Ruby RISC-V SoC Hardware and Software User Guide

UG-RISCV-RUBY-v2.3
September 2021
www.elitestek.com

Contents

Introduction ..... **iv**
- VexRiscv RISC-V Core iv
- Required Software iv
- Required Hardware v

Chapter 1: Install Software and SoC **6**
- Install the Efinity® Software 6
- Install the RISC-V SDK 6
- Install the Java JRE 7

Chapter 2: IP Manager **8**
- Customizing the Ruby SoC 10
- Modify Bootloader On-Chip RAM Size 11

Chapter 3: Program the Board with the Ruby RTL Design **12**
- About the Example Design 12
- Enable the On-Board 10 MHz Oscillator 13
- Installing USB Drivers 13
- Program the Development Board 14

Chapter 4: Simulate **15**

Chapter 5: Launch Eclipse **16**
- Set Global Environment Variables 16

Chapter 6: Create and Build a Software Project **18**
- Create a New Project 18
- Import Project Settings (Optional) 18
- Enable Debugging 19
- Build 20

Chapter 7: Debug with the OpenOCD Debugger **21**
- Import the Debug Configuration 21
- Debug 22

Chapter 8: Create Your Own RTL Design **24**
- Target another FPGA 24
- Update the FTDI Driver for another 易灵思 Board 24
- Target Your Own Board 25
- Create a Custom AXI4 Slave Peripheral 26
- Create a Custom APB3 Peripheral 26
- Use another DDR DRAM Module 26
- Use the I^2C Interface for a Peripheral instead of DDR Calibration 26
- Remove Unused Peripherals from the RTL Design 27

Chapter 9: Create Your Own Software **28**
- Deploying an Application Binary 28
  - Boot from a Flash Device 28
  - Boot from the OpenOCD Debugger 29
  - Copy a User Binary to the Flash Device 29
- About the Board Specific Package 30
- Address Map 31
- Example Software 32
  - blinkAndEcho Example 33
  - dhrystone Example 33
  - EfxAxi4Example Design 33
  - EfxApb3Example 33
  - FreeRTOS Examples 34
  - freertosUartInterruptDemo Example 34

i2cDemo Example....35
i2cSlaveDemo Design....35
memTest Example....35
readFlash Example....35
spiDemo Example....35
timerAndGpioInterruptDemo Example....36
UartInterruptDemo Example....36
userInterruptDemo Example....36
writeFlash Example....37

Chapter 10: Using a UART Module....38
Set Up a USB-to-UART Module (Trion)....38
Open a Terminal....39
Enable Telnet on Windows....39

Chapter 11: Using a Soft JTAG Core for Example Designs....41
Enable Soft JTAG Support for the Example Design....41
Modify the Interface Design....41
Connect the FTDI Cable....43

Chapter 12: Troubleshooting....44
Error 0x80010135: Path too long (Windows)....44
OpenOCD Error: timed out while waiting for target halted....44
Memory Test....45
OpenOCD error code (-1073741515)....46
OpenOCD Error: no device found....46
OpenOCD Error: failed to reset FTDI device: LIBUSB_ERROR_IO....46
OpenOCD Error: target 'fpga_spinal.cpu0' init failed....47
Eclipse Fails to Launch with Exit Code 13....47
Efinity® Debugger Crashes when using OpenOCD....47
Undefined Reference to 'cosf'....47

Chapter 13: API Reference....48
Control and Status Registers....48
GPIO API Calls....49
I^2C API Calls....51
I/O API Calls....56
Machine Timer API Calls....57
PLIC API Calls....57
SPI API Calls....58
SPI Flash Memory API Calls....59
UART API Calls....62
Handling Interrupts....63
Revision History....66

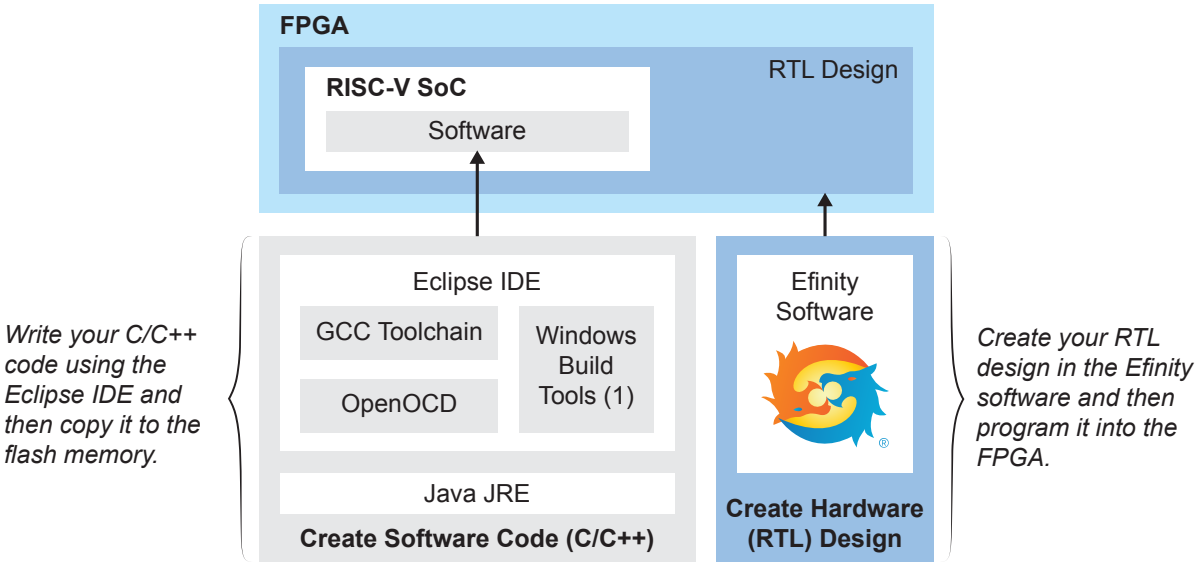
Introduction

易灵思 provides a soft RISC-V SoC, called Ruby, that you can implement in Trion® or 钛金系列 FPGAs. This user guide describes how to:

- Build RTL designs using the Ruby RISC-V SoC using an example design targeting an 易灵思® development board, and how to extend the example for your own application.
- Set up the software development environment using an example project, create your own software based on example projects, and use the API.

Note: This user guide describes how to use the Ruby SoC that is provided with the Efinity® v2020.2 or higher. Previous versions of the SoC were available as downloads in the Support Center. Although the functionality of the SoC is essentially the same, the IP Manager allows you to set parameters to customize the Ruby SoC, and the resulting directory structure is different.

Figure 1: Designing Hardware and Software for the Ruby RISC-V SoC



1. Windows build tools required on Windows platforms only.

Learn more: Refer to the Ruby RISC-V SoC Data Sheet for detailed specifications on the SoC.

VexRiscv RISC-V Core

The Ruby SoC is based on the VexRiscv core created by Charles Papon. The VexRiscv core is a 32-bit CPU using the ISA RISCV32I with M and C extensions, has five pipeline stages (fetch, decode, execute, memory, and writeback), and a configurable feature set.

In the Ruby SoC, the VexRiscv core supports the AXI4 and APB3 bus interfaces, has instruction and data caches, and can run at speeds up to 1.16 DMIPS/MHz.

The VexRiscv core won first place in the RISC-V SoftCPU contest in 2018.[1]

Required Software

[1] https://www.businesswire.com/news/home/20181206005747/en/RISC-V-SoftCPU-Contest-Winners-Demonstrate-Cutting-Edge-RISC-V

To write software for the Ruby SoC, you need the following tools. The SDK is available as a single download in the Support Center for Windows and Ubuntu operating systems.

Efinity® Software

易灵思® development environment for creating RTL designs targeting Trion® or 钛金系列 FPGAs. The software provides a complete RTL-to-bitstream flow, simple, easy to use GUI interface, and command-line scripting support.

Version: 2020.2 or higher

RISC-V SDK

Eclipse MCU—Open-source Java-based development environment that uses plug-ins to extend and customize its functionality. The GNU MCU Eclipse plug-in lets you develop applications for ARM and RISC-V cores.

Version: 2020-09 (4.17.0)

Disk space required: 433 MB (Windows), 433 MB (Linux)

xPack GNU RISC-V Embedded GCC—Open-source, prebuilt toolchain from the xPack Project.

Version: 8.3.0-1.1

Disk space required: 1.78 GB (Windows), 1.73 GB (Linux)

OpenOCD Debugger—The open-source Open On-Chip Debugger (OpenOCD) software includes configuration files for many debug adapters, chips, and boards. Many versions of OpenOCD are available. The RISC-V flow requires a custom version of OpenOCD that includes the VexRiscv 32-bit RISC-V processor.

Version: 20200421

Disk space required: 9.4 MB (Windows), 7.4 MB (Linux)

GNU MCU Eclipse Windows Build Tool (Windows Only)—This open-source Windows-specific package helps to manage build projects and includes GNU make.

Version: 2.12-20190422-1053

Disk space required: 3.8 MB

Java JRE

Open-source Java 64-bit runtime environment; required for Eclipse.

Version: 8 Update 241

https://www.java.com/en/download/manual.jsp (Java 8 official release)

https://developers.redhat.com/products/openjdk/download (OpenJDK 8 or 11)

http://jdk.java.net/16/ (OpenJDK 16)

Required Hardware

- Trion® T120 BGA324 Development Board
- 5 or 12 V power cable
- Micro-USB cable
- Computer or laptop
- (Optional) USB to UART converter module
- (Optional) FTDI chip cable, C232HM-DDHSL-0, if you want to use the OpenOCD debugger and Efinity® Debugger simultaneously

Note: Some of the software examples provided with the SoC use a UART terminal to display messages. See **Set Up a USB-to-UART Module (Trion)** on page 38 for more information.

Chapter 1

Install Software and SoC

Contents:

- Install the Efinity Software
- Install the RISC-V SDK
- Install the Java JRE

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 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 or JDK for your operating system from
   https://www.java.com/en/download/manual.jsp (Java 8 official release)
   https://developers.redhat.com/products/openjdk/download (OpenJDK 8 or 11)
   http://jdk.java.net/16/ (OpenJDK 16)
2. Follow the installation instructions on the 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

IP Manager

Contents:

- Customizing the Ruby SoC
- Modify Bootloader On-Chip RAM Size

Starting with v2020.2, the Ruby SoC is delivered with the Efinity® software; you do not need to download and install it separately. You use the IP Manager to configure the Ruby SoC and add it to your project.

The Efinity® IP Manager is an interactive wizard that helps you customize and generate 易灵思® IP cores. The IP Manager performs validation checks on the parameters you set to ensure that your selections are valid. When you generate the IP core, you can optionally generate an example design targeting an 易灵思 development board and/or a testbench. This wizard is helpful in situations in which you use several IP cores, multiple instances of an IP core with different parameters, or the same IP core for different projects.

The IP Manager consists of:

- IP Catalog—Provides a catalog of IP cores you can select. Open the IP Catalog using the toolbar button or using **Tools** > **Open IP Catalog**.
- IP Configuration—Wizard to customize IP core parameters, select IP core deliverables, review the IP core settings, and generate the custom variation.
- IP Editor—Helps you manage IP, add IP, and import IP into your project.

Generating Ruby SoC with the IP Manager

The following steps explain how to customize an IP core with the IP Configuration wizard.

1. Open the IP Catalog.
2. Choose an IP core and click **Next**. The **IP Configuration** wizard opens.
3. Enter the module name in the **Module Name** box.

Note: You cannot generate the core without a module name.

4. Customize the IP core using the options shown in the wizard. For detailed information on the options, refer to the IP core's user guide or on-line help.
5. (Optional) In the **Deliverables** tab, specify whether to generate an IP core example design targeting an 易灵思® development board and/or testbench. For SoCs, you can also optionally generate embedded software example code. These options are turned on by default.
6. (Optional) In the **Summary** tab, review your selections.
7. Click **Generate** to generate the IP core and other selected deliverables.
8. In the **Review configuration generation** dialog box, click **Generate**. The Console in the **Summary** tab shows the generation status.

Note: You can disable the **Review configuration generation** dialog box by turning off the **Show Confirmation Box** option in the wizard.

9. When generation finishes, the wizard displays the **Generation Success** dialog box. Click **OK** to close the wizard.

The wizard adds the IP to your project and displays it under **IP** in the Project pane.

Generated RTL Files

The IP Manager generates these files and directories:

- **<module name>_define.vh**—Contains the customized parameters.
- **<module name>_tmpl.v**—Verilog HDL instantiation template.
- **<module name>_tmpl.vhd**—VHDL instantiation template.
- **<module name>.v**—IP source code.
- **settings.json**—Configuration file.
- **<kit name>_devkit**—Has generated RTL, example design, and Efinity® project targeting a specific development board.
- **Testbench**—Contains generated RTL and testbench files.

Note: Refer to the IP Manager chapter of the Efinity Software User Guide for more information about the Efinity IP Manager.

Generated Software Code

If you choose to output embedded software, the IP Manager saves it into the <project>**/embedded_sw/**<SoC module> directory.

- **bsp**—Board specific package.
- **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.
- **software**—Software examples.
- **tool**—Helper scripts.
- **cpu0.yaml**—CPU file for debugging.

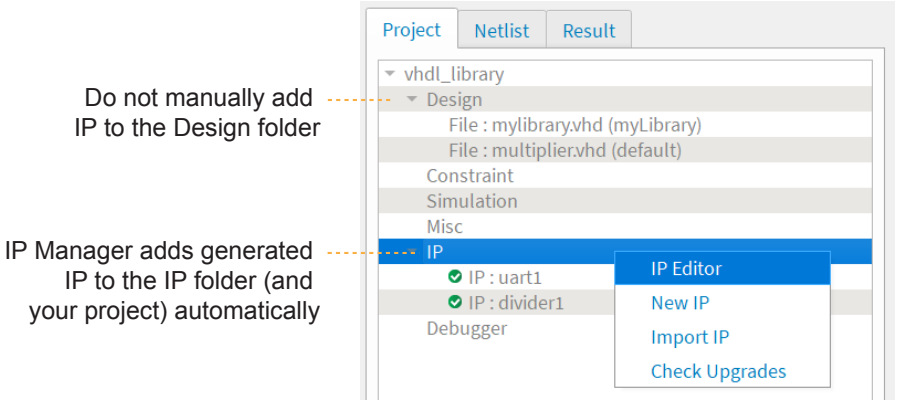
Instantiating the SoC

The IP Manager creates these template files in the <project>**/ip/**<module name> directory:

- <module name>**.v_tmpl.v** is the Verilog HDL module.
- <module name>**.v_tmpl.vhd** is the VHDL component declaration and instantiation template.

To use the IP, copy and paste the code from the template file into your design and update the signal names to instantiate the IP.

Important: When you generate the IP, the software automatically adds the module file (<module name>**.v**) to your project and lists it in the **IP** folder in the Project pane. Do not add the <module name>**.v** file manually (for example, by adding it using the Project Editor); otherwise the Efinity® software will issue errors during compilation.



Customizing the Ruby SoC

The core has parameters so you can customize its function. You set the parameters in the General tab of core IP Configuration window.

Table 1: Ruby SoC Parameters

Parameter	Options	Description
SoC Operating Frequency (Hz)	20000000 - 350000000	Enter the frequency in Hz. For the example design, if you change the frequency, you need to manually change the PLL setting and SDC timing constraint for io_systemClk to match the new frequency. Default: 100000000
SoC On-Chip Ram Size (Bytes)	4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288	The size of the on-chip block RAM for the SoC. Default: 4096
Enable Soft JTAG TAP	True, False	Choose whether you want to include a soft debug TAP for debugging. False: Default. The SoC uses the JTAG User TAP interface block to communicate with the OpenOCD debugger. True: The SoC has a soft JTAG interface to communicate with the OpenOCD debugger. You need to use this setting for T8 BGA49 or BGA81 designs or if you want to use the soft JTAG interface instead of the JTAG User TAP. After enabling the soft JTAG TAP, you need to manually assign the pins with the Interface Designer.

Important: When running the SoC at high frequencies, 易灵思 recommends that you use the TIMING_1 place and route optimization. To set this option:

1. Open the Project Editor.
2. Click the **Place and Route** tab.
3. Double-click the **Value** cell for **--optimization_level**.
4. Choose **TIMING_1**.
5. Click **OK** and then compile.

Modify Bootloader On-Chip RAM Size

When you generate the Ruby SoC, the IP Manager creates a pre-built bootloader **.bin** to transfer 124 KB of data from the SPI flash to the external memory. If you want to create a custom bootloader, use the following instructions.

Note: You need the embedded software example code to make these changes; if you have not already done so, generate it.

Modify the Bootloader Software

First you need to modify the bootloader code:

1. Open the **bootloaderConfig** file in the **embedded_sw/**<SoC module>**/bsp/efinix/EfxRubySoc/app** directory.
2. Change the `#define USER_SOFTWARE_SIZE` parameter for the new on-chip RAM size and save.
3. In Eclipse, create a new project from the makefile in the **embedded_sw/**<SoC module>**/software/standalone/bootloader** directory and compile it.

Re-Generate the Memory Initialization Files

Next, you need to re-generate the memory initialization files using the **binGen.py** helper script. You find this script in the <project>**/embedded_sw/**<SoC module>**/tool** directory. Use the command:

```
python binGen.py -b bootloader.bin -c rubysoc -t <TAP> -s <RAM size>
```

where:

- <TAP> is hard or soft, depending on whether you are using the soft JTAG TAP.
- <RAM size> is the on-chip RAM size you want to use.

This command generates the new memory initialization files. Copy these files into the same directory as your project **.xml** file, replacing the existing files.

Compile your design.

Chapter 3

Program the Board with the Ruby RTL Design

Contents:

- About the Example Design
- Enable the On-Board 10 MHz Oscillator
- Installing USB Drivers
- Program the Development Board

Before working with software code, recommends that you program your board with an RTL design that instantiates the Ruby SoC. When you generate the Ruby SoC with the IP Manager, you can optionally generate an example Efinity® project and bitstream file to get you started quickly.

About the Example Design

This example targets the Trion® T120 BGA324 Development Kit. You can access the RTL design files in the **T120F324_devkit** directory.

This example uses a simple dual-port RAM module to write to and read from the development board's memory module using the AXI interface:

- For the Trion® T120 BGA324 Development Board, the design uses the board's LPDDR3 DRAM module.
- For the 钛金系列 Ti60 F225 Development Board, the design uses the board's HyperRAM module.

The example software blinks an LED and displays messages on a UART terminal.

The Ruby SoC is configured for:

- 100 MHz frequency
- 4096 RAM size
- Soft TAP is false

Figure 2: Example Design Block Diagram

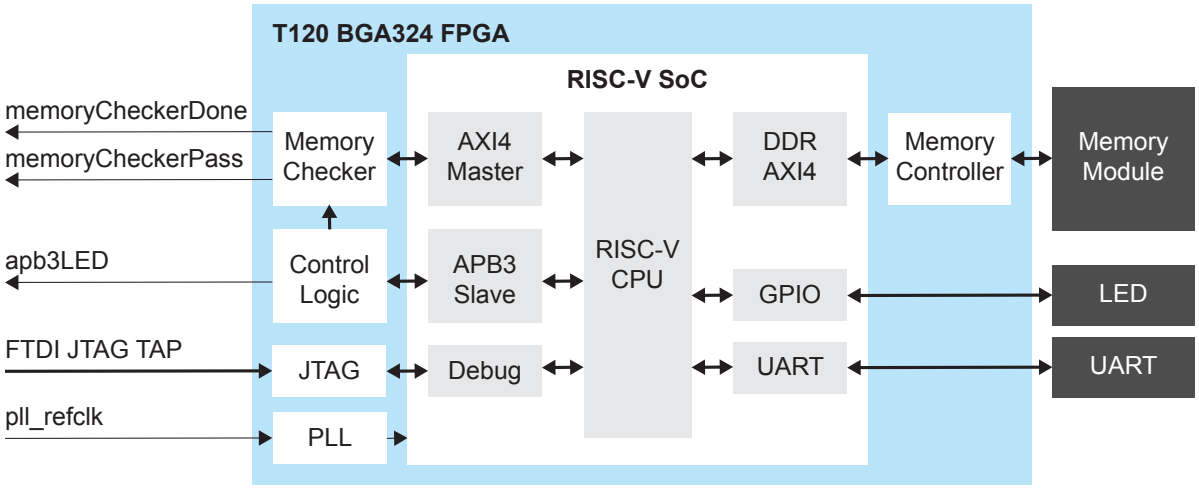


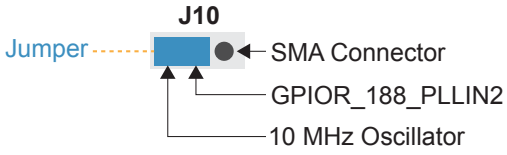
Table 2: Example Design Implementation

FPGA	Logic Utilization (LUTs)	Memory Blocks	f_{MAX} (MHz)	Language	Efinity Version
T120 BGA324 C4	12,108	78	111.794	Verilog HDL	2020.2

Enable the On-Board 10 MHz Oscillator

The SoC design uses the on-board 10 MHz oscillator. To enable it, add a jumper to connect pins 2 and 3 on header J10.

Figure 3: Connect Pins 2 and 3 on J10



Installing USB Drivers

易灵思 development boards have FTDI chips (FT232H, FT2232H, or FT4232H) to communicate with the USB port and other interfaces such as SPI, JTAG, or UART. These chips have separate channels for each interface. If you install a 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 single composite device.

- Trion® T120 BGA324 Development Kit—Install a composite driver for this board.

When working with OpenOCD on Windows, you need to install the **libusbK** driver as described in the following sections.

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 (Trion® T20 BGA256 Development Kit)

You use the Zadig software to install the driver, but instead of installing for separate interfaces, you install a composite driver.

1. Connect the board to your computer with the appropriate cable and power it up.
2. Download the Zadig software from **zadig.akeo.ie**. (You do not need to install it; simply run the downloaded executable.)
3. Run the Zadig software.

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

4. Choose **Options** > **List All Devices**.
5. Turn off **Options** > **Ignore Hubs or Composite Parents**.
6. Select the Trion® T20 BGA256 Development Kit.
7. Choose **libusbK** in the **Driver** drop-down list.
8. Click **Replace Driver**.
9. Close the Zadig software.

Installing Drivers on Linux (All Kits)

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 Development Board

When you generate the Ruby SoC in the IP Manager, you can optionally generate an example design targeting an 易灵思 development board. Example designs include a bitstream file, **soc_rubySoc.hex**, so you can get started quickly without having to compile the design.

Table 3: Available Example Designs

Board	Location
Trion® T120 BGA324 Development Board	T120F324_devkit

Download the **.hex** file to the board using these steps:

1. Connect the board to your computer using a USB cable.
2. Use the Efinity® Programmer and SPI active mode to program the bitstream file into the flash memory on the board.

   Note: You use SPI active mode because you need to reset the FPGA.

3. Press SW2 (CRESET) on the 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 and board documentation are available in the Support Center.

Chapter 4

Simulate

The Ruby SoC has a testbench so you can simulate applications in the ModelSim simulator. The simulation files are located in the **Testbench** directory. To simulate:

1. Open the **run.bat** (Window) or **run.sh** (Linux) file.
2. Change the value of the `MyApp` variable for the software binary you want to use with the simulation. If you do not specify a binary, the simulation defaults to using the blinkAndEcho binary.

   run.bat—To change or specify a software binary, uncomment the `set MyApp` line by removing the `::` and then enter the filename:

   ```
   ::set "MyApp=blinkAndEcho.bin"        :: commented line
   set "MyApp=mySoftware.bin"            :: point to user file
   ```

   run.sh—By default, `MyApp` is undefined. Specify a filename for `MyApp`:

   ```
   MyApp="mySoftware.bin"
   ```
3. (Optional) If you are not using the default, copy the application binary for the software code into the **Testbench** directory.
4. Open a Command Prompt (Windows) or terminal (Linux).
5. Change to the **Testbench** directory.
6. Execute the command `./run.bat` or `./run.sh`.

The Ruby SoC simulation uses a special bootloader configuration to speed up simulation by bypassing the SPI flash data retrieval step. Do not use this bootloader in your Efinity® project.

Note: The simulation model includes a simple memory model. If you need to simulate with the DDR controller and DRAM model, contact at support@efinixinc.com.

Note: By default, the memory initialization files contain a bootloader application that fetches 124 KB of data from the SPI flash and transfers it to the external memory. If you want build a custom bootloader, refer to Modify Bootloader On-Chip RAM Size on page 11.

Chapter 5

Launch Eclipse

Contents:

- Set Global Environment Variables

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 BSP, and launches Eclipse. Always use this executable to launch Eclipse; do not launch Eclipse directly.

When you first start working with the Ruby 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 SoC projects. recommends using the **embedded_sw/<SoC module>** directory as the workspace directory.

Note: With IP Manager, you can generate multiple SoCs with different options. Using the **embedded_sw/<SoC module>** directory as your workspace means that you can explore more than one SoC by simply switching workspaces.

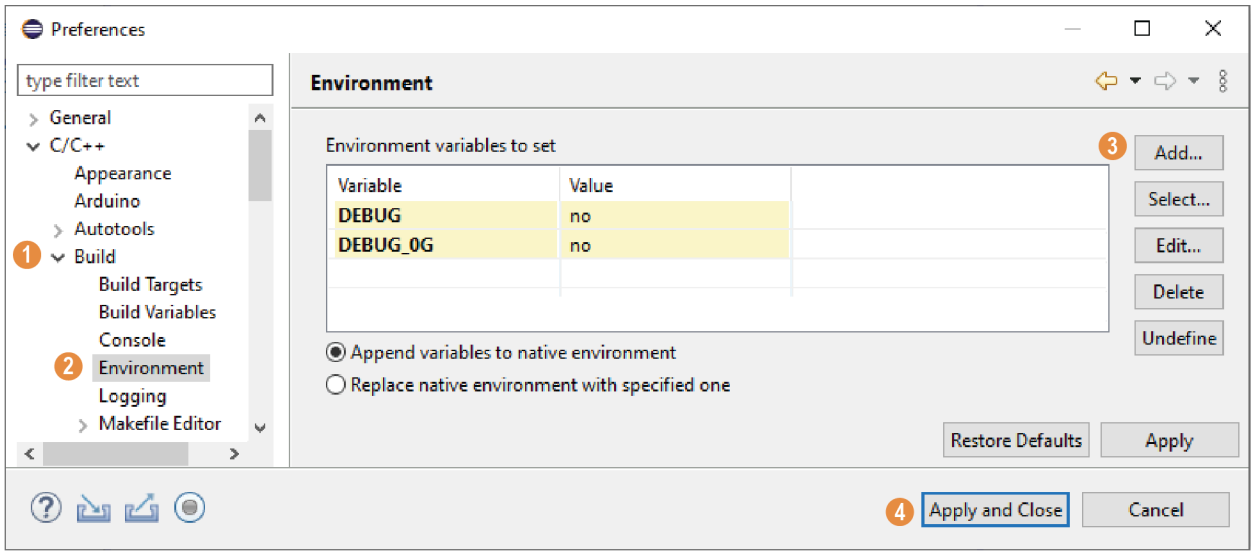
Follow these steps to launch Eclipse and set up your workspace:

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 2 for Ruby and press enter.
3. If this is the first time you are running Eclipse, create a new workspace that points to the **embedded_sw/<SoC module>** directory. Otherwise, choose **File** > **Switch Workspace** > **Other** to choose an existing workspace directory and click **Launch**.

Set Global Environment Variables

You need to set two environment variables for OpenOCD. It is simplest to set them as global environment variables for all projects in your workspace. Then, you can adjust them as needed for individual projects.

Choose **Window** > **Preferences** to open the **Preferences** window and perform the following steps.



1. In the left navigation menu, expand **C/C++** > **Build**.
2. Click **C/C++** > **Build** > **Environment**.
3. 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.

4. Click **Apply and Close**.

Chapter 6

Create and Build a Software Project

Contents:

- Create a New Project
- Import Project Settings (Optional)
- Enable Debugging
- Build

After you set up your Eclipse workspace, you are ready to create a new project and build it. These instructions walk you through the process using the **EfxAxiExample** example project from the **software** directory.

Create a New Project

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

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

Import the **EfxAxiExample** example:

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

Import Project Settings (Optional)

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.

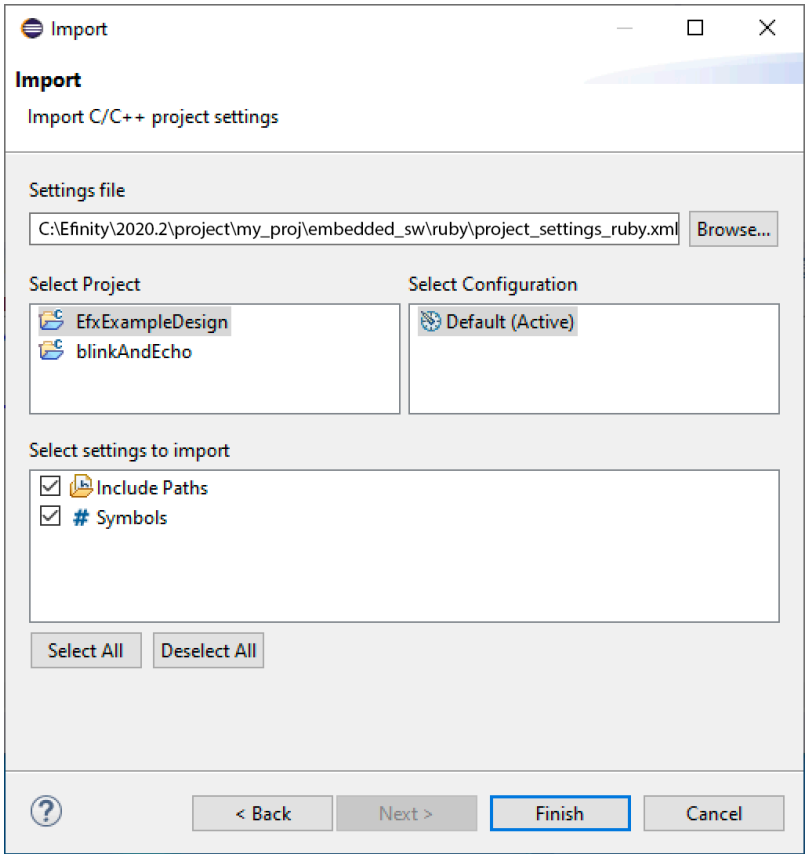
Note: You are not required to import the project settings to build. These settings simply make it easier for you to write and debug code.

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	**embedded_sw\<SoC module>\config\project_settings_ruby.xml**
Linux	**embedded_sw/<SoC module>/config_linux/project_settings_ruby_linux.xml**

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



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.

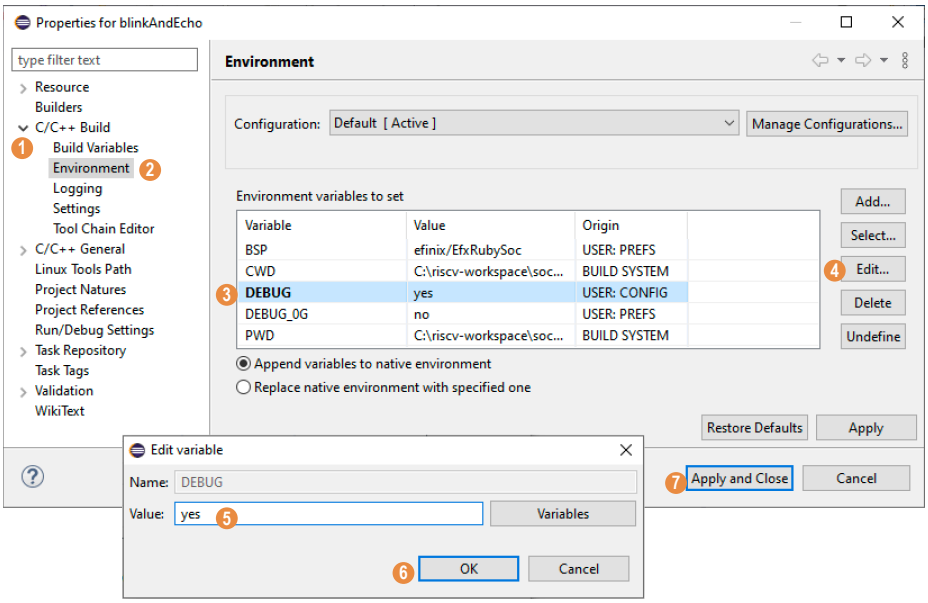
Enable Debugging

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 **EfxAxiExample** in the Project Explorer and choose **Properties** from the pop-up menu.



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:

- **EfxAxiExample.asm**—Assembly language file for the firmware.
- **EfxAxiExample.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.
- **EfxAxiExample.elf**—Use this file when debugging with the OpenOCD debugger.
- **EfxAxiExample.hex**—Hex file for the firmware. (Do not use it to program the FPGA.)
- **EfxAxiExample.map**—Contains the SoC address map.

Chapter 7

Debug with the OpenOCD Debugger

Contents:

- Import the Debug Configuration
- Debug

With the development board programmed and the software built, you are ready to configure the OpenOCD debugger and perform debugging. These instructions use the **EfxAxiExample** 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 47 for information on using two USB cables to operate both debuggers simultaneously.

Import the Debug Configuration

To simplify the debugging steps, the Ruby SoC includes debug configurations that you import. There are several configuration files, depending on what functionality you want to use.

Table 4: Debug Configurations

Debug Configuration	Use for
default	Debugging software on Trion® development boards.
default_ti	Debugging software on 钛金系列 development boards.
default_softTap	Debugging software on Trion or 钛金系列 development boards with the soft JTAG TAP interface. For example, you would need to use the soft TAP if you want to use the OpenOCD debugger and the Efinity® Debugger at the same time. (See **Using a Soft JTAG Core for Example Designs** on page 41.)

To import a debug configuration and use it to launch a debug session:

1. Connect your board to your computer using a JTAG cable. If you are using an 易灵思 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 **EfxAxiExample** project or select it under **C/C++ Projects**.
5. Right-click the **EfxAxiExample** 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	**embedded_sw\<SoC module>\config**
Linux	**embedded_sw/<SoC module>/config_linux**

9. Check the box next to **config** (Windows) or **config_linux** (Linux).
10. Click **Finish**.
11. Right-click the **EfxAxiExample** project name and choose **Debug As** > **Debug Configurations**.

12. Choose **GDB OpenOCD Debugging** > **default** (Trion FPGAs) or **default_ti** (钛金系列 FPGAs).
13. Enter `EfxAxiExample` in the **Project** box.
14. Enter `build\EfxAxiExample.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 module>** 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:\\Efinity\\2020.2\\project\\<project name>\\embedded_sw\\<SoC module>\\cpu0.yaml

16. Click **Debug**.

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

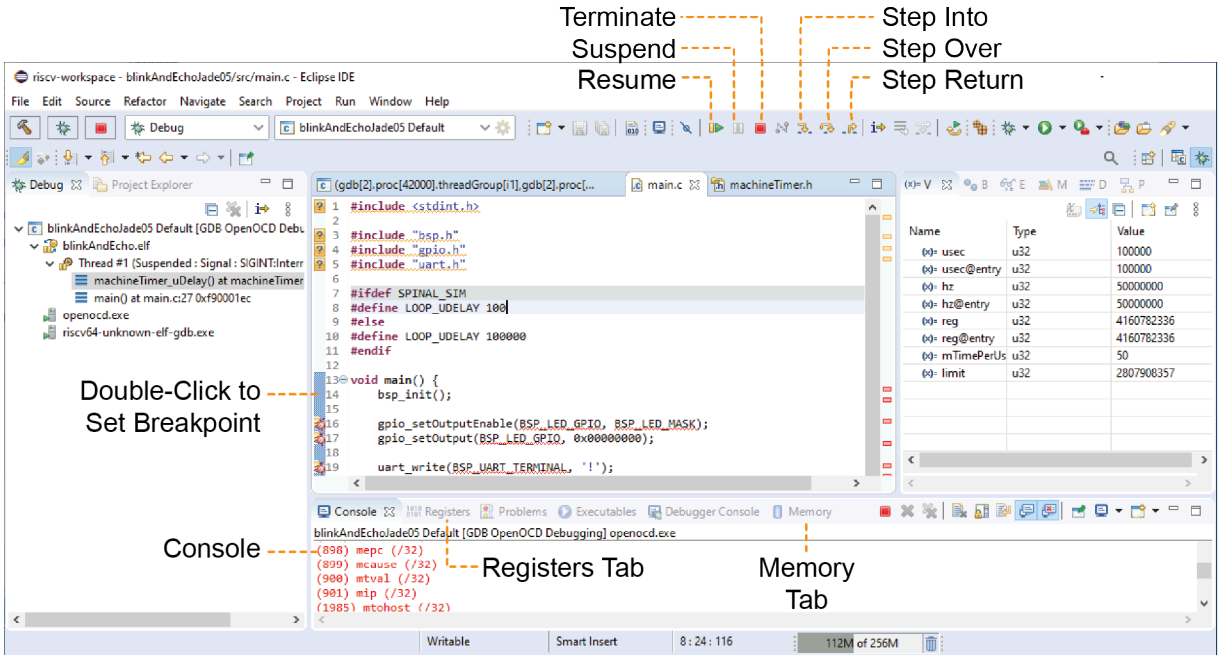
Debug

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. All of the LEDs on the board blink continuously in unison.
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 EfxAxiExample example blinks the LEDs and prints messages on a UART terminal. Refer to **Using a UART Module** on page 38 for steps on setting it up.

Figure 4: Perform Debugging



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

Chapter 8

Create Your Own RTL Design

Contents:

- Target another FPGA
- Update the FTDI Driver for another 易灵思 Board
- Target Your Own Board
- Create a Custom AXI4 Slave Peripheral
- Create a Custom APB3 Peripheral
- Use another DDR DRAM Module
- Use the I2C Interface for a Peripheral instead of DDR Calibration
- Remove Unused Peripherals from the RTL Design

After you have explored the Ruby SoC using the included example Efinity® project, you can use these tips to modify the design for your own use.

Note: recommends that you use the provided example design project as a starting point instead of creating a new project.

Target another FPGA

To change the design to target a different FPGA:

1. Edit the project to change the FPGA, package, and speed grade.
2. Update the interface design.
   a. Open the Interface Designer. The software prompts you that a device change was detected. Click **Update Design**. The Interface Designer opens and shows invalid assignments in the Message Viewer.
   b. Open the Resource Assigner.
   c. Click the instance name in the Message Viewer. The software jumps to that assignment in the Resource Assigner. Pick a new resource and press enter.
   d. Continue re-assigning pins until all assignments are valid.
   e. Generate a constraint file and close the Interface Designer.
3. Compile your modified design.

Update the FTDI Driver for another 易灵思 Board

The Ruby SoC BSP includes FTDI configuration files that specify the FTDI device VID and PID and board description for 易灵思 development boards. You can update the driver(s) to point to a different 易灵思 development board.

Table 5: Provided FTDI Configuration Files

File	Use for
ftdi.cfg	Trion® T120 BGA324 Development Board
ftdi_ti.cfg	钛金系列 Ti60 F225 Development Board

To target a different 易灵思 development board, 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 **ftdi.cfg** or **ftdi_ti.cfg** file in the **embedded_sw/<SoC module>/bsp/efinix/EfxRubySoC/openocd** directory.
5. Change the `ftdi_device_desc` setting to match the board name. For example, use this code to change the name from Trion T120F324 Development Board to Trion T120F576 Development Board:

```
interface ftdi
ftdi_device_desc "Trion T120F324 Development Board"
#ftdi_device_desc "Trion T120F576 Development Board"
ftdi_vid_pid 0x0403 0x6010
```

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

Note: All 易灵思 development boards have the same VID and PID; therefore, you only need to change the board description.

Target Your Own Board

When using an 易灵思 development board, the FTDI chip on the board bridges between the board's USB connector and the JTAG signals on the FPGA. As described in **Update the FTDI Driver for another 易灵思 Board** on page 24, the BSP has configuration files for the FTDI chip.

Additionally, the Ruby SoC also includes a configuration file for the FTDI C232HM-DDHSL-0 JTAG cable, which bridges between your computer's USB connector and the JTAG signals on the FPGA.

Note: Refer to **Connect the FTDI Cable** on page 43 for instructions on using the cable.

Generally, when debugging your own board you use a JTAG cable to connect your computer and the board. Therefore, you need to use the OpenOCD driver for that cable when debugging. OpenOCD includes a number of configuration files for standard hardware products. These files are located in the following directory:

openocd/build-win64/share/openocd/scripts/interface (Windows)

openocd/build-x86_64/share/openocd/scripts/interface (Linux)

You can also write your own configuration file if desired.

Follow these instructions when debugging with your own board:

1. Connect your JTAG cable to the board and to your computer.
2. Copy the OpenOCD configuration file for your cable to the **bsp/efinix/EfxRubySoc/openocd** directory.
3. Follow the instructions for debugging, except target your configuration file instead of the **ftdi.cfg** (Trion) or **ftdi_ti.cfg** (钛金系列) file.

```
-f <path>/bsp/efinix/EfxRubySoc/openocd/<my cable>.cfg
```

Create a Custom AXI4 Slave Peripheral

When you generate an example design for the Ruby SoC, the IP Manager creates an example AXI4 peripheral and software code that you can use as a template to create your own peripheral. This example uses the simple dual-port RAM design to write to and read from the CPU through the AXI4 interface.

- Refer to **axi4_slave.v** in the **T120F324_devkit** directory for the RTL design.
- Refer to **main.c** in the **embedded_sw/<SoC module>/software/standalone/EfxAxi4Example/src** directory for the C code.

Create a Custom APB3 Peripheral

When you generate an example design for the Ruby SoC, the IP Manager creates an APB3 peripheral and software code that you can use as a template to create your own peripheral. This simple example shows how to implement an APB3 slave wrapper.

- Refer to **apb3_slave.v** in the **T120F324_devkit** directory for the RTL design.
- Refer to **main.c** in the **embedded_sw/<SoC module>/software/standalone/EfxApb3Example/src** directory for the C code.

Use another DDR DRAM Module

The Trion® T120 BGA324 Development Board has an LPDDR3 DRAM module with 256 Mbits x 16 bits supporting up to 4 Gb. If you want to target a different module, you need to update the DDR block in the Interface Designer to reflect the specifications for your rmodule.

Note: Refer to the Trion DDR DRAM Block User Guide for more information on changing the DDR block.

Use the I^{2}C Interface for a Peripheral instead of DDR Calibration

The example design uses one of the I^{2}C interfaces (I^{2}C 2) to calibrate and reset the DDR interface. If you do not want to use calibration:

1. In the Efinity® Interface Designer, click **Disable Control** in the **Control** tab of the DDR Block Editor.
2. Remove the DDR control pins in the top-level project file (`ddr_inst1_RSTN`, `ddr_inst1_CFG_SCL_IN`, `ddr_inst1_CFG_SDA_IN`, `ddr_inst1_CFG_SDA_OEN`).

If you do not use I^{2}C 2 for calibration, you can use it for your own purposes instead.

Remove Unused Peripherals from the RTL Design

The Ruby SoC includes a variety of peripherals. if you do not want to use a peripheral, simply remove the signal name from within the parentheses () in the RubySoc RubySoc_inst definition in the top-level Verilog HDL file. For example, the SoC instantiation has these signals:

```
.system_i2c_0_io_sda_write          (system_i2c_0_io_sda_write),
.system_i2c_0_io_sda_read           (system_i2c_0_io_sda_read),
.system_i2c_0_io_scl_write          (system_i2c_0_io_scl_write),
.system_i2c_0_io_scl_read           (system_i2c_0_io_scl_read),
```

To disable I^2C 0, remove the signal name in () as shown below:

```
.system_i2c_0_io_sda_write          (),
.system_i2c_0_io_sda_read           (),
.system_i2c_0_io_scl_write          (),
.system_i2c_0_io_scl_read           (),
```

Chapter 9

Create Your Own Software

Contents:

- Deploying an Application Binary
- About the Board Specific Package
- Address Map
- Example Software

Now that you have explored the methodology for designing with the Ruby SoC, you can develop your own software applications.

Note: The Ruby SoC does not currently support floating point calculations, such as sine and cosine.

Deploying an Application Binary

During normal operation, your user binary application file (**.bin**) is stored in a SPI flash device. When the FPGA powers up, the Ruby 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 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 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 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 21 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 directory that has the **cpu0.yaml** file.
5. Type the following commands to set up the OpenOCD server:

   Windows (Trion):

   ```
   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. 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 development board.

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 6: 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.

Address Map

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

Table 7: 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 GB	–	Cache
GPIO	SYSTEM_GPIO_0_IO_APB	4K	[0]: 12 [1]: 13	I/O
I^2C 0	SYSTEM_I2C_0_IO_APB	4K	8	I/O
I^2C 1	SYSTEM_I2C_1_IO_APB	4K	9	I/O
I^2C 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
UART 2	SYSTEM_UART_2_IO_APB	4K	3	I/O
User peripheral 0	IO_APB_SLAVE_0_APB	64K	–	I/O
User peripheral 1	IO_APB_SLAVE_1_APB	64K	–	I/O
On-chip BRAM	SYSTEM_RAM_A_BMB	4 - 512 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.

Example Software

To help you get started writing software for the Ruby, provides a variety of example software code that performs functions such as communicating through the UART, controlling GPIO interrupts, performing Dhrystone benchmarking, etc. Each example includes a **makefile** and **src** directory that contains the source code.

Note: Many of these examples display messages on a UART. Refer to the following topics for information on attaching a UART module and connecting to it in a terminal:

Learn how to attach a UART module.

Learn how to open an Eclipse terminal and connect to the UART module.

Table 8: Example Software Code

Directory	Description
axiInterruptDemo	Shows how to use the AXI bus interrupt pin to trigger a software interrupt.
blinkAndEcho	This example blinks an LED and prints a string on the UART terminal.
bootloader	This software is the bootloader for the system.
common	Provides linking for the makefiles.
dhrystone	This example is a synthetic computing benchmark program.
driver	This directory contains the system drivers for the peripherals (I^2C, UART, SPI, etc.). Refer to **API Reference** on page 48 for details.
EfxApb3Example	This example shows how to implement an ABP3 slave.
EfxAxi4Example	This example illustrates how to implement a user AXI4 slave.
freertosDemo and freertosDemo2	This example uses the freeRTOS schedular to execute programs using task and queue allocation.
freertosUartInterruptDemo	This example demonstrates UART interrupts using the FreeRTOS software framework.
i2cDemo	This example shows how to connect to an MCP4725 digital-to-analog converter (DAC) using an I^2C peripheral.
i2cSlaveDemo	This example illustrates how an I^2C slave communicates with the master.
memTest	This code performs a memory address and data test.
readFlash	This example shows how to read from a SPI flash device.
spiDemo	This code reads the device ID and JEDEC ID of a SPI flash device and echoes the characters on a UART.
timerAndGpioInterruptDemo	This example shows how to use use interrupts with a timer and GPIO.
userInterruptDemo	This example demonstrates user interrupts with UART messages.
uartInterruptDemo	This exmple shows how to use a UART interrupt.
writeFlash	This example shows how to write to a SPI flash device.

blinkAndEcho Example

The blink and echo example (**blinkAndEcho** directory) is a simple example that shows how to use a register pointer to output data for the GPIO and UART. The design blinks LED D10. When you type a character, it echoes it on a UART terminal.

dhrystone Example

The Dhrystone example (**dhrystone** directory) is a classic benchmark for testing CPU performance. When you run this application, it performs dhrystone benchmark testing and displays messages and results on a UART terminal.

This benchmark includes accessing the DDR DRAM module on the Trion® T120 BGA324 Development Board.

The following code shows example results:

```
Dhrystone Benchmark, Version C, Version 2.2
   Program compiled without 'register' attribute
   Using time(), HZ=12000000
   Trying 500 runs through Dhrystone:
   Final values of the variables used in the benchmark:
   Int_Glob:    5
   should be:   5
   Bool_Glob:   1
   should be:   1
....
   Enum_Loc:    1
   should be:   1
   Str_1_Loc:   DHRYSTONE PROGRAM, 1'ST STRING
   should be:   DHRYSTONE PROGRAM, 1'ST STRING
   Str_2_Loc:   DHRYSTONE PROGRAM, 2'ND STRING
   should be:   DHRYSTONE PROGRAM, 2'ND STRING

   Microseconds for one run through Dhrystone: 40
   Dhrystones per Second:                      24472
   User_Time : 245176
   Number_Of_Runs : 500
   HZ : 12000000
   DMIPS per Mhz:                              1.16
```

EfxAxi4Example Design

This example performs a write and read test for the internal BRAM using the AXI interface. First the software writes to the internal BRAM. Then it reads back the data and compares it to the expected value. When the test completes, the application blinks LED D10 and displays the data that is read back on a UART terminal:

```
AXI4 Slave Example
00000000
00000004
00000008
0000000C
00000010
00000014
00000018
0000001C
00000020
```

EfxApb3Example

This simple software design illustrates how to use an APB3 slave peripheral.

When you run the application, it blinks LED D10, turns on LEDs D7, D9, and D9, and displays the message `Enabling Memory Checker` on the UART terminal.

FreeRTOS Examples

The Ruby SoC supports the popular FreeRTOS real-time operating system, and inlcudes example software projects targeting the RTOS. For more details on using FreeRTOS, go to their web site at **https://www.freertos.org**.

Download the FreeRTOS

The freeRTOS examples require you to download FreeRTOS.

1. Download the FreeRTOS zip file from **https://www.freertos.org**.
2. Unzip the files into the **soc_Ruby/soc_Ruby_sw/software** directory.

After you have downloaded the FreeRTOS, you use the software projects in the same manner as the other example software.

freertosDemo

This example shows how the FreeRTOS schedular handles two program executions using task and queue allocation. Generally, the FreeRTOS queue is used as a thread FIFO buffer and for intertask communication. This example creates two tasks and one queue; the queue sends and receives traffic. The receive traffic (or receive queue) blocks the program execution until it receives a matching value from the send traffic (or send queue).

Tasks in the send queue sit in a loop that blocks execution for 1,000 miliseconds before sending the value `100` to the receive queue. Once the value is sent, the task loops, i.e., blocks for another 1,000 miliseconds.

When the receive queue receives the value `100`, it begins executing its task, which sends the message `Blink` to the UART peripheral and toggles an LED on the development board.

```
Hello world, this is FreeRTOS
Blink
Blink
Blink
```

freertosDemo2

This example shows how FreeRTOS schedular handles two program executions using a binary semaphore. The semaphore holds the hardware resource until one of the tasks execute, which then releases it to the next task. If the hardware resource is running a task, no other task can use that resource. In this example, two tasks use the same UART peripheral to print messages. By using a semaphore, the two tasks have alternate access to the UART peripheral.

```
Hello world, this is FreeRTOS
Inside uart task 1 loop
Inside uart task 2 loop
Inside uart task 1 loop
Inside uart task 2 loop
Inside uart task 1 loop
Inside uart task 2 loop
```

freertosUartInterruptDemo Example

This demo illustrates the same operation as the **uartInterruptDemo**, but it executes using the FreeRTOS software framework. The tasks and queues are allocated to an interrupt routine so that the FreeRTOS scheduler can control the execution with the given priority.

The application displays messages on a UART terminal:

```
Hello world
RX FIFO not empty interrupt
RX FIFO not empty interrupt
RX FIFO not empty interrupt
```

i2cDemo Example

The I^2C interrupt example (**i2cDemo** directory) provides example code for an I^2C master writing data to and reading data from an off-chip MCP4725 device with interrupt. The Microchip MCP4725 device is a single channel, 12-bit, voltage output digital-to-analog converter (DAC) with an I^2C interface.

The MCP4725 device is available on breakout boards from vendors such as Adafruit and SparkFun. You can connect the breakout board's SDA and SCL pins to a development board.

Trion® T120 BGA324 Development Board:

- SCL—GPIOT_RXP21, which is pin 3 on PMOD J12
- SDA—GPIOT_RXN21, which is pin 4 on PMOD J12

The code assumes that the I^2C block is the only master on the bus, and it sends frames in blocks. When you run it, the application connects to the MCP4725 device and increases the DAC value. It also prints the message `Start` on a UART terminal.

In this example:

- `void trap()` traps entries on exceptions and interrupt events
- `void externalInterrupt()` triggers an interrupt event

i2cSlaveDemo Design

This example illustrates how an I^2C slave communicates with the master. It uses a 16-bit address and 16-bit data register for read and write. The slave is ready to access the master after the `Init Done` message displays on the UART.

memTest Example

The memory test example (**memTest** directory) provides example code that performs a memory test on the DDR DRAM module and reports the results on a UART terminal. A successful test prints:

```
Memory test
Success
```

If the memory test fails, the application prints `Failed at address` <address>.

readFlash Example

The read flash example (**readFlash** directory) shows how to read data from the SPI flash device on the development board. The software reads 124K of data starting at address 0x380000, which is the default location of the user binary in the flash device. The application displays messages on a UART terminal:

```
Read Flash Start
Addr 00380000 : =FF
Addr 00380001 : =FF
Addr 00380002 : =FF
...
Addr 0039EFFE : =FF
Addr 0039EFFF : =FF
Read Flash End
```

spiDemo Example

The SPI example (**spiDemo** directory) provides example code for reading the device ID and JEDEC ID of the SPI flash device on the development board.

- The default base address map of the SPI flash master is 0xF801_4000.
- The default `SCK` frequency is half of the SoC system clock frequency.
- The default base address of the UART is 0xF801_0000 with a default baud rate of 115200.

The application displays the results on a UART terminal. It continues to print to the terminal until you suspend or stop the application.

```
Hello world
Device ID : 17
CMD 0x9F : EF4018
CMD 0x9F : EF4018
...
```

timerAndGpioInterruptDemo Example

The GPIO interrupt example (**timerAndGpioInterruptDemo** directory) provides example code for implementing a rising edge interrupt trigger with a GPIO pin. When an interrupt occurs, a UART terminal displays Hello world and then the timer interval. It continues to print the timer interval until you suspend or stop the application.

```
Hello world
BSP_MACHINE_TIMER 0
BSP_MACHINE_TIMER 1
...
```

In this example:

- `void trap()` traps entries on exceptions and interrupt events
- `void externalInterrupt()` triggers a GPIO interrupt event

UartInterruptDemo Example

The UartInterruptDemo example shows how to use a UART interrupt to indicate taslk completion when sending or receiving data over a UART. The UART can trigger a interrupt when data is available in the UART receiver FIFO or when the UART transmitter FIFO is empty. In this example, when you type a character in a UART terminal, the data goes to the UART receiver and fills up FIFO buffer. This action interrupts the processor and forces the processor to execute an interrupt/priority routine that allows the UART to read from the buffer and send a message back to the terminal.

The application displays messages on a UART terminal:

```
RX FIFO not empty interrupt
RX FIFO not empty interrupt
RX FIFO not empty interrupt
```

userInterruptDemo Example

The user interrupt example (**userInterruptDemo** directory) uses one bit from an APB3 slave peripheral as an interrupt signal to RISC-V processor. The main routine sets up an interrupt routine, then triggers an interrupt signal to the user interrupt port by programming bit 2 on the ABP3 slave to high.

When the RISC-V processor receives the interrupt signal, program execution jumps from the main routine to the interrupt (or priority) routine. The interrupt routine sets bit 2 low so the processor can leave the interrupt routine.

The application displays the messages on a UART terminal:

```
User Interrupt Demo, waiting for user interrupt...
Entered User Interrupt Routine
Turn off Interrupt Signal
Leaving User Interrupt Routine
```

writeFlash Example

The read flash example (**writeFlash** directory) shows how to write data to the SPI flash device on the development board. The software writes data starting at address 0x380000, which is the default location of the user binary in the flash device. The application displays address and data messages on a UART terminal:

```
Write Flash Start
WR Addr 00380000 : =00
WR Addr 00380001 : =01
WR Addr 00380002 : =02
...
WR Addr 003800FD : =FD
WR Addr 003800FE : =FE
WR Addr 003800FF : =FF
Write Flash End
```

Chapter 10

Using a UART Module

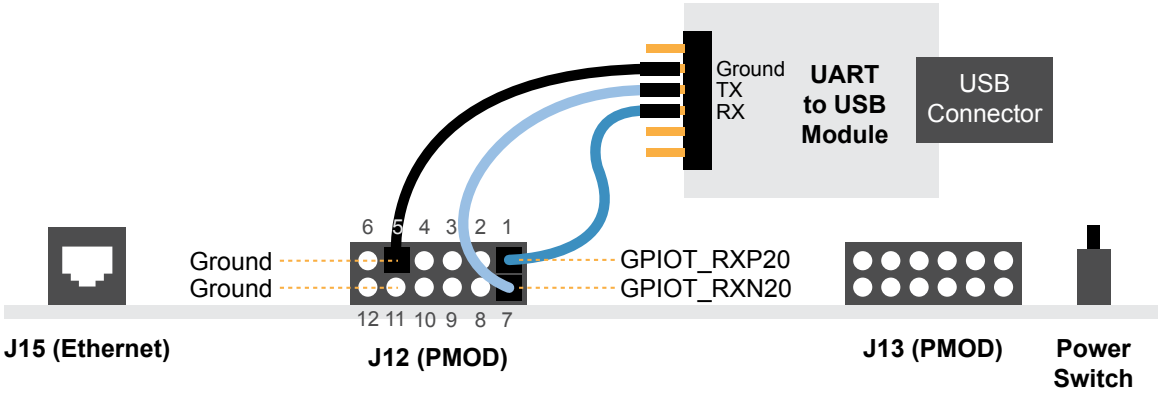
Contents:

- Set Up a USB-to-UART Module (Trion)
- Open a Terminal
- Enable Telnet on Windows

Set Up a USB-to-UART Module (Trion)

A number of the software examples display messages on a UART terminal. The Trion® T120 BGA324 Development Board does not have a USB-to-UART converter, therefore, you need to use a separate USB-to-UART converter module. A number of modules are available from various vendors; any USB-to-UART module should work.

Figure 5: Connect the UART Module to PMOD Connector J12



1. Connect the UART module to the PMOD port J12
   - RX—GPIOT_RXP20, which is pin 1 on PMOD J12
   - TX—GPIOT_RXN20, which is pin 7 on PMOD J12
   - Ground—Use ground pin 5 or 11 on PMOD J12.
2. Plug the UART module into a USB port on your computer. The driver should install automatically if needed.

Finding the COM Port (Windows)

1. Type Device Manager in the Windows search box.
2. Expand **Ports (COM & LPT)** to find out which COM port Windows assigned to the UART module; it is listed as USB Serial Port (COMn) where n is the assigned port number. Note the COM number.

Finding the COM Port (Linux)

In a terminal, type the command:

```
dmesg | grep ttyUSB
```

The terminal displays a series of messages about the attached devices.

```
usb <number>: <adapter> now attached to ttyUSB<number>
```

There are many USB-to-UART converter modules on the market. Some use an FTDI chip which displays a message similar to:

```
usb 3-3: FTDI USB Serial Device converter now attached to ttyUSB0
```

However, the Trion® T120 BGA324 Development Board also has an FTDI chip and gives the same message. So if you have both the UART module and the board attached at the same time, you may receive three messages similar to:

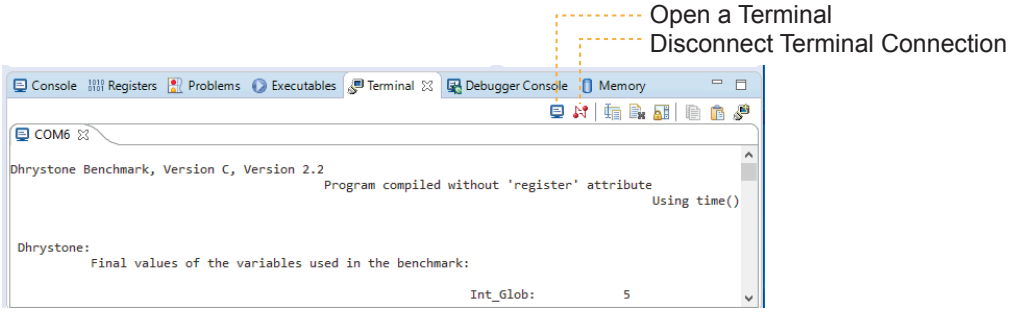
```
usb 3-3: FTDI USB Serial Device converter now attached to ttyUSB0
usb 3-2: FTDI USB Serial Device converter now attached to ttyUSB1
usb 3-2: FTDI USB Serial Device converter now attached to ttyUSB2
```

In this case the second 2 lines (marked by `usb 3-2`) are the development board and the first line (`usb 3-3`) is the UART module.

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.



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	COMn (Windows) or ttyUSBn (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.

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**.

Chapter 11

Using a Soft JTAG Core for Example Designs

Contents:

- Enable Soft JTAG Support for the Example Design
- Modify the Interface Design
- Connect the FTDI Cable

The Efinity® Debugger uses the hard JTAG TAP interface. Out of the box, the Ruby SoC example design also uses the hard JTAG TAP interface for OpenOCD. If you try to use the same USB connection to the development board for both applications at the same time, they will conflict. To solve this problem, you use a soft JTAG block to handle the OpenOCD JTAG communication. With this method, you use an FTDI chip cable to connect the board to your computer (the Efinity® Debugger uses the USB cable).

The simplest way to implement a soft JTAG interface is to use the IP Manager to output an example design that enables the soft JTAG interface.

Note: recommends you use the C232HM-DDHSL-0 FTDI chip cable rather than a JTAG mini-module because the software generated by the IP Manager includes the debug configuration file for the cable.

Enable Soft JTAG Support for the Example Design

When **Enable Soft JTAG TAP** is set to **True**, the example design top-level file includes the soft JTAG TAP signals. To enable this option:

1. Open your project.
2. Right-click the SoC module name under IP in the Project pane to open a context-sensitive menu.
3. Choose **Configure**.
4. Choose **True** for the **Enable Soft JTAG TAP** option.
5. In the **Deliverables** tab, enable one or more example designs.
6. Click **Generate** to generate the SoC with the soft JTAG TAP interface.

Modify the Interface Design

Although you turned on the soft **Enable Soft JTAG TAP** option, you also need to make some changes in the Interface Designer:

1. Open the example design project.

2. Open the Interface Designer and make these changes:
   a. Remove the JTAG User Tap block.
   b. Create these GPIO blocks for the soft JTAG pins:
      - `io_jtag_tck`—Configure as input; enable Schmitt trigger.
      - `io_jtag_tdi`—Configure as input.
      - `io_jtag_tdo`—Configure as output.
      - `io_jtag_tms`—Configure as input.

Note: Make sure that the instance names and pin names match the soft JTAG I/O ports in the `top_rubySoc` module in the `top_rubySoc`**.v** file (top-level RTL).

   c. In the Resource Assigner, assign the soft JTAG ports to the I/O pins you want to use.

Note: When using the C232HM-DDHSL-0 FTDI chip cable, make sure that the assigned I/O pins use the 3.3V I/O standard and the I/O pin you assign for `io_jtag_tck` supports Schmitt Trigger.

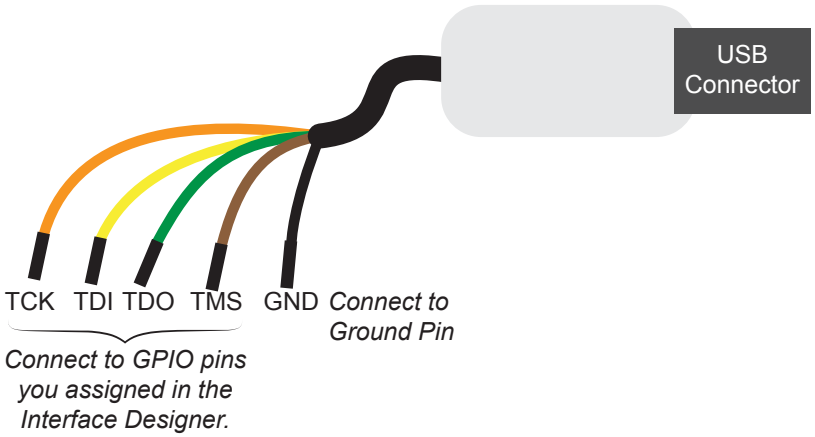
   d. Check the design to ensure that you have connected everything correctly. The Interface Designer issues warnings or error messages if it finds design issues.
   e. Save and exit the Interface Designer.
3. Add a Virtual I/O or Logic Analyzer core to your design. For instructions, refer to the Debugging chaper in the **Efinity Software User Guide**.
4. Compile the design.
5. Download the resulting bitstream file to your board using the Efinity® Programmer and a USB cable connected to the board.

Connect the FTDI Cable

When you modified your project to use the soft JTAG block, you assigned GPIO for the 4 JTAG signals, TMS, TDK, TCI, TDO. These GPIO should be assigned to header pins on the development board so you can connect the C232HM-DDHSL-0 FTDI chip cable to those pins.

1. Connect the cable to your board using the following figure as a guide.

Figure 6: Connecting the C232HM-DDHSL-0 Cable



Note: If you have not already done so, install the driver for the FTDI cable as described in **Installing USB Drivers** on page 13.

2. Open your Eclipse project.
3. Run or debug the software with the OpenOCD debugger using the **default_softTap** launch configuration.
4. Refer to **Debug with the OpenOCD Debugger** on page 21 for complete instructions.
5. Open the Debugger to perform hardware debugging.

Chapter 12

Troubleshooting

Contents:

- Error 0x80010135: Path too long (Windows)
- OpenOCD Error: timed out while waiting for target halted
- Memory Test
- OpenOCD error code (-1073741515)
- OpenOCD Error: no device found
- OpenOCD Error: failed to reset FTDI device: LIBUSB_ERROR_IO
- OpenOCD Error: target 'fpga_spinal.cpu0' init failed
- Eclipse Fails to Launch with Exit Code 13
- Efinity Debugger Crashes when using OpenOCD
- Undefined Reference to 'cosf'

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 SoC design.
- You may not have the correct PLL settings to work with the Ruby 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 2 and 3. This jumper enables the 10 MHz on-board oscillator. Refer to **Enable the On-Board 10 MHz Oscillator** on page 13.
- Check the JTAG connection.
- Ensure that the FPGA is programmed with the Ruby SoC. Refer to **Program the Development Board** on page 14.

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 directory that has the **cpu0.yaml** file.
5. Type the following commands to set up the OpenOCD server:

   Windows (Trion):

   ```
   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 0s.

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 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 易灵思 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, OpenOCD 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 **ftdi.cfg** (Trion) or **ftdi_ti.cfg** (钛金系列) file in the **/bsp/efinix/EFXRubySoC/openocd** directory.
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 have 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: recommends using the **libusbK** driver, which you install using the Zadig software. Refer to **Installing USB Drivers** on page 13

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 for Example Designs** on page 41 for details.

Undefined Reference to 'cosf'

You may receive an error similar to this when using calculating square root, sine, or cosine with floating-point numbers in your application. The Ruby SoC does not currently support floating point.

Chapter 13

API Reference

Contents:

- Control and Status Registers
- GPIO API Calls
- I2C API Calls
- I/O API Calls
- Machine Timer API Calls
- PLIC API Calls
- SPI API Calls
- SPI Flash Memory API Calls
- UART API Calls
- Handling Interrupts

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_getFilteringHit()

Usage	`gpio_getFilteringHit(reg)`
Parameters	[IN] reg struct of I^2C setting value
Include	**driver/i2c.h**
Description	Read the 32-bit I^2C register filter hit with a call back function.

Example	`if(gpio_getFilteringHit(I2C_CTRL) == 1)`
	`// Check filter hit value, bit [7] from slave address,`
	`// read =’1’ write =’0’`

gpio_getFilteringStatus()

Usage	`gpio_getFilteringStatus(reg)`
Parameters	[IN] reg struct of I^2C setting value
Include	**driver/i2c.h**
Description	Read the 32-bit I^2C register filter hit with a call back function.

Example	`if(gpio_getFilteringStatus (I2C_CTRL) == 1)`
	`// Check filter hit status, bit [7] from slave address, read =’1’`
	`write =’0`

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_getInterruptFlag()

Usage	`gpio_getInterruptFlag(reg)`
Parameters	[IN] reg struct of I^2C setting value
Include	**driver/i2c.h**
Description	Read the 32-bit I^2C register interrupt flag with a call back function.

Example	`Int flag = gpio_getInterruptFlag(I2C_CTRL) & I2C_INTERRUPT_DROP;`
	`// Get Drop interrupt flag from Interrupt register`
	`//[2] I2C_INTERRUPT_TX_DATA`
	`//[3] I2C_INTERRUPT_TX_ACK`
	`//[7] I2C_INTERRUPT_DROP`
	`//[16] I2C_INTERRUPT_CLOCK_GEN_BUSY`
	`//[17] I2C_INTERRUPT_FILTER`

gpio_getMasterStatus()

Usage	`gpio_getMasterStatus(reg)`
Parameters	[IN] reg struct of I^2C setting value
Include	**driver/i2c.h**
Description	Read the 32-bit I^2C register master status with a call back function.

Example	`int status = gpio_getMasterStatus(I2C_CTRL) & I2C_MASTER_BUSY;`
	`// Get master busy status from status register`
	`[0]I2C_MASTER_BUSY`
	`[4]I2C_MASTER_START`
	`[5]I2C_MASTER_STOP`
	`[6]I2C_MASTER_DROP`

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_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_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_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^2C API Calls

i2c_applyConfig()

Usage	`void i2c_applyConfig(u32 reg, I2c_Config *config)`
Parameters	[IN] `reg` struct of I^2C setting value [IN] `config` struct of I^2C configuration
Include	**driver/i2c.h**
Description	Apply I^2C 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: • [2] I2C_INTERRUPT_TX_DATA • [3] I2C_INTERRUPT_TX_ACK • [7] I2C_INTERRUPT_DROP • [16] I2C_INTERRUPT_CLOCK_GEN_BUSY • [17] I2C_INTERRUPT_FILTER
Include	**driver/i2c.h**
Description	Disable I^{2}C interrupt.
Example	`i2c_disableInterrupt(I2C_CTRL, I2C_INTERRUPT_TX_ACK);` `// Enable I2C interrupt  with interrupt TX Ack`

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: • [2] I2C_INTERRUPT_TX_DATA • [3] I2C_INTERRUPT_TX_ACK • [7] I2C_INTERRUPT_DROP • [16] I2C_INTERRUPT_CLOCK_GEN_BUSY • [17] I2C_INTERRUPT_FILTER
Include	**driver/i2c.h**
Description	Enable I^{2}C interrupt.

Example	`i2c_enableInterrupt(I2C_CTRL, I2C_INTERRUPT_FILTER |`
	`  I2C_INTERRUPT_DROP);`
	`// Enable I2C interrupt  with interrupt filter and drop`

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.
Example	`i2c_masterDrop(I2C_CTRL);`

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^2C 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^2C setting value [IN] `filterID` filter configuration ID number [IN] `config` struct of I^2C configuration: • [9:0] I2C slave address • [14] I2C_FILTER_10BITS • [15] I2C_FILTER_ENABLE
Include	**driver/i2c.h**
Description	Set the filter configuration.
Example	`i2c_setFilterConfig(I2C_CTRL, 0, 0x30 | I2C_FILTER_ENABLE);` `// Enable filter with ID=0 slave addr = 0x30 default 7 bit filter`

i2c_txAck()

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

i2c_txAckBlocking()

Usage	`void i2c_txAckBlocking(u32 reg)`
Parameters	[IN] `reg` struct of I^2C 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^2C 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^2C register [IN] `byte` 8 bits data to send out
Include	**driver/i2c.h**
Description	Transfers one byte to the I^2C 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_emptyInterruptEna()

Usage	`uart_emptyInterruptEna(u32 reg char ena)`
Include	**driver/uart.h**
Parameters	[IN] `reg` struct of the UART register [IN] `ena` Enable interrupt
Description	Enable the TX FIFO empty interrupt.

uart_NotemptyInterruptEna()

Usage	`uart_NotemptyInterruptEna(u32 reg char ena)`
Include	**driver/uart.h**
Parameters	[IN] `reg` struct of the UART register [IN] `ena` Enable interrupt
Description	Enable the RX FIFO not empty interrupt.

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_status_read()

Usage	`uart_status_read(u32 reg)`
Include	**driver/uart.h**
Parameters	[IN] `reg` struct of the UART register
Returns	[OUT] 32-bit status register from the UART
Description	Read the UART status.

uart_status_write()

Usage	`uart_status_write(u32 reg)`
Include	**driver/uart.h**
Parameters	[IN] `reg` struct of the UART register [IN] `data` input data for the UART status.
Returns	[OUT] 32-bit status register from the UART
Description	Write the UART status.

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.

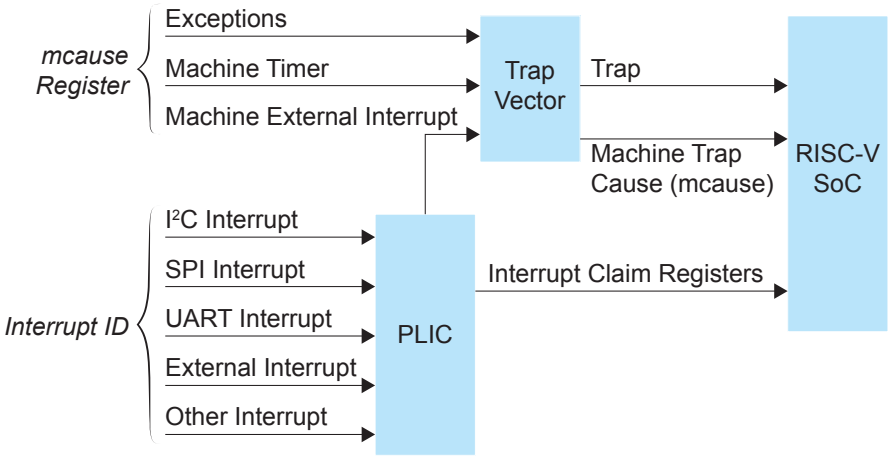
uart_writeAvailability()

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

Handling Interrupts

There are two kinds of interrupts, trap vectors and PLIC interrupts, and you handle them using different methods.

Figure 7: Types of Interrupts

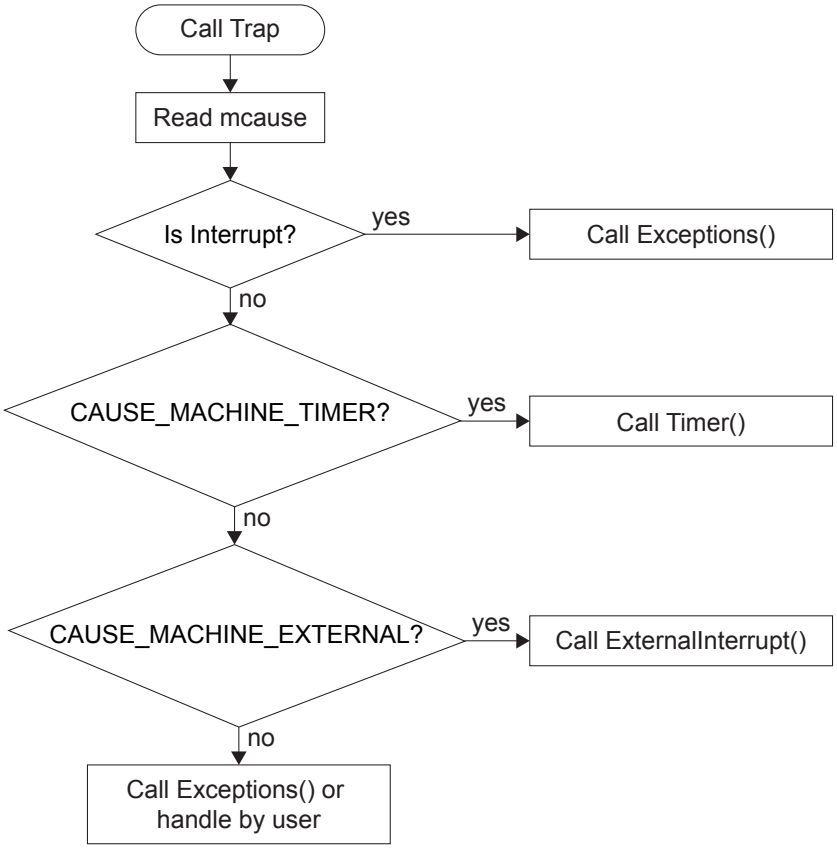


Trap Vectors

Trap vectors trap interrupts or exceptions from the system. Read the Machine Cause Register (`mcause`) to identify which type of interrupt or exception fthe system is generating. Refer to "Machine Cause Register (mcause): 0x342" in the data sheet for your SoC for a list of the exceptions and interrupts used for trap vectors. The following flow chart explains how to handle trap vectors.

For CAUSE_MACHINE_EXTERNAL, it will call the subroutine to process the PLIC level interrupts.

Figure 8: Handling Trap Vectors

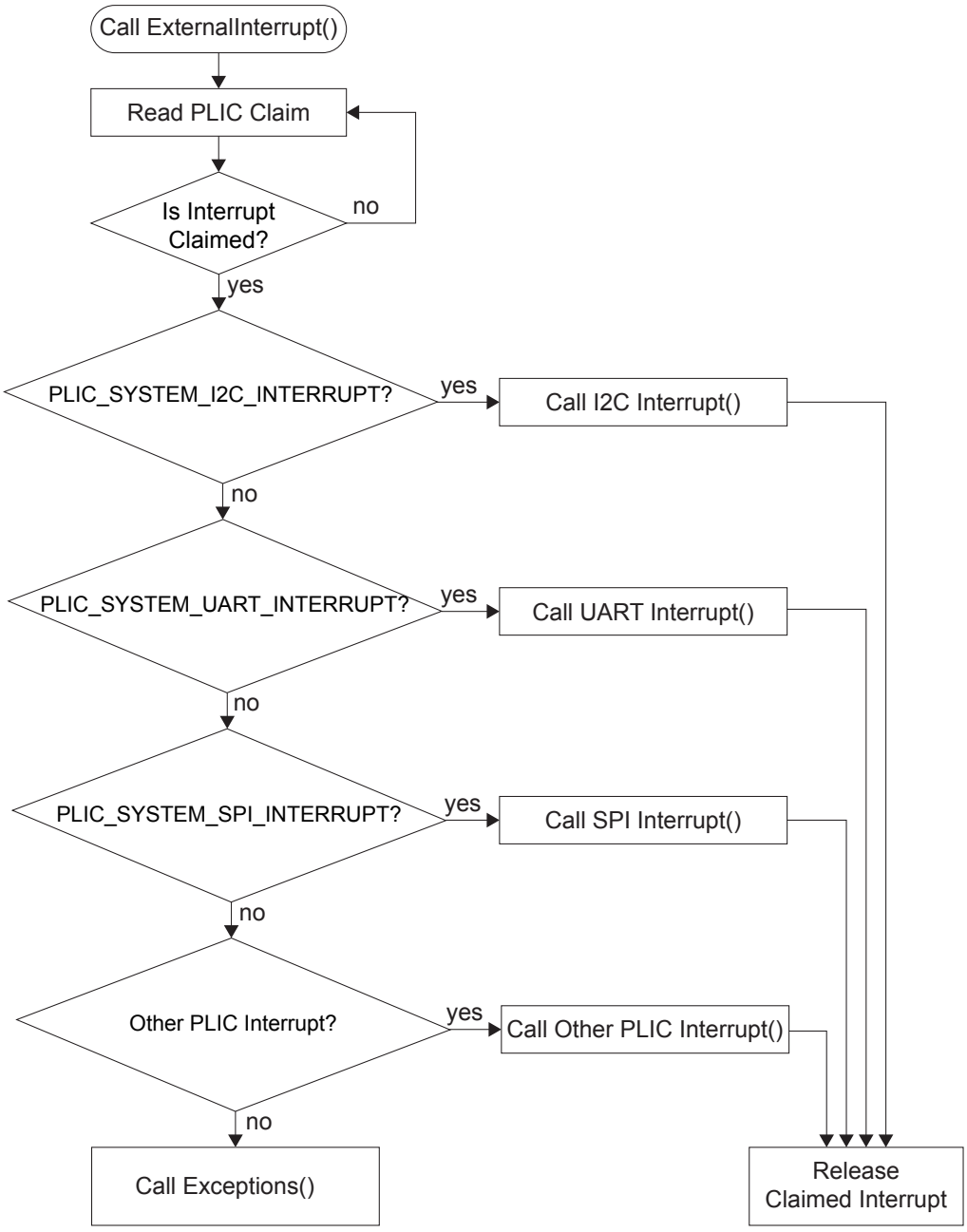


PLIC Interrupts

The PLIC collects external interrupts and is also used for CAUSE_MACHINE_EXTERNAL cases. Read the interrupt claim registers (PLIC claim) to identify the source of the external interrupt. Refer to **Address Map** on page 31for a list of the interrupt IDs.

The following flow chart shows how the PLIC handles interrupts.The PLIC identifies the interrupt ID and processes the corresponding interrupts.

Figure 9: Handling PLIC Interrupts



Revision History

Table 9: Revision History

Date	Version	Description
September 2021	2.3	The SoC Operating Frequency (Hz) minimum frequency changed to 20000000 Hz. (DOC-544)
July 2021	2.2	Added link to OpenJDK (DOC-457) Added information about the flow for handling interrupts in the API Reference chapter. (DOC-398) Updated the GPIO, I2C, and UART API calls. (DOC-454) Added additional instructions on using the Efinity and OpenOCD debuggers at the same time. (DOC-426) Added descriptions for the axiInterruptDemo, dmasg, and i2cSlaveDemo examples.
February 2021	2.1	Corrected the pin numbers for PMOD Connector J12 in Trion® T120 BGA324 Development Board.
December 2020	2.0	Updated to describe new configuration and deliverables via IP Manager in Efinity® v2020.2.
November 2020	1.2	Added uartInterruptDemo example. Added freeRTOS example.
August 2020	1.1	Updated for v1.1 of the SDK; the process for setting up Eclipse environment variables and debug configurations is streamlined. Added chapter on simulation. User peripheral address size changed to 64K. io_apbSlave_PADDR size changed to 15:0. AXI user slave peripheral address size changed to 16 MB. Added the freeRTOS example software description. Added FTDI Dual RS232 HS mini module in steps to install the USB driver. Updated instructions for installing USB drivers on Linux.
June 2020	1.0	Initial release.