UG0772
User Guide
RISC-V Soft Processor Hardware Abstraction Layer v2.0

Microsemi
Power Matters.

Microsemi Corporate Headquarters
One Enterprise, Aliso Viejo,
CA 92656 USA
Within the USA: +1 (800) 713-4113
Outside the USA: +1 (949) 380-6100
Fax: +1 (949) 215-4996
Email: sales.support@microsemi.com
www.microsemi.com

About Microsemi

Microsemi Corporation (Nasdaq: MSCC) offers a comprehensive portfolio of semiconductor and system solutions for aerospace & defense, communications, data center and industrial markets. Products include high-performance and radiation-hardened analog mixed-signal integrated circuits, FPGAs, SoCs and ASICs; power management products; timing and synchronization devices and precise time solutions, setting the world's standard for time; voice processing devices; RF solutions; discrete components; enterprise storage and communication solutions, security technologies and scalable anti-tamper products; Ethernet solutions; Power-over-Ethernet ICs and midspans; as well as custom design capabilities and services. Microsemi is headquartered in Aliso Viejo, California, and has approximately 4,800 employees globally. Learn more at www.microsemi.com.

50200772. 1.0 4/17

Contents

Figures

Tables

1 Revision History

The revision history describes the changes that were implemented in the document. The changes are listed by revision, starting with the most current publication.

1.1 Revision 1.0

This is the first revision of this document for RISC-V HAL v2.0. Version v2.0 is the first production release of RISC-V HAL.

2 Introduction

RISC-V soft processor hardware abstraction layer (RISC-V HAL) provides boot code, interrupt handling, and hardware access methods for RISC-V soft processors.

The RISC-V HAL package is a combination of C and assembly source code.

2.1 Features

RISC-V HAL provides the following features:

- RISC-V boot code for the SoftConsole toolchain
- Platform-Level Interrupt Controller (PLIC) access functions
- Power, Reset, Clock, and Interrupt (PRCI) access functions
- Hardware abstraction layer for Microsemi soft IP drivers
- Optional redirection of the standard C library output to one of the UARTs using the CoreUARTapb soft IP and driver

2.2 Supported Hardware IP

Use RISC-V HAL with CoreRISCV_AXI4 IP v2.0 onwards. CoreRISCV_AXI4 is a DirectCore.

The CoreRISCV_AXI4 IP:

- Supports the RV32IM Instruction Set Architecture (ISA)
- Provides a single hardware thread (hart)
- Provides a machine-mode privileged architecture
- Provides high-performance single-issue in-order 32-bit execution pipeline
- Provides Integrated 8 KB instructions cache and 8 KB data cache
- Supports Two external AXI interfaces for IO and memory
- Supports up to 31 programmable interrupts
- Supports debug unit with a JTAG interface

3 Files Provided

The following resources are provided as part of RISC-V HAL:

- Documentation
- Core source code (including the linker setup)
- Debug configuration files

RISC-V HAL is distributed through the Firmware Catalog of Microsemi SoC Products Group. The Firmware Catalog:

- Provides access to the documentation of RISC-V HAL
- Generates the RISC-V HAL source files into an application project
- Generates debug configuration files for the SoftConsole tool chain.

The Firmware Catalog is available at www.microsemi.com/soc/products/software/firmwarecat/default.aspx.

3.1 Documentation

The Firmware Catalog provides access to the following documents:

- User guide (this document)
- Release notes

3.2 RISC-V HAL Source Code

The Firmware Catalog generates the RISC-V HAL source code into the `riscv_hal` subdirectory of the selected software project directory. For example, `<my_project>/riscv_hal`. The following sections describe all the files that form RISC-V HAL.

3.2.1 encoding.h

This header file contains the register bit mask for the RISC-V soft processor.

3.2.2 entry.S

This assembly code file contains the first code executed after the CoreRISCV_AXI4 comes out of the reset. This file initializes the trap vector address, global pointer and stack pointer and then calls the _init() function, which completes the rest of the initializations in the C code.

3.2.3 init.c

This C source code file copies and fills all the memory that is required on startup. This file uses memory locations and sizes defined in the linker file to determine what to copy and where to copy. This file is called from the startup code in the entry.S file.

3.2.4 riscv_CoreplexE31.h

This header file contains PLIC and PRCI data structures and access functions of CoreRISCV_AXI4. All the functions in this file are defined as inline functions. These functions provide initialization, configuration, and access methods for PLIC and PRCI. This file is used to:

- Define PLIC and PRCI memory map and data structures
- Initialize and configure the PLIC
- Provide PLIC interrupt handling functions

3.2.5 riscv_hal.c

This C source code file implements the hardware abstraction layer for the RISC-V processor. This file provides functions to control the global interrupt and implements the trap handler, which determines the reason for the trap and gives the provision to call the user defined interrupt handlers. This file also provides the function to configure the internal machine mode timer in the PRCI module. Unexpected

traps other than those due to interrupts are reported as error condition along with the cause as encoded by the mcause register.

3.2.6 riscv_hal.h

This header file contains the public application programming interface (API) of the RISCV_HAL. This file should be included in all the C source files that uses RISC-V HAL.

3.2.7 riscv_hal_stubs.c

This C source code file contains the default exception handlers of CoreRISCV_AXI4 IP. If the user does not provide an implementation of these exception handlers, they will be linked to the application code. These functions are defined with weak linking so that they can be overridden by a function with same prototype in the application code.

3.2.8 syscall.c

This C source code file contains stubs for the library system calls. These system calls are tool specific. In this file, the system calls are implemented for SoftConsole.

3.2.9 microsemi-riscv-ram.ld

This GNU linker script is an example linker script for users to use with SoftConsole. This script is used as a starting point for applications that execute from the fabric SRAM.

3.2.10 sample_hw_platform.h

This header file contains the sample hardware specific information about a RISC-V soft processor Libero Design. This information is required for RISC-V HAL. The sample values must be modified based on the application-specific Libero design.

3.2.11 Soft IP Bare Metal Drivers Hardware Abstraction Layer

These files are located in the directory `<my_project>/hal`.

The Firmware Catalog generates the CoreRISCV_AXI4 Soft IP Bare Metal Hardware Abstraction Layer source code into the `hal` subdirectory of the selected software project directory. These file are used in the implementation of bare metal drivers for hardware soft IP. The application code does not need to include any of the header files in this subdirectory, because they are only used by Microsemi provided bare metal drivers.

If soft IP drivers are used, the `hal_irq.c` and `hw_reg_access.c` files must be part of the application code.

3.3 Soft IP Bare Metal Drivers

These files are located in the directory `<my_project>/driver`.

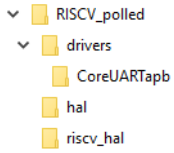
The Firmware Catalog or the Libero hardware design flow generates the DirectCore soft IP drivers into the subdirectories of this folder. All the C source code files present must be included as part of your project.

4 RISC-V HAL Deployment

RISC-V HAL is deployed from the Firmware Catalog into a software project by generating the RISC-V HAL source files into the project directory.

The following example shows the directory structure of a SoftConsole project for RISC-V soft processor. This project uses the CoreUARTapb driver. This driver relies on RISC-V HAL for accessing the hardware. The contents of the drivers directory are created by generating the source files of the drivers used in the project from the list of drivers in the Firmware Catalog. The contents of the `riscv_hal` and `hal` directories are created by generating the source files of the RISC-V HAL into the project.

Figure 1 • **RISC-V Soft Processor Project Example**



5 Hardware Setup

During the creation of a CoreRISCV_AXI4 design in Libero, the following elements are required:

- Specific memory map
- Non-volatile storage
- JTAG setup

5.1 Debug Mode Configuration

In the debug mode configuration, the CoreRISCV_AXI4 boots from the fabric SRAM. The complete memory for CoreRISCV_AXI4 is explained in the *CoreRISCV_AXI4 Handbook*.

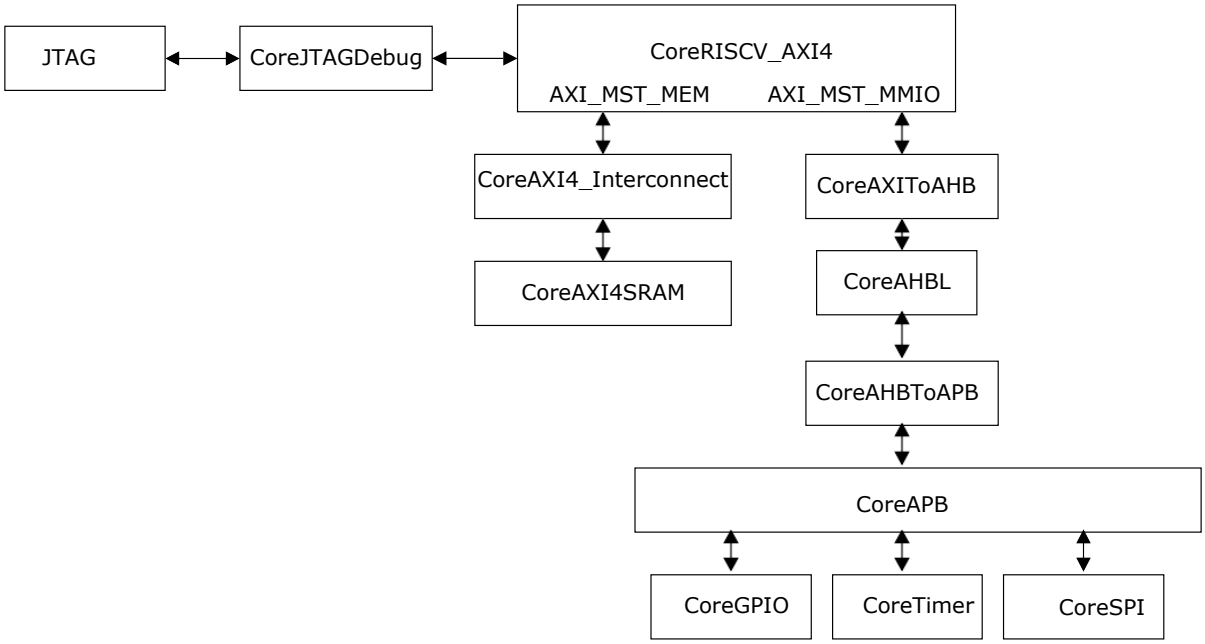
The CoreRISCV_AXI4 MEM interface ranges from 0x8000_0000 to 0x8FFF_FFFF. The code and data section of the RISC-V processor must be stored in this memory range.

The MMIO interface on CoreRISCV_AXI4 ranges from the 0x6000_0000 to 0x7FFF_FFFF. This memory range is used for the memory mapped IO accesses.

Use the CoreJTAGDebug IP to access the CoreRISCV_AXI4 through the JTAG interface. The CoreJTAGDebug IP instantiates a UJTAG macro and has a module for data tunneling. The JTAG standard signals are automatically connected to the correct pins of the device by Libero.

The following figure shows the debug mode configuration on the PolarFire device.

Figure 2 • **Debug Mode Hardware Configuration on the PolarFire Device**



In this configuration, the application image is directly loaded to the fabric SRAM in the code memory space using FlashPro and SoftConsole. The CoreRISCV_AXI4 IP then executes the application after coming out of the reset.

5.2 Production Mode configuration

In the production mode configuration, the CoreRISCV_AXI4 IP boots from any one of the following non-volatile memories available on the PolarFire device:

- µPROM
- sNVM
- On-board SPI Flash

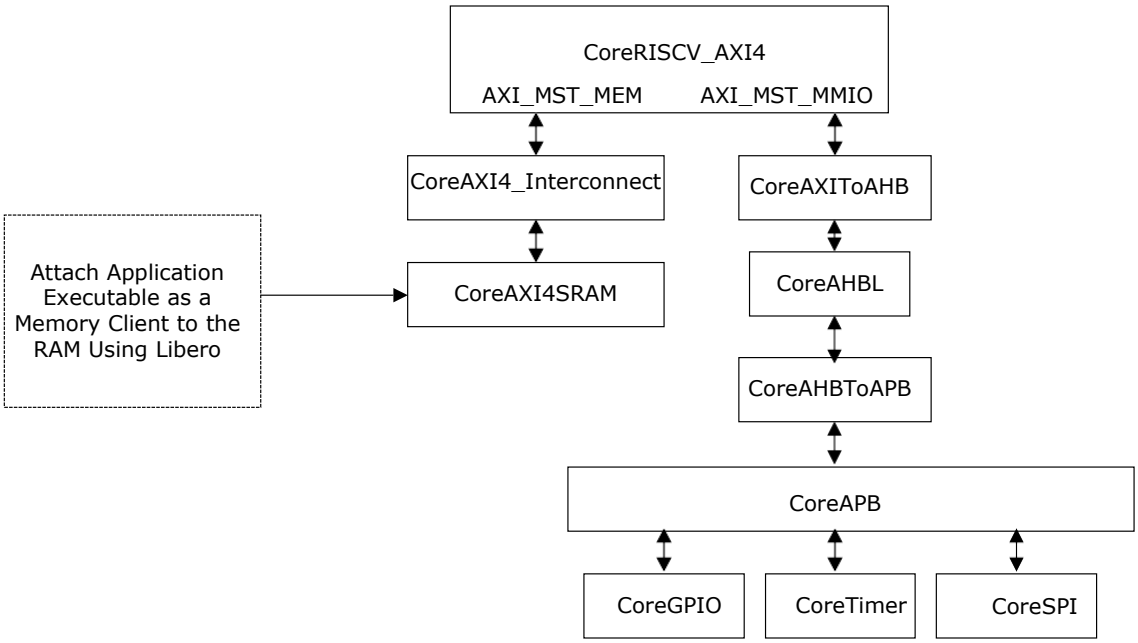
In this configuration, the design initialization method is used to initialize the fabric SRAM with the application executable stored in one of the non-volatile memories in the system.

Follow these steps to implement this configuration in Libero:

1. Connect the CoreAXI4SRAM IP to the code memory space of the CoreRISCV_AXI4 IP memory map.
2. Use the CoreAXI4SRAM configurator to attach the memory initialization file. This file must be the application executable.
3. Use the Design and Memory Initialization interface of Libero after Place and Route to store the memory client on the selected non-volatile memory.
   This memory client will be part of bit-stream output of the Libero design. On system reset, System Controller initializes the destination CoreAXI4SRAM with memory client from the selected memory.

The CoreRISCV_AXI4 IP executes the application executable from CoreAXI4SRAM after coming out of reset. The system integration diagram for this configuration is shown in the following figure.

Figure 3 • **Production Mode Configuration Using Design Initialization**



The CoreBootStrap IP can also be used to boot the application code stored in the on-board SPI flash memory.

To use the CoreBootStrap IP as the bootloader, connect CoreAXI4SRAM on code memory space of CoreRISCV_AXI4 memory map. The application executable must be pre-loaded on the SPI Flash memory. The CoreBootStrap IP holds the CoreRISCV_AXI4 in reset while copying the application executable from the on-board SPI flash memory to the CoreAXI4SRAM IP after system reset. The CoreBootStrap IP then releases reset on CoreRISCV_AXI4 after copying the application executable to CoreAXI4SRAM. The CoreRISCV_AXI4 IP now fetches and executes the application code from CoreAXI4SRAM.

In these two configurations:

- The application code always executes from the fabric SRAM
- The linker script must be in sync with the address space of the fabric SRAM

6 RISC-V HAL Configuration

RISC-V HAL provides an infrastructure code for handling the boot sequence and system configuration of the CoreRISCV_AXI4 soft processor. RISC-V HAL relies on the hardware configuration data generated during the Libero design flow. RISC-V HAL must not be configured manually.

RISC-V HAL consumes hardware configuration data in the form of C header files and data structures. Once the Libero design flow is completed, the following information must be extracted from Libero into the firmware project:

- The Memory map
- The Interrupt mapping from peripheral Cores to the CoreRISCV_AXI4 IP
- Clock dependencies
  - To set the tick rate (if used)
  - Clock setup for certain cores and the corresponding core drivers

The linker script and `hw_platform.h` files contain hardware related information extracted from the Libero design.

6.1 Linker Script

The linker script file for the tool must be updated with the settings being used. Copy an example linker script file from the example project provided with RISC-V HAL to the root directory and edit it as required.

This will be the linker file used in the application project.

6.2 <hw_platform.h> File

Copy and rename the `sample_hw_platform.h` file to the application root directory from the RISC-V HAL package. For more information, see the instructions given in the `sample_hw_platform.h`.

Take the information from the Libero project and enter it in the <hw_platform.h> file.

7 RISC-V HAL Features

This section describes the features of RISC-V HAL.

7.1 Interrupt Service Routines

RISC-V HAL provides basic resources for handling traps in the RISC-V processor. A trap is generated in the RISC-V processor due to asynchronous interrupts or synchronous exceptional conditions happening within the RISC-V processor.

The interrupts in the RISC-V processor are categorized as Internal interrupts or external interrupts. The CoreRISCV_AXI4 IP supports the following internal interrupts:

- Machine mode timer interrupt
- Machine mode software interrupt

The CoreRISCV_AXI4 IP supports up to 31 external interrupts. The external interrupts are handled through the PLIC controller. Default exception and interrupt handler functions are provided for all the internal and external interrupts.

These default exception and interrupt handler functions provided by RISC-V HAL are overridden by the application by providing functions with predefined names. These function names are listed in the following sections.

7.1.1 Internal Interrupts

The following table lists the function names that must be used by the application to provide a handler for a specific CoreRISCV_AXI4 internal interrupt.

Table 1 • **CoreRISCV_AXI4 Internal Interrupts**

Handler Function	Description
SysTick_Handler()	Machine mode timer interrupt handler
Software_IRQHandler()	Software interrupt handler

RISC-V HAL sets the system tick interrupt by using the `SysTick_Config()` function.

The software interrupt is generated using the `raise_soft_interrupt()` function. The function clear_soft_interrupt() is used to clear the previously raised software interrupt.

7.1.2 External Interrupts

The external interrupts can be controlled by the application using the PLIC interrupt control functions. Each interrupt source is identified in these functions using a unique PLIC interrupt number that is passed as an argument during the function call. The following PLIC configuration functions are provided as part of RISC-V HAL:

```
static inline void PLIC_init(void)

static inline void PLIC_EnableIRQ(IRQn_Type IRQn)

static inline void PLIC_DisableIRQ(IRQn_Type IRQn)

static inline void PLIC_SetPriority(IRQn_Type IRQn, uint32_t priority)

static inline uint32_t PLIC_GetPriority(IRQn_Type IRQn)

static inline uint32_t PLIC_ClaimIRQ(void)

static inline void PLIC_CompleteIRQ(uint32_t source)
```

The following table summarizes the PLIC interrupt numbers and the name of the handler function for each interrupt source. Each interrupt number and its handler function corresponds to an external

interrupt pin on the CoreRISCV_AXI4 soft IP. Add the handling of an external interrupt in the application by creating a function using one of these predefined exception handler names.

The names of these exception handler functions are defined as part of the startup code of RISC-V HAL. The default implementation provided as part of the startup code of RISC-HAL is defined with weak linking. Hence, any handler function of the same name defined as part of the user implementation overrides the default implementation.

Note that the function `PLIC_DisableIRQ` must not be called from the external interrupt handler. The return value of these functions must be used to indicate whether the external interrupt is to be disabled or to be kept enabled at the end of the execution of the external interrupt handler. The constants `EXT_IRQ_KEEP_ENABLED` and `EXT_IRQ_DISABLE` must be used by the user defined external interrupt handler for this purpose. The `PLIC_DisableIRQ` function can be used to disable the external interrupt from outside external interrupt handler function.

Table 2 • **External Interrupts Summary**

Interrupt Number	Handler Function	Interrupt Source
1 - 32	External_1_IRQHandler External_2_IRQHandler ... External_31_IRQHandler	Input pin on CoreRISCV_AXI4

7.2 Redirecting Standard Library Output to MMUART

RISC-V HAL enables debugging by redirecting the output of standard library functions such as `printf()` to a UART. The UART must be instantiated using CoreUARTapb.

This feature is enabled by adding the following directives to your project settings:

```
MSCC_STDIO_THRU_CORE_UART_APB
```

To use this feature, the following additional directives must be set.

```
#define MSCC_STDIO_UART_BASE_ADDR
```

This parameter must be set to the base address of the CoreUARTapb. The value of MSCC_STDIO_UART_BASE_ADDR is obtained from the Libero design.

```
#define MSCC_STDIO_BAUD_VALUE 57600
```

The baud rate is set to 115200 by default. Define the following parameter to use a different baud rate.

8 SoftConsole

This section describes the supported SoftConsole version and the linker script configuration required to set in the **Project Settings** window of SoftConsole.

8.1 Supported SoftConsole Versions

This version of RISC-V HAL supports SoftConsole v5.1 or later.

8.2 Linker Script

RISC-V HAL provides an example linker script file. The linker script must be edited with the details of your hardware memory map and the memory range in which the user program should reside. This script must be used with the SoftConsole debugger.

Specify the path of this script in **Project Properties->RISC-V GCC/Newlib C Linker->General** as shown in the following figure.

Figure 4 • **Selecting SoftConsole Linker Script**