.

Chapter 6

Customizing the Firmware

Contents:

Once you have the basic out-of-box streaming video working, you can begin to customize software. The **main.c** file includes commented code that you can enable to perform various functions, some in hardware and some in software. After you change the **main.c** file, you must recompile the software and download the new firmware file (**.elf**) to the board. The following sections describe how to perform the customizations.

Grayscale and Sobel

This customization turns on the grayscale and Sobel filters. The RISC-V processor performs the grayscale conversion while the Hardware Accelerator performs the Sobel filtering. Make these changes in the **main.c** file using the Eclipse project you have already set up.

In the Trigger Display section, comment out the code for `cam_array` and uncomment the code for `sobel_array`:

```
/*
  //Initialize test image in cam_array
   for (int y=0; y<IMG_HEIGHT; y++) {
      for (int x=0; x<IMG_WIDTH; x++) {
         if ((x<3 && y<3) || (x>=IMG_WIDTH-3 && y<3) || (x<3 && y>=IMG_HEIGHT-3) ||
 (x>=IMG_WIDTH-3 && y>=IMG_HEIGHT-3)) {
            cam_array [y*IMG_WIDTH + x] = 0x000000FF; //RED
         } else if (x<(IMG_WIDTH/4)) {
            cam_array [y*IMG_WIDTH + x] = 0x0000FF00; //GREEN
         } else if (x<(IMG_WIDTH/4 *2)) {
            cam_array [y*IMG_WIDTH + x] = 0x00FF0000; //BLUE
         } else if (x<(IMG_WIDTH/4 *3)) {
            cam_array [y*IMG_WIDTH + x] = 0x000000FF; //RED
         } else {
            cam_array [y*IMG_WIDTH + x] = 0x00FF0000; //BLUE
         }
      }
   }
*/
   //Initialize test image in sobel_array
   for (int y=0; y<IMG_HEIGHT; y++) {
      for (int x=0; x<IMG_WIDTH; x++) {
         if ((x<3 && y<3) || (x>=IMG_WIDTH-3 && y<3) || (x<3 && y>=IMG_HEIGHT-3) ||
 (x>=IMG_WIDTH-3 && y>=IMG_HEIGHT-3)) {
            sobel_array [y*IMG_WIDTH + x] = 0x000000FF; //RED
         } else if (x<(IMG_WIDTH/4)) {
            sobel_array [y*IMG_WIDTH + x] = 0x0000FF00; //GREEN
         } else if (x<(IMG_WIDTH/4 *2)) {
            sobel_array [y*IMG_WIDTH + x] = 0x00FF0000; //BLUE
         } else if (x<(IMG_WIDTH/4 *3)) {
            sobel_array [y*IMG_WIDTH + x] = 0x000000FF; //RED
         } else {
            sobel_array [y*IMG_WIDTH + x] = 0x00FF0000; //BLUE
         }
      }
   }
```

Also in the Trigger Display section, comment the DMA source address for the camera and uncomment the one for the Sobel filter:

```
//SELECT start address of to be displayed data accordingly
//dmasg_input_memory(DMASG_BASE, DMASG_DISPLAY_MM2S_CHANNEL, CAM_START_ADDR, 16);
//dmasg_input_memory(DMASG_BASE, DMASG_DISPLAY_MM2S_CHANNEL, GRAYSCALE_START_ADDR, 16);
dmasg_input_memory(DMASG_BASE, DMASG_DISPLAY_MM2S_CHANNEL, SOBEL_START_ADDR, 16);
```

Perform these additional code changes:

1. Uncomment the whole RISC-V Processing section, which activates the `rgb2grayscale` function.
2. Uncomment the whole HW Accelerator section, which activates the Hardware Accelerator for Sobel filtering.
3. Clean and then build the project.
4. Press SW2 (reset) on the Trion® T120 BGA324 Development Board.

Note: You MUST reset the board before downloading the new **.elf** to the board. Otherwise the video will not display correctly.

5. Download the resulting **.elf** file to the board.

Place an object with obvious sharp color edges, for example a calendar, in front of the Raspberry Pi camera. The display shows the detected edges in a binary frame (black and white).

Grayscale, Sobel, and Dilation

This customization builds on the previous example. In addition to the grayscale and Sobel filering, this example adds the dilation function. The RISC-V processor performs the grayscale conversion while the Hardware Accelerator performs the Sobel filtering and dilation. Make these changes in the **main.c** file using the Eclipse project you have already set up.

Note: Make the code changes described in the Grayscale and Sobel on page 29 example if you have not already done so.

In the Hardware Accelerator section, comment out the `Sobel only - Default` line and uncomment the `Sobel+Dilation` line.

```
//SELECT HW accelerator mode - Make sure match with DMA transfer length setting
//write_u32(0x00000000, EXAMPLE_AXI4_SLV+EXAMPLE_AXI4_SLV_REG1_OFFSET);   //2'd0: Sobel only -
 Default
write_u32(0x00000001, EXAMPLE_AXI4_SLV+EXAMPLE_AXI4_SLV_REG1_OFFSET); //2'd1: Sobel+Dilation
//write_u32(0x00000002, EXAMPLE_AXI4_SLV+EXAMPLE_AXI4_SLV_REG1_OFFSET); //2'd2: Sobel+Erosion
```

Also in the Hardware Accelerator section, comment out the `Sobel only` line and uncomment the `Sobel + Dilation/Erosion` line.

```
//SELECT dma transfer length - Make sure match with HW accelerator mode selection
//Additonal data is required to be fed for line buffer(s) data flushing
//dmasg_direct_start(DMASG_BASE, DMASG_HW_ACCEL_MM2S_CHANNEL,
 ((IMG_WIDTH*IMG_HEIGHT)+(IMG_WIDTH+1))*4, 0);      //Sobel only
dmasg_direct_start(DMASG_BASE, DMASG_HW_ACCEL_MM2S_CHANNEL,
 ((IMG_WIDTH*IMG_HEIGHT)+(2*IMG_WIDTH+2))*4, 0); //Sobel + Dilation/Erosion
```

Press SW2 (reset) on the Trion® T120 BGA324 Development Board.

Clean and rebuild the project, then download the resulting **.elf** file to the board.

Place the same object you used in the Grayscale and Sobel on page 29 example in front of the Raspberry Pi camera to see the difference. The display shows the detected edges with bolder, thicker lines.

Grayscale, Sobel, and Erosion

This customization builds on the previous example. In addition to the grayscale and Sobel filering, this example adds the erosion function. The RISC-V processor performs the grayscale conversion while the Hardware Accelerator performs the Sobel filtering and erosion. Make these changes in the **main.c** file using the Eclipse project you have already set up.

Note: Make the code changes described in the Grayscale and Sobel on page 29 example if you have not already done so.

In the Hardware Accelerator section, comment out the `Sobel only - Default` line and uncomment the `Sobel+Erosion` line.

```
//SELECT HW accelerator mode - Make sure match with DMA transfer length setting
//write_u32(0x00000000, EXAMPLE_AXI4_SLV+EXAMPLE_AXI4_SLV_REG1_OFFSET);   //2'd0: Sobel only -
 Default
//write_u32(0x00000001, EXAMPLE_AXI4_SLV+EXAMPLE_AXI4_SLV_REG1_OFFSET); //2'd1: Sobel+Dilation
write_u32(0x00000002, EXAMPLE_AXI4_SLV+EXAMPLE_AXI4_SLV_REG1_OFFSET); //2'd2: Sobel+Erosion
```

If you have not already done so, in the Hardware Accelerator section, comment out the `Sobel only` line and uncomment the `Sobel + Dilation/Erosion` line.

```
//SELECT dma transfer length - Make sure match with HW accelerator mode selection
//Additonal data is required to be fed for line buffer(s) data flushing
//dmasg_direct_start(DMASG_BASE, DMASG_HW_ACCEL_MM2S_CHANNEL,
 ((IMG_WIDTH*IMG_HEIGHT)+(IMG_WIDTH+1))*4, 0);     //Sobel only
dmasg_direct_start(DMASG_BASE, DMASG_HW_ACCEL_MM2S_CHANNEL,
 ((IMG_WIDTH*IMG_HEIGHT)+(2*IMG_WIDTH+2))*4, 0); //Sobel + Dilation/Erosion
```

Press SW2 (reset) on the Trion® T120 BGA324 Development Board.

Clean and rebuild the project, then download the resulting **.elf** file to the board.

Place the same object you used in the Grayscale and Sobel on page 29 example in front of the Raspberry Pi camera to see the difference. The display shows the detected edges with fainter, thinner lines.

Chapter 7

Using Your Own Hardware Accelerator

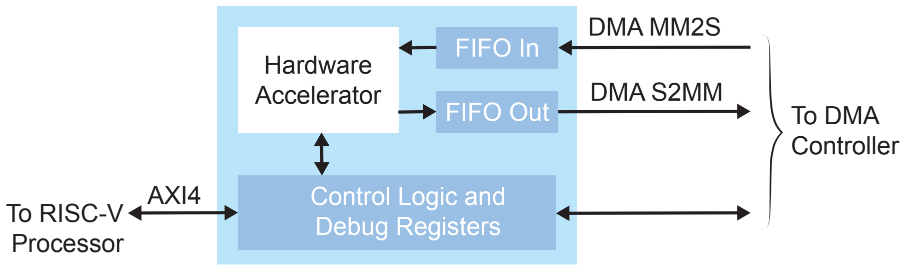
The Image Signal Processing example has a Hardware Accelerator that performs filtering functions. The top-level wrapper (**hw_accel_wrapper.v**) comprises the Hardware Accelerator, FIFO buffers, and control logic and debug registers. The wrapper has ports that connect to the DMA Controller and an AXI4 slave module.

The slave interface is used for communication between the RISC-V processor and Hardware Accelerator, mainly for debug and control purposes. Although this design uses an AXI4 slave interface, you can change the wrapper to use other slave interfaces such as APB3 or AXI4-lite.

- For an example APB3 interface, refer to the apb3_cam.v file. This file implements an APB3 slave module that is shared by the camera and display blocks for communication with the RISC-V processor.
- For an example AXI4-lite interface, refer to the APB3 to AXI4-Lite Core in the Support Center.

The DMA Controller facilitates the input/output data stream between the Hardware Accelerator and main memory. FIFOs handle any differences between the input/output data rate and Hardware Accelerator data processing rate. Control logic in the wrapper file generates the FIFOs and DMA read/write related signals, according to the behaviour of the Hardware Accelerator (for example, the required input/output data patterns or flow, firmware setup for DMA transfers, etc).

Figure 6: Hardware Accelerator Connections



If you want to drop in your own custom accelerator, you may need to make adjustments to the wrapper and to the firmware. The following sections describes areas you should consider.

Slave Interface

First, decide which type of slave interface you want to use, AXI4, APB3, or AXI4-lite. Next, determine what communication you need between the RISC-V processor and Hardware Accelerator for for setup control and debug registers. For example, you can use the RISC-V processor to trigger the start of computations and handshaking, set up different operating modes, or probe critical signals for debugging purposes. Finally, update the control logic and debug registers in the Hardware Accelerator wrapper, according to your design.

DMA Interface

The first step is to determine the input(s) and output(s) data requirements. The DMA Controller uses AXI-ST interfaces to the Hardware Accelerator. By default, the design has 32-bit input and output streams to retrieve data and store it from/to the main memory.

The DMA Controller supports asynchronous mode for individual channels. If you need the input/output stream to run at a different clock rate, update the clock signal connection to the respective DMA Controller channel(s). (See DMA Controller on page 12.)

FIFO

Determine the FIFO requirements (data width, depth, mode, etc.) and the read/write behaviours based on the needs of your custom hardware accelerator, and modify the FIFOs accordingly. Next, examine potential FIFO underflows and overflows and adjust the FIFO control signals or DMA transfer settings accordingly.

RISC-V Firmware

After modifying RTL side of the design, you must update the RISC-V firmware to match your changes. For example, modify **axi4_hw_accel.h** if you add more addressing offset.

If you use a different slave interface, create a corresponding header for ease of access in **main.c**. Refer to the Camera Capture, HW Accelerator, Check AXI4 Slave Status (HW Accelerator), and Check APB3 Slave Status (Camera & Display) sections in **main.c** for example APB3 and AXI4 slave interface usage. The following code snippet shows some example read/write usage:

```
//Write to AXI4 slave - SET Sobel edge detection threshold via AXI4 slave
write_u32(100, EXAMPLE_AXI4_SLV+EXAMPLE_AXI4_SLV_REG0_OFFSET); //Default value 100; Range 0 to
 255
//Read from AXI4 slave - RETRIEVE HW accelerator DMA FIFOs status
rdata = axi_slave_read32(EXAMPLE_AXI4_SLV+EXAMPLE_AXI4_SLV_REG4_OFFSET);

//Write to APB3 slave - SELECT RGB or grayscale output from camera pre-processing block.
EXAMPLE_APB3_REGW(EXAMPLE_APB3_SLV, EXAMPLE_APB3_SLV_REG3_OFFSET, 0x00000000); //RGB
//Read from APB3 slave - RETRIEVE camera & display DMA FIFOs status
rdata = example_register_read(EXAMPLE_APB3_SLV_REG6_OFFSET);
```

When you use a custom hardware accelerator, you should update the DMA transfer setting in the firmware. For example, update the DMA transfer length per captured frame based on the accelerator's behaviour. (The third argument of `dmasg_direct_start()` is the transfer length in number of bytes.) By default, the example design uses direct mode DMA transfer.

```
//For HW accel MM2S (fetch data from main memory to HW accel building block)
dmasg_direct_start(DMASG_BASE, DMASG_HW_ACCEL_MM2S_CHANNEL, ((IMG_WIDTH*IMG_HEIGHT)+(IMG_WIDTH
+1))*4, 0);

//For HW accel S2MM (store data from HW accel building block to main memory)
dmasg_direct_start(DMASG_BASE, DMASG_HW_ACCEL_S2MM_CHANNEL, (IMG_WIDTH*IMG_HEIGHT)*4, 0);
```

Chapter 8

Deploying an Application Binary

Contents:

During normal operation, your user binary application file (**.bin**) is stored in a SPI flash device. When the FPGA powers up, the Ruby Vision SoC copies your binary file from the SPI flash device to the DDR DRAM module, and then begins execution.

For debugging, you can load the user binary (**.elf**) directly into the Ruby Vision SoC using the OpenOCD Debugger. After loading, the binary executes immediately.

Note: The settings in the linker prevent user access to the DDR DRAM address. This setting allows the embedded bootloader to work properly during a system reset after the user binary is executed but the FPGA is not reconfigured.

Boot from a Flash Device

When the FPGA boots up, the Ruby SoC copies your binary application file from a SPI flash device to the DDR DRAM module, and then begins execution. The SPI flash binary address starts at `0x0038_0000`.

To boot from a SPI flash device:

1. Power up your board. The FPGA loads the configuration image from the on-board flash device.
2. When configuration completes, the bootloader begins cloning a 124 KByte user binary file from the flash device at physical address `0x0038_0000` to an off-chip DRAM logical address of `0x0000_1000`.

   Note: It takes ~300 ms to clone a 124 KByte user binary (this is the default size).

3. The Ruby Vision SoC jumps to logical address `0x0000_1000` to execute the user binary.

Boot from the OpenOCD Debugger

To boot from the OpenOCD debugger:

1. Power up your board. The FPGA loads the configuration image from the on-board flash device.
2. Launch Eclipse and set up the debug environment for your project.
3. When you click **Debug**, the debugger sends a soft reset to the SoC, and then writes the user binary file to logical address `0x0000_1000`, which is the starting address of the DDR memory.
4. The Ruby Vision SoC jumps to logical address `0x0000_1000` to execute the user binary.
5. The user binary is suspended on boot up. Click the Resume button to start the program.

Note: Refer to Debug with the OpenOCD Debugger on page 25 for complete instructions on debugging.

Copy a User Binary to the Flash Device

To boot from a flash device, you need to copy the binary to the device. These instructions decribe how to use a command prompt or shell to flash the user binary file. You use two command prompts or shells:

- The first terminal opens an OpenOCD connection to the SoC.
- The second connects to the first terminal to write to the flash.

Important: If you are using the OpenOCD debugger in Eclipse, terminate any debug processes before attempting to flash the memory.

Set Up Terminal 1

1. Open a Windows command prompt or Linux shell.
2. Change to **SDK_Windows** or **SDK_Ubuntu**.
3. Execute the **setup.bat** (Windows) or **setup.sh** (Linux) script.
4. Change to the **soc_Ruby_sw** directory.
5. Type the following commands to set up the OpenOCD server:

   Windows:

   ```
   openocd.exe -f bsp\efinix\EfxRubySoc\openocd\ftdi.cfg
      -c "set CPU0_YAML cpu0.yaml"
      -f bsp\efinix\EfxRubySoc\openocd\flash.cfg
   ```

   Linux:

   ```
   openocd -f bsp/efinix/EfxRubySoc/openocd/ftdi.cfg
      -c "set CPU0_YAML cpu0.yaml"
      -f bsp/efinix/EfxRubySoc/openocd/flash.cfg
   ```

   The OpenOCD server connects and begins listening on port 4444.

Set Up Terminal 2

1. Open a second command prompt or shell.
2. Enable telnet if it is not turned on. Turn on telnet (Windows)
3. Open a telnet local host on port 4444 with the command `telnet localhost 4444`.
4.
5. In the OpenOCD shell or command prompt, use the following command to flash the user binary file:

   ```
   flash write_image erase unlock <path>/<filename>.bin 0x380000
   ```

   Where <*path*> is the full, absolute path to the **.bin** file.

   Note: For Windows, use \\ as the directory separators.

Close Terminals

When you finish:

- Type `exit` in terminal 2 to close the telnet session.
- Type Ctrl+C in terminal 1 to close the OpenOCD session.

Important: OpenOCD cannot be running in Eclipse when you are using it in a terminal. If you try to run both at the same time, the application will crash or hang. Always close the terminals when you are done flashing the binary.

Reset the FPGA

Press the reset button (SW2) on the Trion® T120 BGA324 Development Board.

Chapter 9

Troubleshooting

Contents:

Error 0x80010135: Path too long (Windows)

When you unzip the SDK on Windows, you may get the error message:

```
An unuexpected error is keeping you from copying the file. If you continue
to receive this error, you can use the error code to search for help with
this problem.

Error 0x80010135: Path too long
```

This error occurs if you try to unzip the SDK files into a deep folder hierarchy instead of one that is close to the root level. Instead unzip to **c:\riscv-sdk**.

OpenOCD Error: timed out while waiting for target halted

The OpenOCD debugger console may display this error when:
- There is a bad contact between the FPGA header pins and the programming cable.
- The FPGA is not configured with a Ruby Vision SoC design.
- You may not have the correct PLL settings to work with the Ruby Vision SoC.
- Your computer does not have enough memory to run the program.

To solve this problem:
- Make sure that all of the cables are securly connected to the board and your computer.
- Ensure that you have placed a jumper on J10 connecting pins 1 and 2. This jumper enables the 10 MHz on-board oscillator. Refer to Enable the On-Board 10 MHz Oscillator.
- Check the JTAG connection.
- Ensure that the FPGA is programmed with the Ruby Vision SoC. Refer to Program the Development Board.

Memory Test

Your user binary may not boot correctly if there is a memory corruption problem (that is, the communication between the DDR hard controller and memory module is not functioning). This issue can appear when booting using the SPI flash or OpenOCD debugger. The following instructions provide a debugging flow to determine whether you system has this problem. You use two command prompts or shells to perform the test:

- The first terminal opens an OpenOCD connection to the SoC.
- The second connects to the first terminal for performing the test.

Important: If you are using the OpenOCD debugger in Eclipse, terminate any debug processes before performing this test.

Set Up Terminal 1

1. Open a Windows command prompt or Linux shell.
2. Change to **SDK_Windows** or **SDK_Ubuntu**.
3. Execute the **setup.bat** (Windows) or **setup.sh** (Linux) script.
4. Change to the **soc_Ruby_sw** directory.
5. Type the following commands to set up the OpenOCD server:

   Windows:

   ```
   openocd.exe -f bsp\efinix\EfxRubySoc\openocd\ftdi.cfg
      -c "set CPU0_YAML cpu0.yaml"
      -f bsp\efinix\EfxRubySoc\openocd\flash.cfg
   ```

   Linux:

   ```
   openocd -f bsp/efinix/EfxRubySoc/openocd/ftdi.cfg
      -c "set CPU0_YAML cpu0.yaml"
      -f bsp/efinix/EfxRubySoc/openocd/flash.cfg
   ```

   The OpenOCD server connects and begins listening on port 4444.

Set Up Terminal 2

1. Open a second command prompt or shell.
2. Enable telnet if it is not turned on. Turn on telnet (Windows)
3. Open a telnet host on port 4444 with the command `telnet localhost 4444`.
4. To test the on-chip RAM, use the `mdw` command to get the bootloader binary. Type the command `mdw` <*address*> <*number of 32-bit words*> to display the content of the memory space. For example: `mdw 0xF900_0000 32`.
5. To test the DRAM:
   - Use the `mww` command to write to the memory space: `mww` <*address*> <*data*>. For example: `mww 0x00001000 16`.
   - Then, use the `mdw` command to write to the memory space: `mdw` <*address*> <*data*>. For example: `mdw 0x00001000 16`. If the memory space has collapsed, the console shows all `0`s.

Close Terminals

When you finish:

- Type `exit` in terminal 2 to close the telnet session.
- Type Ctrl+C in terminal 1 to close the OpenOCD session.

Important: OpenOCD cannot be running in Eclipse when you are using it in a terminal. If you try to run both at the same time, the application will crash or hang. Always close the terminals when you are done flashing the binary.

Reset the FPGA

Press the reset button (SW2) on the Trion® T120 BGA324 Development Board.

OpenOCD error code (-1073741515)

The OpenOCD debugger may fail with error code -1073741515 if your system does not have the **libusb0.dll** installed. To fix this problem, install the DLL. This issue only affects Windows systems.

OpenOCD Error: no device found

The FTDI driver included with the Ruby SoC specifies the FTDI device VID and PID, and board description. In some cases, an early revision of the Efinix development board may have a different name than the one given in the driver file. If the board name does not match the name in the driver, Open OCD will fail with an error similar to the following:

```
Error: no device found
Error: unable to open ftdi device with vid 0403, pid 6010, description 'Trion T20 Development
   Board', serial '*' at bus location '*'
```

To fix this problem, follow these steps with the development board attached to the computer:

1. Open the Efinity Programmer.
2. Click the **Refresh USB Targets** button to display the board name in the **USB Target** drop-down list.
3. Make note of the board name.
4. In a text editor, open the file **soc_Ruby_sw/bsp/efinix/EFXRubySoC/openocd/ftdi.cfg**.
5. Change the `ftdi_device_desc` setting to match your board name. For example, use this code to change the name from Trion T20 Development Board to Trion T20 Developer Board:

```
interface ftdi
ftdi_device_desc "Trion T20 Developer Board"
#ftdi_device_desc "Trion T20 Development Board"
ftdi_vid_pid 0x0403 0x6010
```

6. Save the file.
7. Debug as usual in OpenOCD.

OpenOCD Error: failed to reset FTDI device: LIBUSB_ERROR_IO

This error is typically caused because you have the wrong Windows USB driver for the development board. If you hav e the wrong driver, you will get an error similar to:

```
Error: failed to reset FTDI device: LIBUSB_ERROR_IO
Error: unable to open ftdi device with vid 0403, pid 6010, description
'Trion T20 Development Board', serial '*' at bus location '*'
```

Important: Efinix recommends using the **libusbK** driver, which you install using the Zadig software. Refer to Installing USB Drivers on page 9

OpenOCD Error: target 'fpga_spinal.cpu0' init failed

You may receive this error when trying to debug after creating your OpenOCD debug configuration. The Eclipse Console gives an error message similar to:

```
Error cpuConfigFile C:RiscVsoc_Jadesoc_jade_swcpu0.yaml not found
Error: target 'fpga_spinal.cpu0' init failed
```

This error occurs because the path to the **cpu0.yaml** file is incorrect, specifically the slashes for the directory separators. You should use:

- a single forward slash (/)
- 2 backslashes (\\)

For example, either of the following are good:

```
C:\\RiscV\\soc_Jade\\soc_jade_sw\\cpu0.yaml
C:/RiscV/soc_Jade/soc_jade_sw/cpu0.yaml
```

Eclipse Fails to Launch with Exit Code 13

The Eclipse software requires a 64-bit version of the Java JRE. If you use a 32-bit version, when you try to launch Eclipse you will get an error that Java quit with exit code 13.

If you are downloading the JRE using a web browser from www.java.com, it defaults to getting the 32-bit version. Instead, go to https://www.java.com/en/download/manual.jsp to download the 64-bit version.

Efinity® Debugger Crashes when using OpenOCD

The Efinity® Debugger crashes if you try to use it for debugging while also using OpenOCD. Both applications use the same USB connection to the development board, and conflict if you use them at the same time. To avoid this issue:

- Do not use the two debuggers at the same time.

- Use an FTDI cable and a soft JTAG core for OpenOCD debugging. See **Using a Soft JTAG Core instead of the JTAG User Tap** for details.

Chapter 10

API Reference

Contents:

The following sections describe the API for the code in the **driver** directory.

Control and Status Registers

csr_clear()

Usage	`csr_clear(csr, val)`
Include	**driver/riscv.h**
Description	Clear a CSR.

csr_read()

Usage	`csr_read(csr)`
Include	**driver/riscv.h**
Description	Read from a CSR.
Example	`csrr (t0, mepc) // Write mepc in regfile[t0]`

csr_read_clear()

Usage	`csr_read_clear(csr, val)`
Include	**driver/riscv.h**
Description	CSR read and clear bit.

csr_read_set()

Usage	`csr_read_set(csr, val)`
Include	**driver/riscv.h**
Description	CSR read and set bit.

csr_set()

Usage	`csr_set(csr, val)`
Include	**driver/riscv.h**
Description	CSR set bit.

csr_swap()

Usage	`csr_write(csr, val)`
Include	**driver/riscv.h**
Description	Swaps values in the CSR.

csr_write()

Usage	`csr_write(csr, val)`
Include	**driver/riscv.h**
Description	Write to a CSR.
Example	`csrw (mepc, t0); // Write regfile[t0] in mepc`

GPIO API Calls

gpio_getInput()

Usage	`gpio_getInput(GPIO_Reg, value)`
Parameters	[IN] GPIO_Reg struct of GPIO register [IN] value GPIO pin bitwise
Include	**driver/gpio.h**
Description	Get input from a GPIO.

gpio_getOutput()

Usage	`gpio_getOutput(GPIO_Reg, value)`
Parameters	[IN] GPIO_Reg struct of GPIO register [IN] value GPIO pin bitwise
Include	**driver/gpio.h**
Description	Read the output pin.

gpio_setOutput()

Usage	`gpio_setOutput(GPIO_Reg, value)`
Parameters	[IN] GPIO_Reg struct of GPIO register [IN] value GPIO pin bitwise
Include	**driver/gpio.h**
Description	Set GPIO as 1 or 0.

gpio_setOutputEnable()

Usage	`gpio_setOutputEnable(GPIO_Reg, value)`
Parameters	[IN] GPIO_Reg struct of GPIO register [IN] value GPIO pin bitwise
Include	**driver/gpio.h**
Description	Set GPIO as an output enable.

gpio_getOutputEnable()

Usage	`gpio_getOutputEnable(GPIO_Reg, value)`
Parameters	[IN] GPIO_Reg struct of GPIO register [IN] value GPIO pin bitwise
Include	**driver/gpio.h**
Description	Read GPIO output enable.

gpio_setInterruptRiseEnable()

Usage	`gpio_etInterruptRiseEnable(GPIO_Reg, value)`
Parameters	[IN] GPIO_Reg struct of GPIO register [IN] value GPIO pin bitwise
Include	**driver/gpio.h**
Description	Set an interrupt on the rising edge of the GPIO.

gpio_setInterruptFallEnable()

Usage	`gpio_setInterruptFallEnable(GPIO_Reg, value)`
Parameters	[IN] GPIO_Reg struct of GPIO register [IN] value GPIO pin bitwise
Include	**driver/gpio.h**
Description	Set an interrupt on the falling edge of the GPIO.

gpio_setInterruptHighEnable()

Usage	`gpio_setInterruptHighEnable(GPIO_Reg, value)`
Parameters	[IN] GPIO_Reg struct of GPIO register [IN] value GPIO pin bitwise
Include	**driver/gpio.h**
Description	Set an interrupt when the GPIO is high.

gpio_setInterruptLowEnable()

Usage	`gpio_setInterruptLowEnable(GPIO_Reg, value)`
Parameters	[IN] GPIO_Reg struct of GPIO register [IN] value GPIO pin bitwise
Include	**driver/gpio.h**
Description	Set an interrupt when the GPIO is low.

I^{2}C API Calls

i2c_applyConfig()

Usage	`void i2c_applyConfig(u32 reg, I2c_Config *config)`
Parameters	[IN] `reg` struct of I^{2}C setting value [IN] `config` struct of I^{2}C configuration
Include	**driver/i2c.h**
Description	Apply I^{2}C configuration to register or for initial configuration.

i2c_clearInterruptFlag()

Usage	`void i2c_clearInterruptFlag(u32 reg, u32 value)`
Parameters	[IN] `reg` struct of I^{2}C setting value [IN] `value` I^{2}C interrupt register
Include	**driver/i2c.h**
Description	Clear the I^{2}C interrupt flag.

i2c_disableInterrupt()

Usage	`void i2c_disableInterrupt(u32 reg, u32 value)`
Parameters	[IN] `reg` struct of I^{2}C setting value [IN] `value` I^{2}C interrupt register
Include	**driver/i2c.h**
Description	Disable I^{2}C interrupt.

i2c_enableInterrupt()

Usage	`void i2c_enableInterrupt(u32 reg, u32 value)`
Parameters	[IN] `reg` struct of I^{2}C setting value [IN] `value` I^{2}C interrupt register
Include	**driver/i2c.h**
Description	Enable I^{2}C interrupt.

i2c_filterEnable()

Usage	`void i2c_filterEnable(u32 reg, u32 filterId, u32 config)`
Parameters	[IN] `reg` struct of I^{2}C setting value [IN] `filterID` filter configuration ID number [IN] `config` struct of I^{2}C configuration
Include	**driver/i2c.h**
Description	Enable the filter configuration.

i2c_listenAck()

Usage	`void i2c_listenAck(u32 reg)`
Parameters	[IN] `reg` struct of I^{2}C register
Include	**driver/i2c.h**
Description	Listen acknowledge from the slave.

i2c_masterBusy()

Usage	`void i2c_masterBusy(u32 reg)`
Parameters	[IN] `reg` struct of I^{2}C register
Include	**driver/i2c.h**
Description	Get the I^{2}C busy status.

i2c_masterDrop()

Usage	`void i2c_masterDrop(u32 reg)`
Parameters	[IN] `reg` struct of I^{2}C register
Include	**driver/i2c.h**
Description	Change the I^{2}C master to the drop state.

i2c_masterStart()

Usage	`void i2c_masterStart(u32 reg)`
Parameters	[IN] `reg` struct of I^{2}C register
Include	**driver/i2c.h**
Description	Change the I^{2}C master to the start status.

i2c_masterStartBlocking()

Usage	`void i2c_masterStartBlocking(u32 reg)`
Parameters	[IN] `reg` struct of I^{2}C register
Include	**driver/i2c.h**
Description	Asserts a start condition.

i2c_masterStop()

Usage	`void i2c_masterStop(u32 reg)`
Parameters	[IN] `reg` struct of I^{2}C register
Include	**driver/i2c.h**
Description	Change the I^{2}C master to the stop status.

i2c_masterStopBlocking()

Usage	`void i2c_masterStartBlocking(u32 reg)`
Parameters	[IN] `reg` struct of I^{2}C register
Include	**driver/i2c.h**
Description	Asserts a stop condition.

i2c_masterStopWait()

Usage	`void i2c_masterStopWait(u32 reg)`
Parameters	[IN] `reg` struct of I^{2}C register
Include	**driver/i2c.h**
Description	The stop condition is wait busy..

i2c_setFilterConfig()

Usage	`void i2c_setFilterConfig(u32 reg, u32 filterId, u32 config)`
Parameters	[IN] `reg` struct of I^{2}C setting value [IN] `filterID` filter configuration ID number [IN] `config` struct of I^{2}C configuration
Include	**driver/i2c.h**
Description	Set the filter configuration.

i2c_txAck()

Usage	`void i2c_txAck(u32 reg)`
Parameters	[IN] `reg` struct of I^{2}C register
Include	**driver/i2c.h**
Description	Transmit acknowledge.

i2c_txAckBlocking()

Usage	`void i2c_txAckBlocking(u32 reg)`
Parameters	[IN] `reg` struct of I^{2}C register
Include	**driver/i2c.h**
Description	Assert an ACK on the SDA pin.

i2c_txAckWait()

Usage	`void i2c_txAckWait(u32 reg)`
Parameters	[IN] `reg` struct of I^{2}C register
Include	**driver/i2c.h**
Description	Wait for an acknowledge to transmit.

i2c_txByte()

Usage	`void i2c_txByte(u32 reg, u8 byte)`
Parameters	[IN] `reg` struct of I^{2}C register [IN] `byte` 8 bits data to send out
Include	**driver/i2c.h**
Description	Transfers one byte to the I^{2}C slave.

i2c_txByteRepeat()

Usage	`void i2c_txByteRepeat(u32 reg, u8 byte)`
Parameters	[IN] `reg` struct of I^{2}C register [IN] `byte` 8 bits data to send out
Include	**driver/i2c.h**
Description	Send a byte and then wait until it is fully transmited on the I^{2}C bus.

i2c_txNack()

Usage	`void i2c_txNack(u32 reg)`
Parameters	[IN] `reg` struct of I^{2}C register
Include	**driver/i2c.h**
Description	Transfers a NACK.

i2c_txNackRepeat()

Usage	`void i2c_txNackRepeat(u32 reg)`
Parameters	[IN] `reg` struct of I^{2}C register
Include	**driver/i2c.h**
Description	Send a NACK and then wait until it is fully transmited on the I^{2}C bus.

i2c_txNackBlocking()

Usage	`void i2c_ txNackBlocking(u32 reg)`
Parameters	[IN] `reg` struct of I^{2}C register
Include	**driver/i2c.h**
Description	Assert a NACK on the SDA pin.

i2c_rxAck()

Usage	`int i2c_rxAck(u32 reg)`
Parameters	[IN] `reg` struct of I^{2}C register
Returns	[OUT] 1 bit acknowledge
Include	**driver/i2c.h**
Description	Receive an acknowledge from the I^{2}C slave.

i2c_rxData()

Usage	`unit32_t i2c_rxData(u32 reg)`
Parameters	[IN] `reg` struct of I^{2}C register
Returns	[OUT] 1 byte data from I^{2}C slave
Include	**driver/i2c.h**
Description	Receive one byte data from I^{2}C slave.

i2c_rxNack()

Usage	`int i2c_rxNack(u32 reg)`
Parameters	[IN] `reg` struct of I^{2}C register
Returns	[OUT] 1 bit no acknowledge
Include	**driver/i2c.h**
Description	Receive no acknowledge from the I^{2}C slave.

I/O API Calls

read_u8()

Usage	`u8 read_u8(u32 address)`
Include	**driver/io.h**
Parameters	[IN] `address` SoC address
Description	Read address with unsigned 8 bits.

read_u16()

Usage	`u16 read_u16(u32 address)`
Include	**driver/io.h**
Parameters	[IN] `address` SoC address
Description	Read address with unsigned 16 bits.

read_u32()

Usage	`u32 read_u32(u32 address)`
Include	**driver/io.h**
Parameters	[IN] `address` SoC address
Description	Read address with unsigned 32 bits.

write_u8()

Usage	`void write_u8(u8 data, u32 address)`
Include	**driver/io.h**
Parameters	[IN] `data` SoC address data [IN] `address` SoC address
Description	Write 8 bits unsigned data to the specified address.

write_u16()

Usage	`void write_u16(u16 data, u32 address)`
Include	**driver/io.h**
Parameters	[IN] `data` SoC address data [IN] `address` SoC address
Description	Write 16 bits unsigned data to the specified address.

write_u32()

Usage	`void write_u32(u32 data, u32 address)`
Include	**driver/io.h**
Parameters	[IN] `data` SoC address data [IN] `address` SoC address
Description	Write 32 bits unsigned data to the specified address.

write_u32_ad()

Usage	`void write_u32_ad(u32 address, u32 data)`
Include	**driver/io.h**
Parameters	[IN] `address` SoC address [IN] `data` SoC address data
Description	Write 32 bits unsigned data to the specified address.

Machine Timer API Calls

machineTimer_setCmp()

Usage	`void machineTimer_setCmp(u32 p, u64 cmp)`
Include	**driver/machineTimer.h**
Parameters	[IN] `p` machine timer interrupt [IN] `cmp` machine timer compare register
Description	Set a timer value to trigger an interrupt.

machineTimer_getTime()

Usage	`u64 machineTimer_getTime(u32 p)`
Include	**driver/io.h**
Parameters	[IN] `p` machine timer interrupt
Returns	[OUT] timer value
Description	Gets the timer value.

machineTimer_uDelay()

Usage	`u64 machineTimer_uDelay(u32 usec, u32 hz, u32 reg)`
Include	**driver/io.h**
Parameters	[IN] `usec` microseconds [IN] `hz` core frequency [IN] `reg` machine timer interrupt
Description	Use the machine timer to make a delay.

PLIC API Calls

plic_set_priority()

Usage	`void plic_set_priority(u32 plic, u32 gateway, u32 priority)`
Include	**driver/io.h**
Parameters	[IN] `plic` PLIC register structure [IN] `gateway` interrupt type [IN] `priority` interrupt priority
Description	Set the interrupt priority.

plic_set_enable()

Usage	`void plic_set_enable(u32 plic, u32 target, u32 gateway, u32 enable)`
Include	**driver/io.h**
Parameters	[IN] `plic` PLIC register structure [IN] `target` HART number [IN] `gateway` interrupt type [IN] `enable`
Description	Set the interrupt enable.

plic_set_threshold()

Usage	`void plic_set_threshold(u32 plic, u32 target, u32 threshold)`
Include	**driver/io.h**
Parameters	[IN] `plic` PLIC register structure [IN] `target` HART number [IN] `threshold` enable = 1
Description	Masks individual interrupt sources for the HART.

plic_claim()

Usage	`u32 plic_claim(u32 plic, u32 target)`
Include	**driver/io.h**
Parameters	[IN] `plic` PLIC register structure [IN] `target` HART number
Description	Claim the PLIC interrupt

plic_release()

Usage	`void plic_release(u32 plic, u32 target, u32 gateway)`
Include	**driver/io.h**
Parameters	[IN] `plic` PLIC register structure [IN] `target` HART number [IN] `gateway` interrupt type
Description	Release the PLIC interrupt.

SPI API Calls

spi_applyConfig()

Usage	`void spi_applyConfig(Spi_Reg *reg, Spi_Config *config)`
Include	**driver/spi.h**
Parameters	[IN] `reg` struct of the SPI register [IN] `config` struct of the SPI configuration
Description	Applies the SPI configuration to to a register for initial configuration.

spi_cmdAvailability()

Usage	`spi_cmdAvailability(Spi_Reg *reg)`
Include	**driver/spi.h**
Parameters	[IN] `reg` struct of the SPI register
Description	Read the SPI command buffer.

spi_diselect()

Usage	`void spi_select(Spi_Reg *reg, uint32_t slaveId)`
Include	**driver/spi.h**
Parameters	[IN] `reg` struct of the SPI register [IN] `slaveId` ID for the slave
Description	De-asserts the SPI select (SS) pin.

spi_read()

Usage	`uint8_t spi_write(Spi_Reg *reg)`
Include	**driver/spi.h**
Parameters	[IN] `reg` struct of the SPI register
Returns	[OUT] `reg` One byte of data
Description	Receives one byte from the SPI slave.

spi_rspOccupancy()

Usage	`spi_rspOccupancy(Spi_Reg *reg)`
Include	**driver/spi.h**
Parameters	[IN] `reg` struct of the SPI register
Description	Read the occupancy buffer.

spi_select()

Usage	`void spi_select(Spi_Reg *reg, uint32_t slaveId)`
Include	**driver/spi.h**
Parameters	[IN] `reg` struct of the SPI register [IN] `slaveId` ID for the slave
Description	Asserts the SPI select (SS) pin.

spi_write()

Usage	`void spi_write(Spi_Reg *reg, uint8_t data)`
Include	**driver/spi.h**
Parameters	[IN] `reg` struct of the SPI register [IN] `data` 8 bits of data to send out
Description	Transfers one byte to the SPI slave.

SPI Flash Memory API Calls

spiFlash_f2m_()

Usage	`void spiFlash_f2m_(Spi_Reg * spi, u32 flashAddress, u32 memoryAddress, u32 size)`
Include	**driver/spiFlash.h**
Parameters	[IN] `spi` reg struct of the SPI register [IN] `flashAddress` flash device address [IN] `memoryAddress` memory address [IN] `size` programming address size
Description	Copy data from the flash device to memory.

spiFlash_f2m()

Usage	`void spiFlash_f2m(Spi_Reg * spi, u32 cs, u32 flashAddress, u32 memoryAddress, u32 size)`
Include	**driver/spiFlash.h**
Parameters	[IN] `spi` reg struct of the SPI register [IN] `cs` chip select [IN] `flashAddress` flash device address [IN] `memoryAddress` memory address
Description	Copy data from the flash device to memory with chip select control.

spiFlash_f2m_withGpioCs()

Usage	`void spiFlash_f2m_withGpioCs(Spi_Reg * spi, Gpio_Reg *gpio, u32 cs, u32 flashAddress, u32 memoryAddress, u32 size)`
Include	**driver/spiFlash.h**
Parameters	[IN] `spi` reg struct of the SPI register [IN] `gpio` reg struct of the GPIO register [IN] `cs` chip select [IN] `flashAddress` flash device address [IN] `memoryAddress` memory address [IN] `size` programming address size
Description	Flash device from the SPI master with GPIO chip select.

spiFlash_diselect()

Usage	`void spiFlash_diselect(Spi_Reg *spi, u32 cs)`
Include	**driver/spiFlash.h**
Parameters	[IN] `spi` reg struct of the SPI register [IN] `cs` chip select
Description	De-asserts the SPI flash device from the master chip select.

spiFlash_diselect_withGpioCs()

Usage	`void spiFlash_diselect_withGpioCs(Gpio_Reg *gpio, u32 cs)`
Include	**driver/spiFlash.h**
Parameters	[IN] `gpio` reg struct of the GPIO register [IN] `cs` chip select
Description	De-asserts the SPI flash device from the master with the GPIO chip select.

spiFlash_init_()

Usage	`void spiFlash_init_(Spi_Reg * spi)`
Include	**driver/spiFlash.h**
Parameters	[IN] `spi` reg struct of the SPI register
Description	Initialize the SPI reg struct.

spiFlash_init()

Usage	`void spiFlash_init(Spi_Reg * spi, u32 cs)`
Include	**driver/spiFlash.h**
Parameters	[IN] `spi` reg struct of the SPI register [IN] `cs` chip select
Description	Initialize the SPI reg struct with chip select de-asserted.

spiFlash_init_withGpioCs()

Usage	`void spiFlash_init_withGpioCs(Spi_Reg * spi, Gpio_Reg *gpio, u32 cs)`
Include	**driver/spiFlash.h**
Parameters	[IN] `spi` reg struct of the SPI register [IN] `gpio` reg struct of the GPIO register [IN] `cs` chip select
Description	Initialize the SPI reg struct with GPIO chip select de-asserted.

spiFlash_select()

Usage	`void spiFlash_select(Spi_Reg *spi, u32 cs)`
Include	**driver/spiFlash.h**
Parameters	[IN] `spi` reg struct of the SPI register [IN] `cs` chip select
Description	Select the SPI flash device.

spiFlash_select_withGpioCs()

Usage	`spiFlash_select_withGpioCs(Gpio_Reg *gpio, u32 cs)`
Include	**driver/spiFlash.h**
Parameters	[IN] `gpio` reg struct of the GPIO register [IN] `cs` chip select
Description	Select the SPI flash device with the GPIO chip select.

spiFlash_wake_()

Usage	`void spiFlash_wake_(Spi_Reg * spi)`
Include	**driver/spiFlash.h**
Parameters	[IN] `spi` reg struct of the SPI register
Description	Release power down from the SPI master.

spiFlash_wake()

Usage	`void spiFlash_wake(Spi_Reg * spi, u32 cs)`
Include	**driver/spiFlash.h**
Parameters	[IN] `spi` reg struct of the SPI register [IN] `cs` chip select
Description	Release power down from the SPI master with chip select.

spiFlash_wake_withGpioCs()

Usage	`void spiFlash_wake_withGpioCs(Spi_Reg * spi, Gpio_Reg *gpio, u32 cs)`
Include	**driver/spiFlash.h**
Parameters	[IN] `spi` reg struct of the SPI register [IN] `gpio` reg struct of the GPIO register [IN] `cs` chip select
Description	Release power down from the SPI master with the GPIO chip select.

UART API Calls

uart_applyConfig()

Usage	`char uart_applyConfig(Uart_Reg *reg, Uart_Config *config)`
Include	**driver/uart.h**
Parameters	[IN] `reg` struct of the UART register [IN] `config` struct of the UART configuration
Description	Applies the UART configuration to to a register for initial configuration.

uart_read()

Usage	`char uart_read(Uart_Reg *reg)`
Include	**driver/uart.h**
Parameters	[IN] `reg` struct of the UART register
Returns	[OUT] `reg` character that is read
Description	Reads a character from the UART slave.

uart_readOccupancy()

Usage	`uint32_t uart_readOccupancy(Uart_Reg *reg)`
Include	**driver/uart.h**
Parameters	[IN] `reg` struct of the UART register
Description	Read the number of bytes in the RX FIFO.

uart_write()

Usage	`void uart_write(Uart_Reg *reg, char data)`
Include	**driver/uart.h**
Parameters	[IN] `reg` struct of the UART register [IN] `data` write a character
Description	Write a character to the UART.

uart_writeStr()

Usage	`void uart_writeStr(Uart_Reg *reg, char* str)`
Include	**driver/uart.h**
Parameters	[IN] `reg` struct of the UART register [IN] `str` string to write
Description	Write a string to the UART TX.

uart_writeAvailability()

Usage	`uart_writeAvailability(Uart_Reg *reg)`
Include	**driver/uart.h**
Parameters	[IN] `reg` struct of the UART register
Description	UART read/write FIFO.

Revision History

Table 6: Revision History

Date	Version	Description
December 2020	1.1	Updated to support 1280 x 720 display resolution. Added description for evsoc_ispExample_demo, evsoc_ispExample_demo2, and evsoc_ispExample_timestamp examples. Updated the steps for copying a user binary to flash.
November 2020	1.0	Initial release.