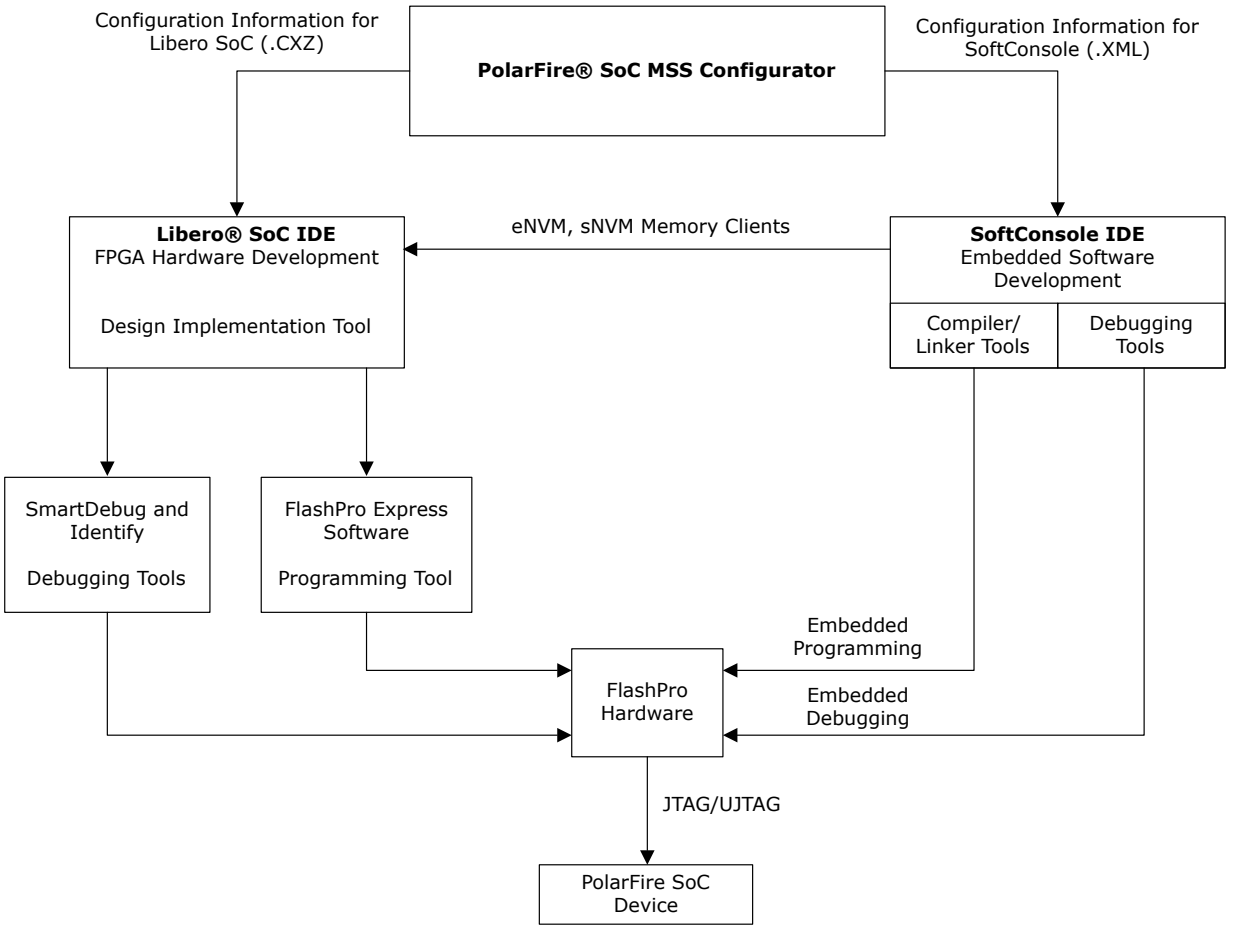
# PolarFire® SoC Software Development and Tool Flow User Guide

## Introduction (Ask a Question)

The PolarFire® SoC tool flow has been constructed to allow embedded designers and FPGA designers to develop applications in the domain of their choice. Embedded designers and FPGA designers prefer using a tool flow they are familiar with. The recommended starting point when designing with PolarFire SoC FPGA is the PolarFire SoC MSS Configurator tool that graphically guides the user to define the initialization parameters for the microprocessor subsystem, MSS peripherals, DDR, and the interfaces between the processor subsystem and the FPGA fabric. The tool is also used to configure MSS I/O.

The following flow diagram illustrates the high-level tool flow.

**Figure 1.** High-Level Tool Flow




After the user completes a configuration, the MSS configurator exports the files needed for the embedded software development flow and FPGA developers. The XML file contains the system configuration information needed to generate header files for Bare Metal system startup code included in the Hart Software Services

(HSS) to configure the microprocessor subsystem to the expected state. The `<project_name>.cxz` can be imported into Libero® SoC (v12.5 and above) and used by the FPGA designer to interface their design to the processor subsystem. The PolarFire SoC MSS Configurator tool is used to change the state of the microprocessor subsystem or any of the interfaces between the FPGA fabric and the microprocessor subsystem.

As the design progresses through the development process, different types of data are shared between the FPGA designer (Libero SoC) and the embedded designer (SoftConsole). The following are some of the examples.

- SoftConsole: outputs that can be part of the FPGA bitstream or programmed directly through SoftConsole using the FlashPro hardware.
  - Boot Mode configuration
  - Secure Boot Device Certificate
  - Embedded Non-Volatile Memory (eNVM) binary
  - Secure Non-Volatile Memory (sNVM) binary
- Libero SoC
  - FPGA memory map
  - FPGA design

The FPGA designer in collaboration with the embedded software designer defines and refines the MSS memory map within the FPGA. The files generated by the PolarFire SoC MSS Configurator must be shared with the embedded software developer for further development.

## References (Ask a Question)

- For information about the PolarFire SoC MSS, see PolarFire SoC MSS Technical Reference Manual.
- For information about MSS peripherals, see PolarFire SoC MSS Technical Reference Manual.
- For information about device power-up, see PolarFire FPGA and PolarFire SoC FPGA Power-Up and Resets User Guide.
- For more information about Bare Metal, Yocto, and Buildroot applications, see GitHub.
- For information about Yocto, see Yocto Project Reference Manual.
- For information about Buildroot, see Buildroot User Manual.
- For more information about PolarFire SoC MSS Configurator, see PolarFire SoC MSS Configurator.
- For more information about how to boot Linux® on Icicle kit using eMMC, see GitHub.

## Acronyms (Ask a Question)

The following acronyms are used in this document.

**Table 1.** Acronyms

| Acronym | Expanded |
|---|---|
| DTIM | Data Tightly Integrated Memory |
| DTS | Device Tree Source |
| eMMC | Embedded Multi-Media Controller |
| eNVM | Embedded Non-Volatile Memory/BootFlash |
| FSBL | First Stage Boot Loader |
| HAL | Hardware Abstraction Layer |
| Hart | Hardware Thread/Core/Processor Core |
| HSS | Hart Software Services |
| µPROM | Micro Programmable Read-Only Memory |
| LIM | Loosely Integrated Memory |
| MSS | Microprocessor Subsystem |
| MPFS | Microchip PolarFire SoC |
| OpenSBI | Open Source Supervisor Binary Interface |
| PMP | Physical Memory Protection |
| POR | Power-on Reset |
| PUF | Physically Unclonable Function |
| ROM | Read-only Memory |
| SBIC | Secure Boot Image Certificate |
| sNVM | Secure Non-volatile Memory |
| SRAM | Static Random-Access Memory |
| SSBL | Second Stage Boot Loader |
| WFI | Wait for Interrupt |
| ZSBL | Zero Stage Boot Loader |

# Table of Contents


Introduction....1

References....2

Acronyms....3

1. Development Tools....5
   - 1.1. PolarFire SoC MSS Configurator....5
   - 1.2. Libero® SoC....6
   - 1.3. SoftConsole....6
   - 1.4. FlashPro Express....25
   - 1.5. RISC-V GCC Bare Metal....25
   - 1.6. RISC-V Linux Toolchain....25
   - 1.7. Yocto....25
   - 1.8. Buildroot....26
   - 1.9. SmartDebug....26
   - 1.10. Identify....27

2. Software Stack....28
   - 2.1. RISC-V Libraries....28
   - 2.2. Hart Software Services (HSS)....29
   - 2.3. Bare Metal Library....29
   - 2.4. Linker Scripts....31
   - 2.5. Linux Images....32
   - 2.6. FreeRTOS™....33
   - 2.7. Third Party Tools....34

3. Application Development....35
   - 3.1. Device Boot and Configuration Process....35
   - 3.2. Boot Mode 0-Idle Boot....37
   - 3.3. Boot Mode 1-Direct Boot from eNVM....37
   - 3.4. Clock Management....41
   - 3.5. Physical Memory Protection (PMP)....41
   - 3.6. Generating Boot Images....42
   - 3.7. Bare Metal Development....47
   - 3.8. Linux Application Development....52

4. Appendix....62

5. Revision History....63

Microchip FPGA Support....64

Microchip Information....64
- Trademarks....64
- Legal Notice....64
- Microchip Devices Code Protection Feature....65

# 1. Development Tools (Ask a Question)

PolarFire SoC comes with a suite of tools to help create a complete hardware and software solution. The following table lists the suite of tools available for creating the FPGA and embedded design targeted for PolarFire SoC.

**Table 1-1.** Development Tools

| Tool | Description |
|---|---|
| PolarFire® SoC MSS Configurator | A standalone tool to configure the MSS clock frequencies, peripherals, DDR, fabric interfaces, and MSS I/O configuration. |
| Libero® SoC | Standard Microchip tool to configure the programmable section of the PolarFire SoC FPGA. |
| SoftConsole | Software development platform to develop and debug the Bare Metal and RTOS applications, which also includes debugging the software. |
| FlashPro Express | Available as a standalone tool or integrated as part of a Libero installation. Used for programming the MSS and programmable logic of the FPGA. |
| RISC-V GCC Bare Metal | The RISC-V GCC toolchain bundled with SoftConsole for Bare Metal development. |
| Yocto | An open source project to create Linux® distributions for embedded and IoT applications. |
| Buildroot | A tool to configure and generate embedded Linux distributions. |
| SmartDebug and Identify | Available as a stand-alone tool or integrated as part of Libero to debug the hardware in the MSS and programmable logic.<br>Identify is the Embedded Logic Analyzer tool for Microchip FPGA devices offered as part of the Libero SoC software tool suite. |

## 1.1 PolarFire SoC MSS Configurator (Ask a Question)

The PolarFire SoC MSS Configurator is a common tool to configure the PolarFire SoC MSS. It provides a seamless experience for the embedded software developers targeting the MSS and hardware engineers developing a solution using the MSS and the FPGA fabric. The PolarFire SoC MSS Configurator application is available in two options:

- As a standalone application
- As part of the Libero SoC Design Suite v12.5 and later

### 1.1.1 Installation (Ask a Question)

The PolarFire SoC MSS Configurator bundled with Libero is available at the following location in the Libero installation folder:

- `Windows: <$Installation_Directory>\Microsemi\Libero_SoC_vX.X\Designer\bin64\pfsoc_mss.exe`
- `Linux: <$Installation_Directory>\Microsemi\Libero_SoC_vX.X\bin64\pfsoc_mss`

**Note:** For Windows®, a start menu entry is created for easy launching.

For more details on how to install Libero, see www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/fpga/libero-software-later-versions#Documentation.

The PolarFire SoC MSS configurator can also be installed as a standalone application. For more information, see www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/fpga/libero-software-later-versions#Documentation.

### 1.1.2 Running the PolarFire SoC MSS Configurator (Ask a Question)

The Standalone MSS Configurator can run in one of the following modes.

### Batch Mode

The PolarFire SoC MSS Configurator application can be executed in the Batch mode for scripted execution as follows:

- Windows:

```
 <Libero SoC or Standalone MSS Configurator installation area>\bin64\pfsoc_mss.exe
-CONFIGURATION_FILE:<absolute path for configuration file name (.cfg)>
-OUTPUT_DIR:<absolute path for output directory>
```

- Linux:

```
<Libero SoC or Standalone MSS Configurator installation area>/bin64/pfsoc_mss
-CONFIGURATION_FILE:<absolute path for configuration file name (.cfg)>
-OUTPUT_DIR:<absolute path for output directory>
```

### Interactive Mode

The Standalone MSS Configurator (`pfsoc_mss`) can be launched from the Libero SoC installation directory (specified above) or from the Windows **Start Menu**.

For more details about Batch mode and Interactive mode usage, see www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/soc-fpga/polarfire-soc-mss-configurator#Documentation.

## 1.2 Libero® SoC (Ask a Question)

Libero SoC design suite offers high productivity with its comprehensive, easy to learn, easy to adopt development tools for designing with Microchip's power efficient Flash FPGAs, SoC FPGAs, and Rad-Tolerant FPGAs. The suite integrates industry standard Synopsys Synplify Pro® synthesis and Siemens ModelSim® simulation with best-in-class constraints management, debug capabilities, and secure production programming support.

For more details, see the *Libero SoC Design Flow User Guide* at:

www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/fpga/libero-software-later-versions#Documentation

### 1.2.1 Version Controlling Bitstreams (Ask a Question)

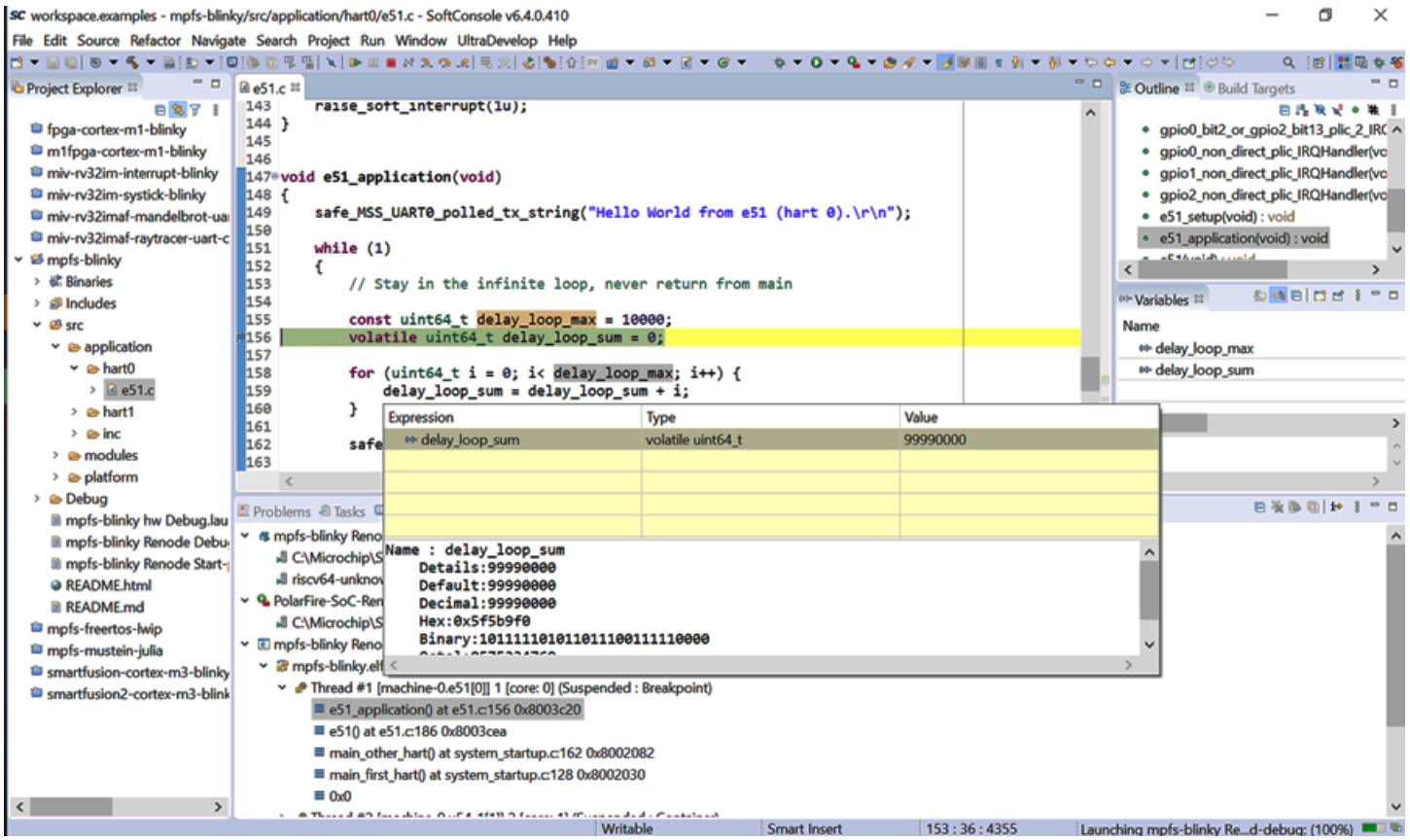
Microchip provides the `bitstream-builder.py` script that enables users to build a bitstream from version controlled sources, such as Libero SoC source files and SoftConsole source files. The `bitstream-builder.py` supports both Linux and Windows environments.

The bitstream builder script executes end-to-end flow from fetching different HDL and software source files, running the Libero SoC design flow, building the bitstream, and finally programming the hardware. For an example demo of how to bring up the PolarFire® SoC Icicle Kit using the bitstream builder script, see Bitstream Builder Example on GitHub.

## 1.3 SoftConsole (Ask a Question)

SoftConsole is an Eclipse-based IDE facilitating the development and debug of Bare Metal and RTOS-based C/C++ applications for Microchip SoC based FPGAs. It provides development and debug support for all Microchip SoC FPGAs and 32-bit soft IP CPUs.

**Figure 1-1.** SoftConsole IDE




For the latest SoftConsole Release Notes, see the SoftConsole webpage.

### 1.3.1 SoftConsole Presets (Ask a Question)

This section provides an outline of the default configurations for building projects for PolarFire SoC in SoftConsole v6.4 and later.

Existing SoftConsole projects can be downloaded from the PolarFire SoC Bare Metal Library: mi-v-ecosystem.github.io/redirects/repo-polarfire-soc-bare-metal-library. To import a project, follow these steps:

1. Click **File > Import**.
2. Select the **Existing Projects into Workspace** option.

**Notes:**

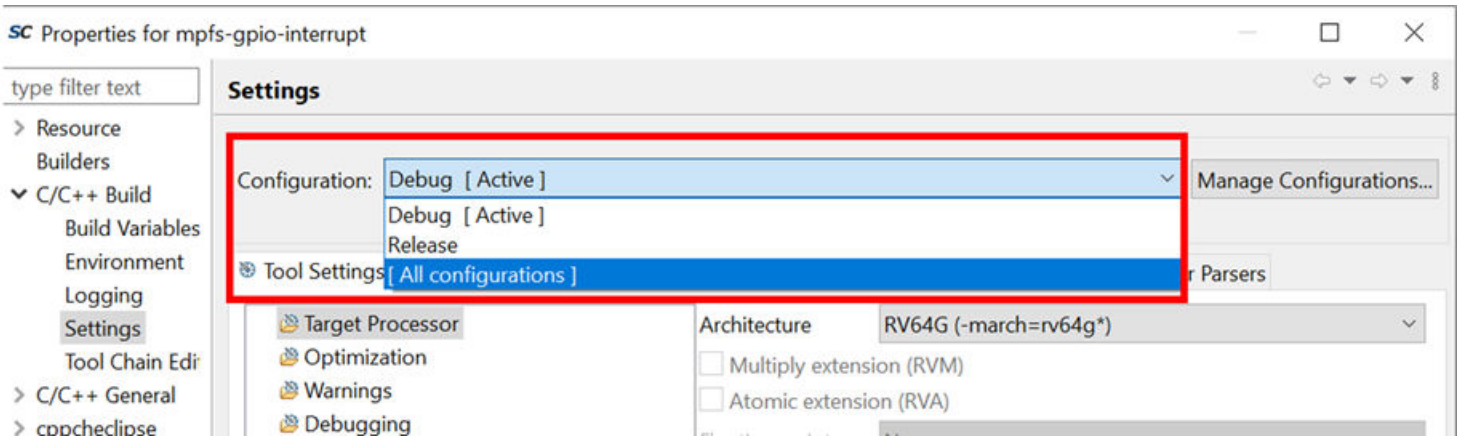
1. The downloaded projects are pre-configured with default settings and can be used as a base to build a new project.
2. The sample XML is included with the Bare Metal example projects; XML for reference designs can be found in the kit design folder on Github. For example, the PolarFire SoC Icicle Kit Libero reference design can be found here: mi-v-ecosystem.github.io/redirects/repo-icicle-kit-reference-design and contains an XML folder with reference XML for eMMC and SD card targets.

#### 1.3.1.1 Build Options (Ask a Question)

To view the properties of a project, right click an open project in the workspace and select **Properties**.

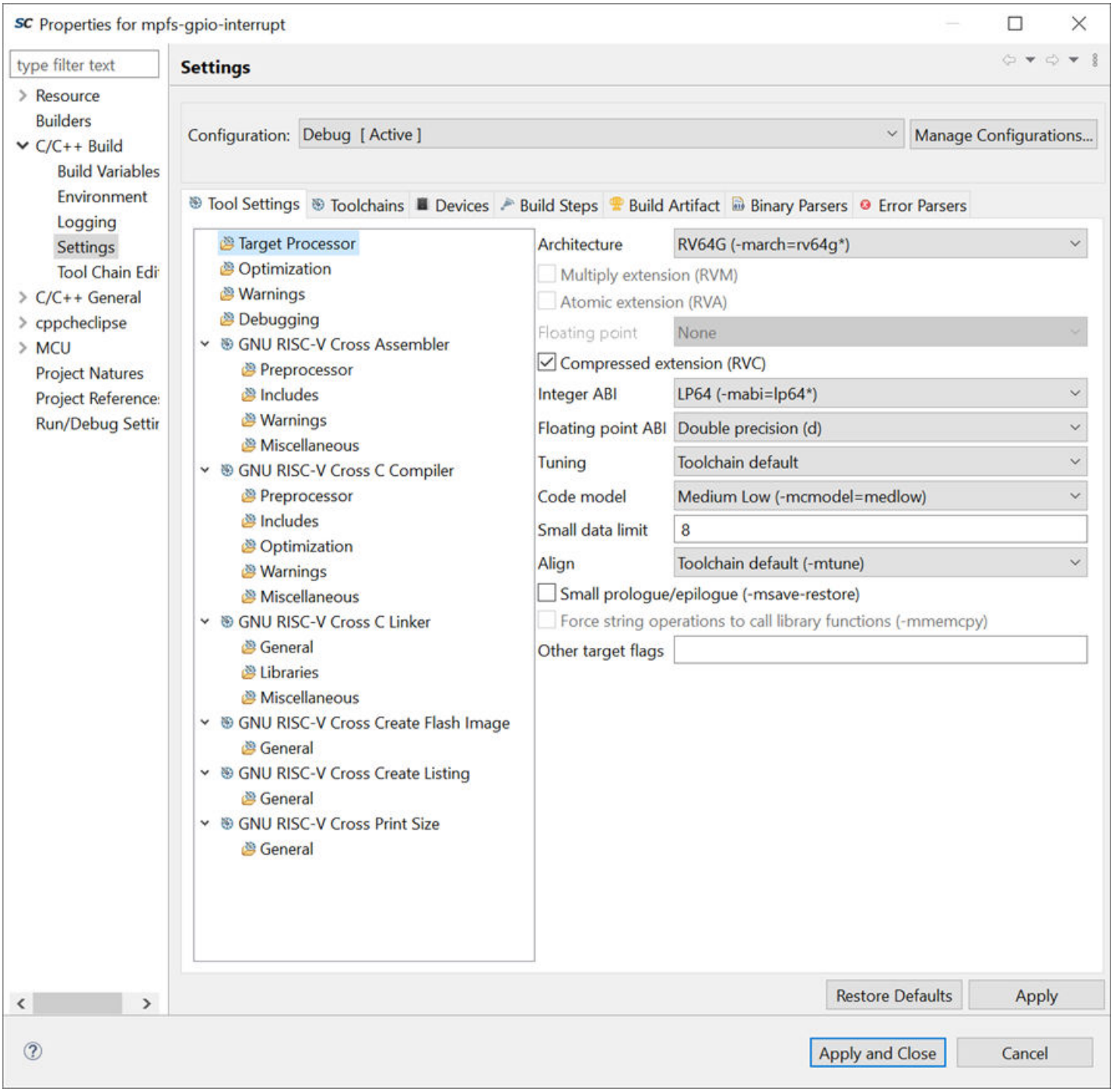
To configure build options, select **C/C++ Build** followed by **Settings**. These options can be configured globally or for individual build configurations using the **Configuration** field as shown in the following figure.

**Figure 1-2.** Project Properties—Build Configurations



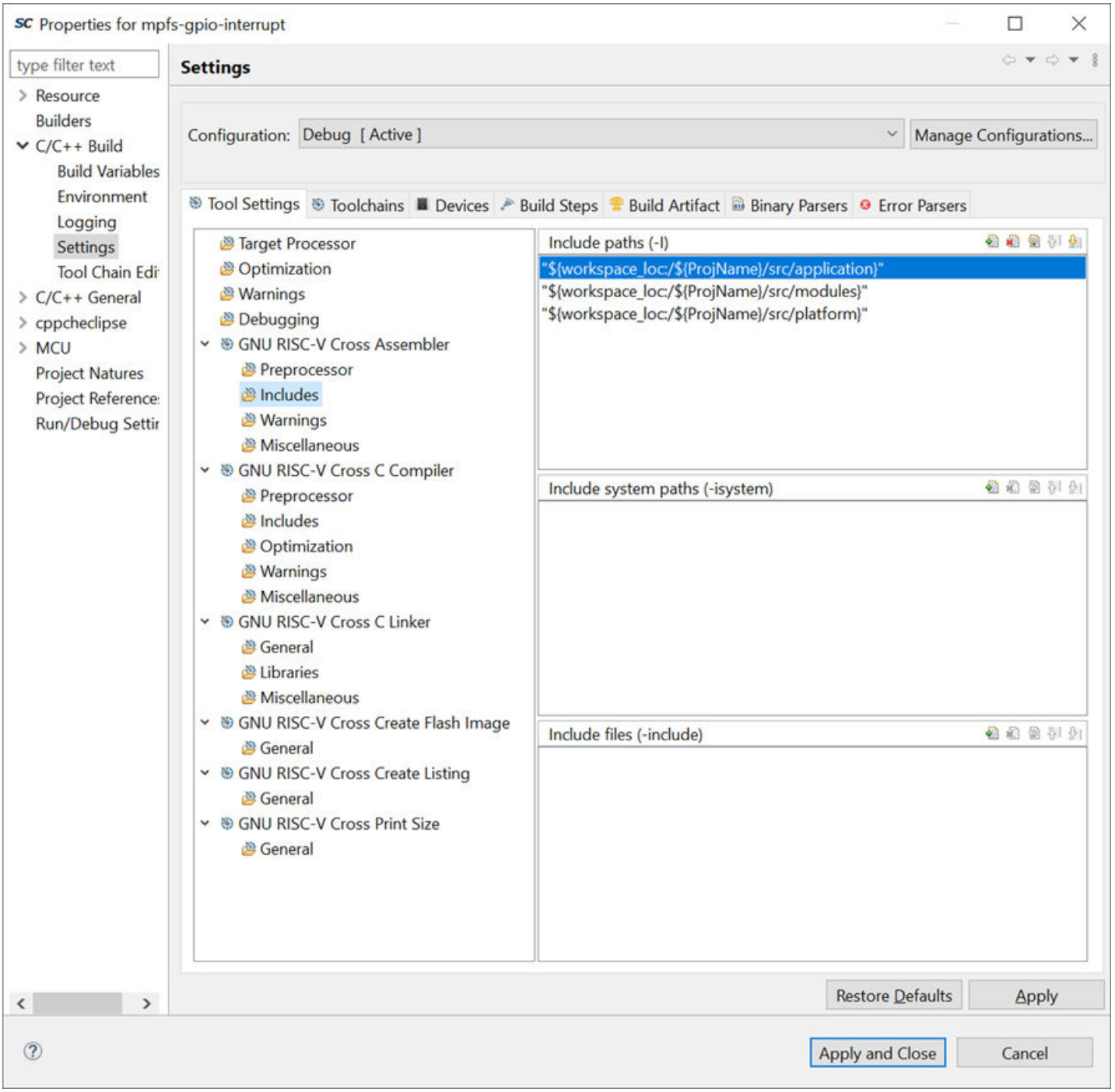
The default build configuration for the target processor section of a Bare Metal project are shown in the following figure.

**Figure 1-3.** Configuration—Target Processor



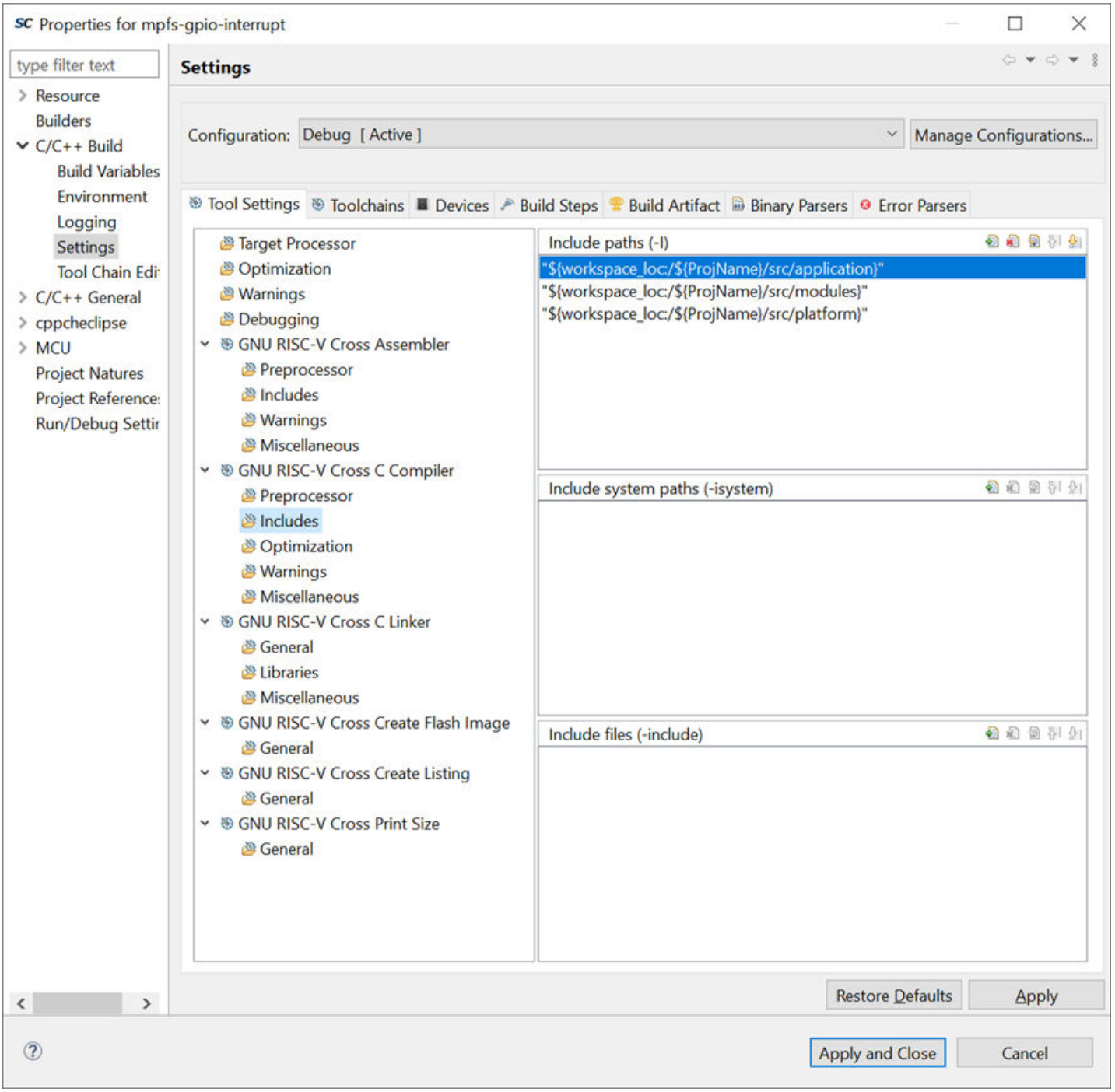
The following figure shows the default **Includes** for the **GNU RISC-V Cross Assembler**.

**Figure 1-4.** Configuration—GNU RISC-V Cross Assembler—Includes



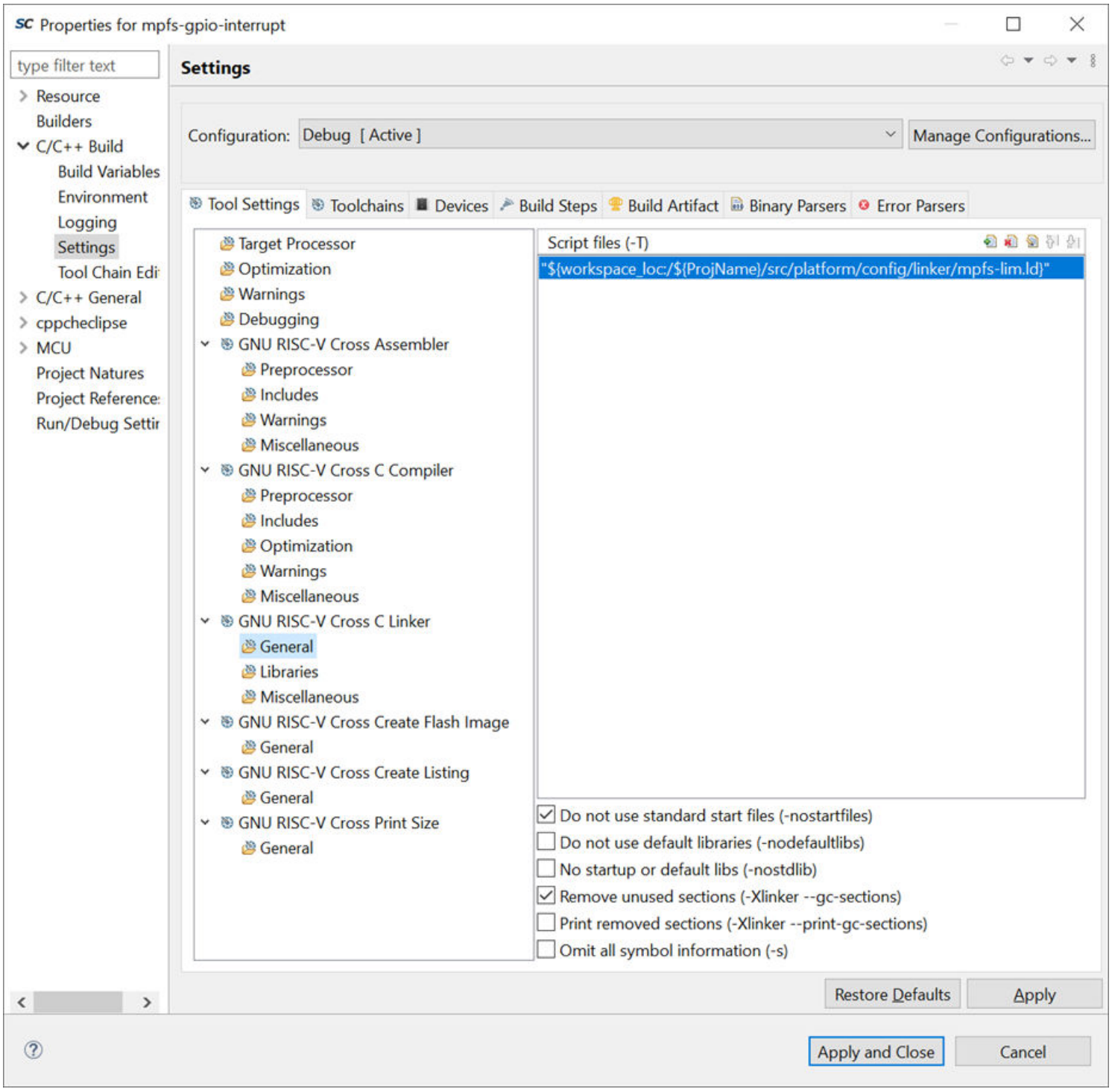
The default **Includes** for the **GNU RISC-V Cross C Compiler** are shown in the following figure.

**Figure 1-5.** Configuration—GNU RISC-V Cross C Compiler—Includes



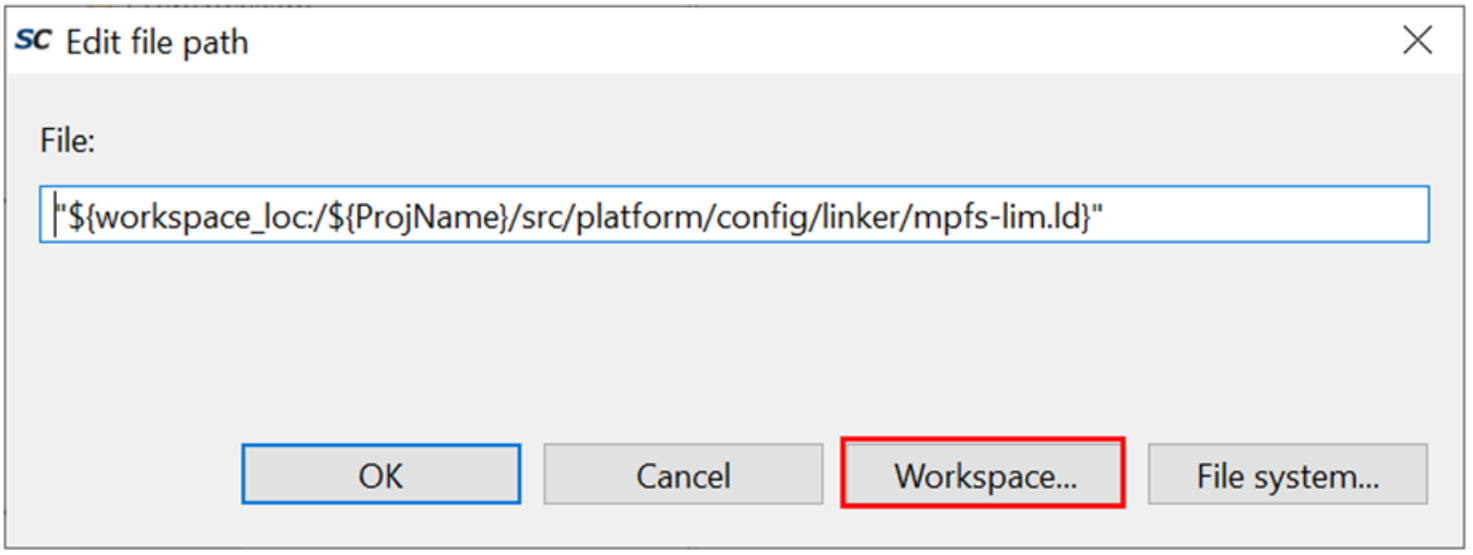
The default Linker script used by the **GNU RISC-V Cross C Linker** targets the LIM as show in the following figure and several sample Linker scripts to target different memory sources are included in the sample projects.

**Figure 1-6.** Configuration—GNU RISC-V Cross C Linker—General



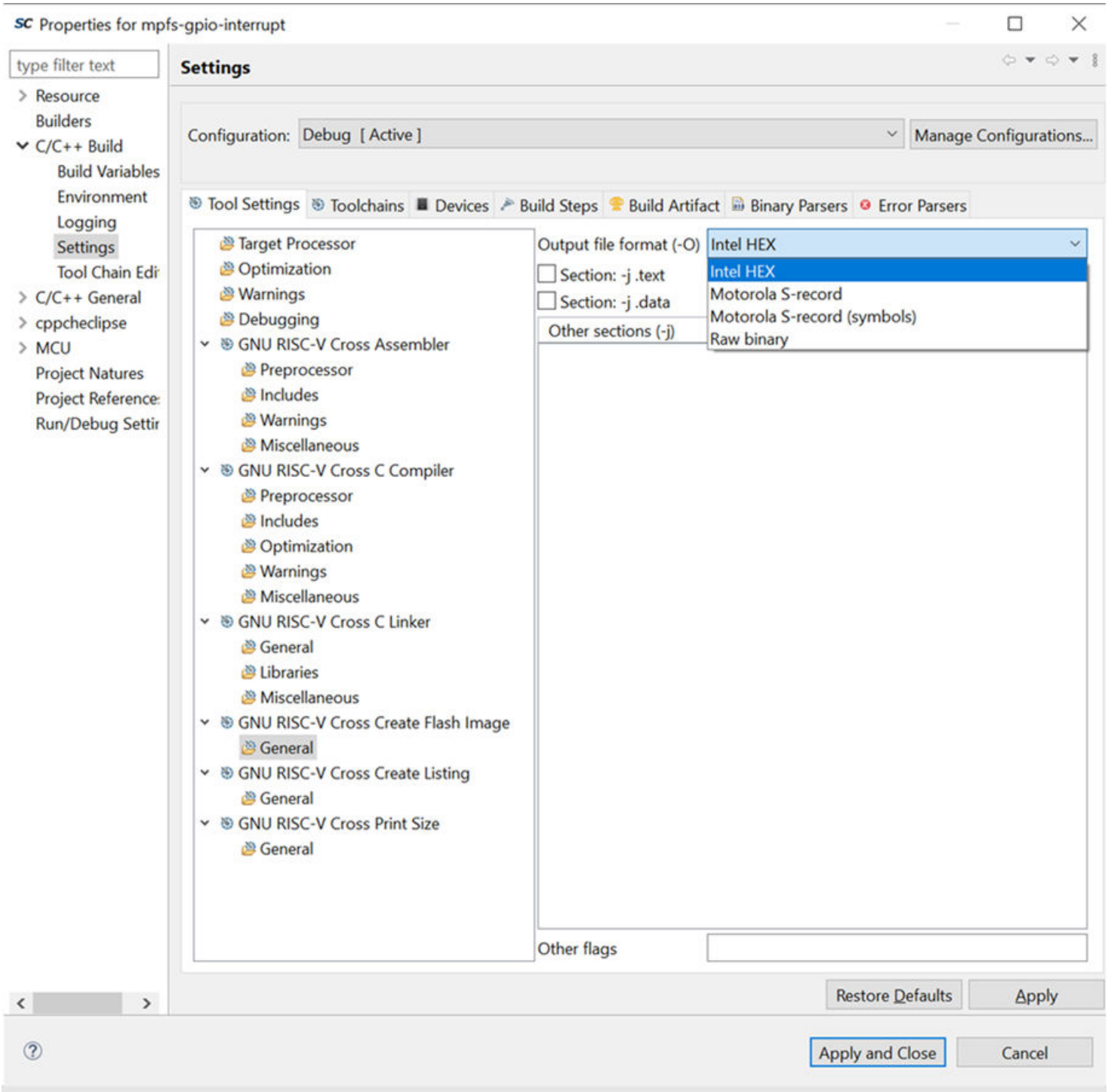
To change the Linker script, double click on the existing Linker script and select the **Workspace** option to select the Linker script from the project as shown in the following figure.

**Figure 1-7.** Edit File Path



The default output file format selected in **GNU RISC-V Cross Create Flash Image** for a Bare Metal project is **Intel HEX**. See the following figure.

**Figure 1-8.** Configuration—GNU RISC-V Cross Create Flash Image—General

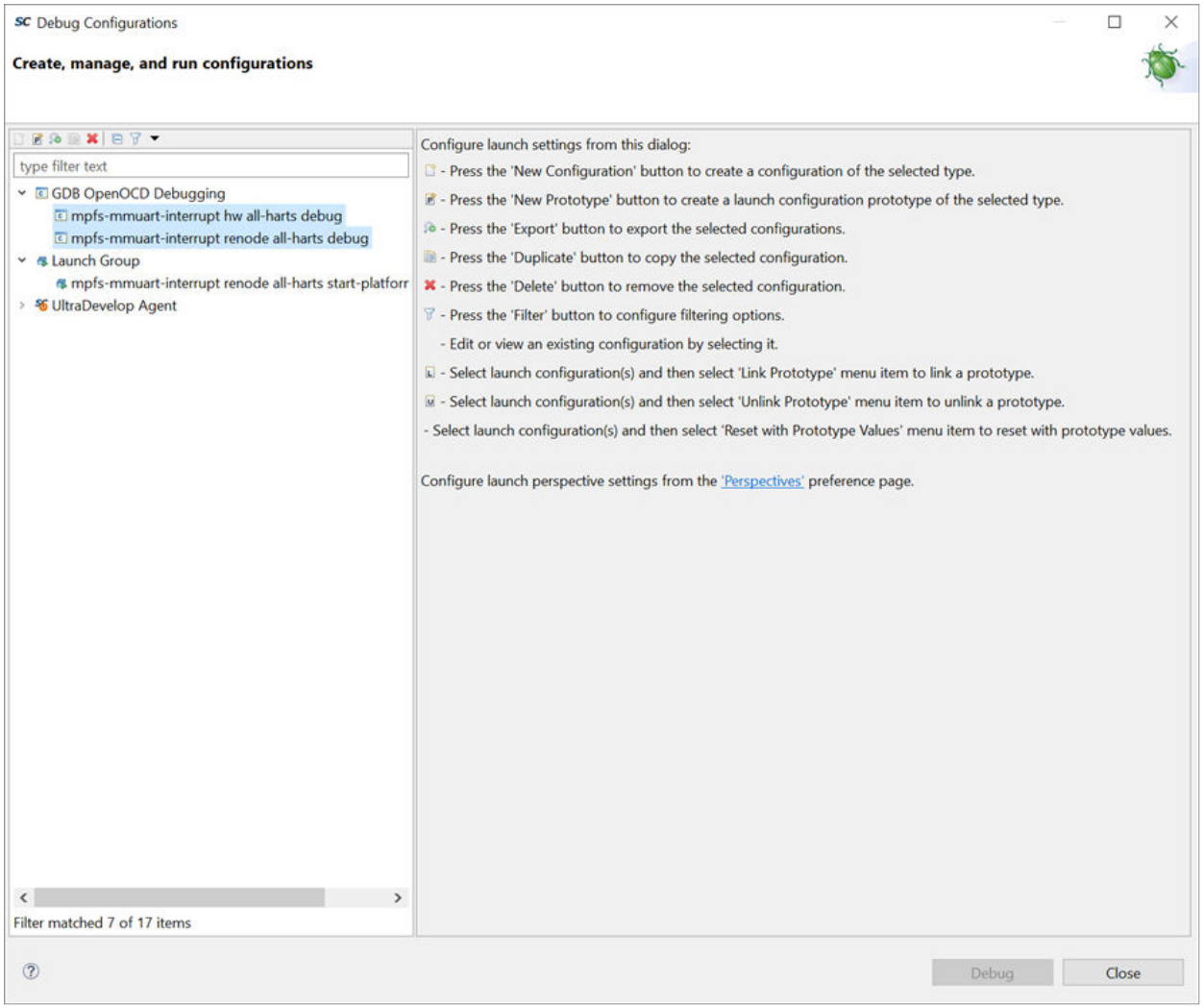


### 1.3.1.2 Debug Configurations (Ask a Question)

#### Debug Configuration Window

All the example Bare Metal projects contain a Renode™ debug configuration and a hardware debug configuration in the **GDB OpenOCD Debugging** section. The debug configuration also contains a **Launch Group** configuration option, which can be used to launch the Renode emulation platform and start the Renode debug configuration in one step as opposed to launching them independently. The following figure shows the **Debug Configuration** window.

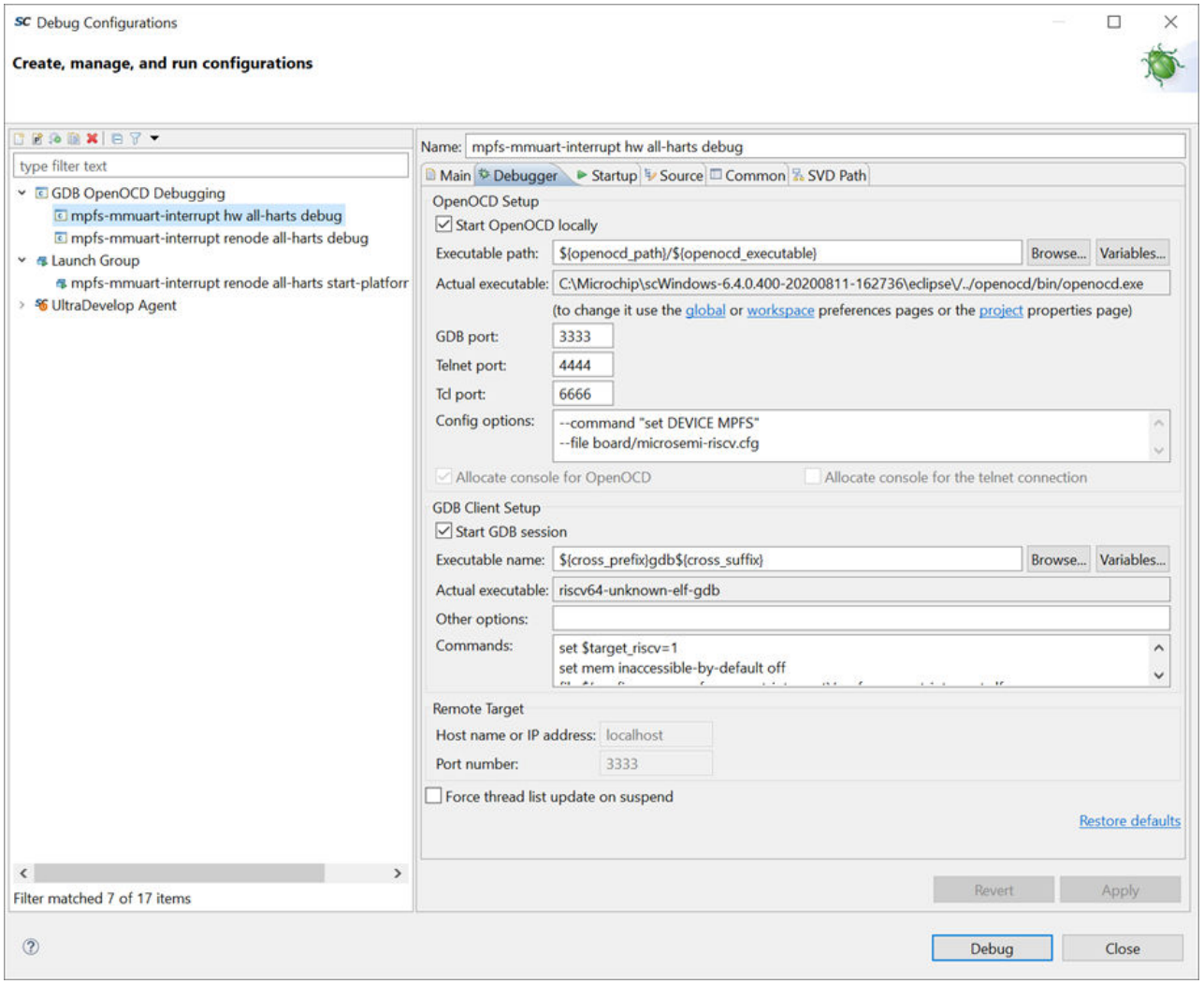
**Figure 1-9.** Debug Configuration Window



### Debugger Tab

The default settings in the **Debugger** tab is shown in the following figure.

**Figure 1-10.** Debug Configurations—Debugger




In the **Debugger** tab, under the **OpenOCD Setup > Config options** section, the default commands used are the following:

```
--command "set DEVICE MPFS"
--file board/microsemi-riscv.cfg
```

In the **Debugger** tab, under the **GDB Client Setup > Commands** section, the default commands used are the following:

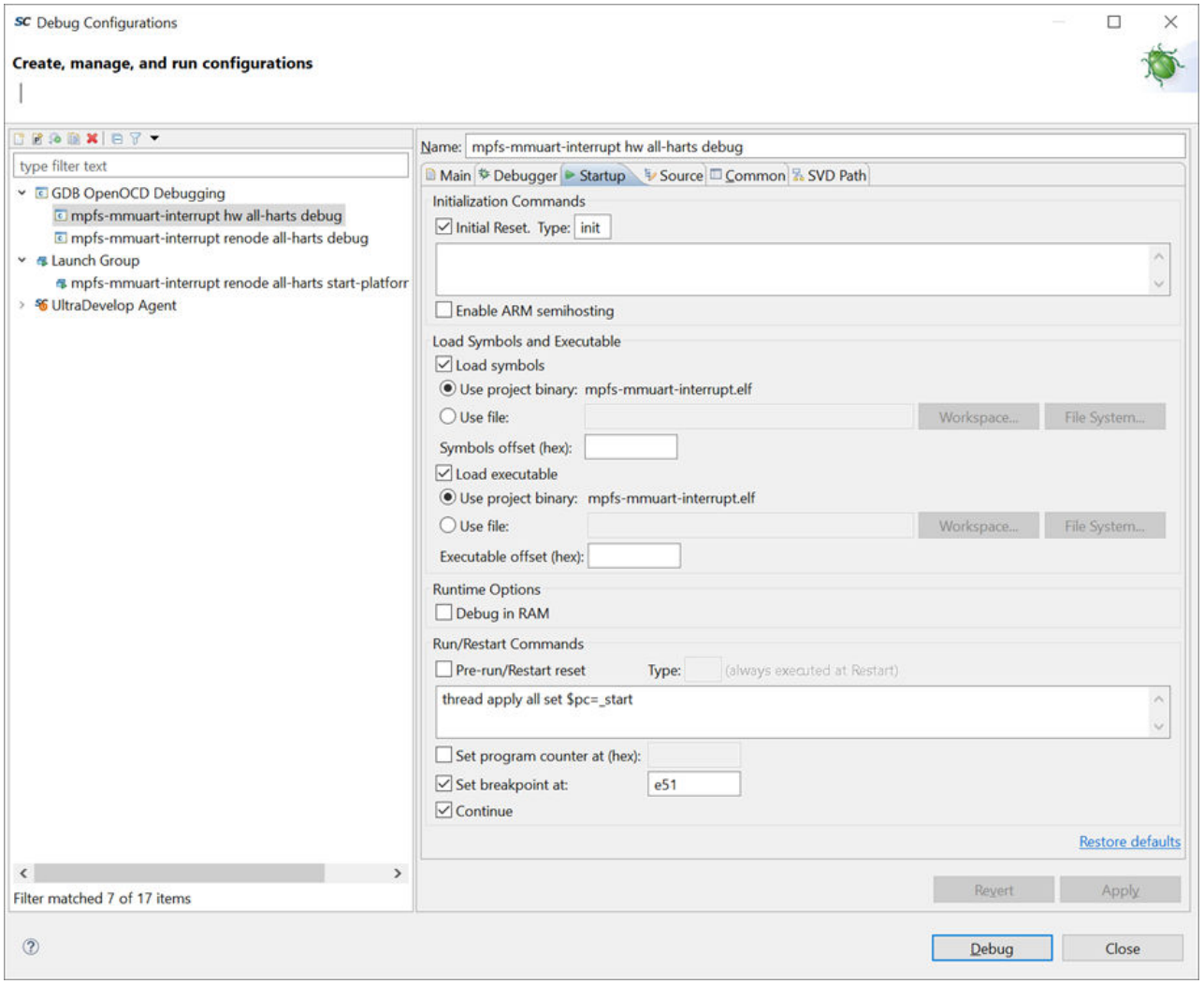
```
set $target_riscv=1
set mem inaccessible-by-default off
file ${config_name:mpfs-mmuart-interrupt}/mpfs-mmuart-interrupt.elf
```

**Note:** The file command shown in the preceding section must match the name of the project being used. The `{config_name:mpfs-mmuart-interrupt}/` section selects the folder for build files used in the configuration (that is, Debug or Release) and the `mpfs-mmuart-interrupt.elf` is the name of the `.elf` file produced on a successful build.

### Startup Tab

The default settings in the **Startup** tab is shown in the following figure.

**Figure 1-11.** Debug Configurations—Startup




In the **Startup** tab, under the **Run/Restart Commands** section, the default commands used are the following:

```
thread apply all set $pc=_start
```

## 1.3.2 Debugging using SoftConsole (Ask a Question)

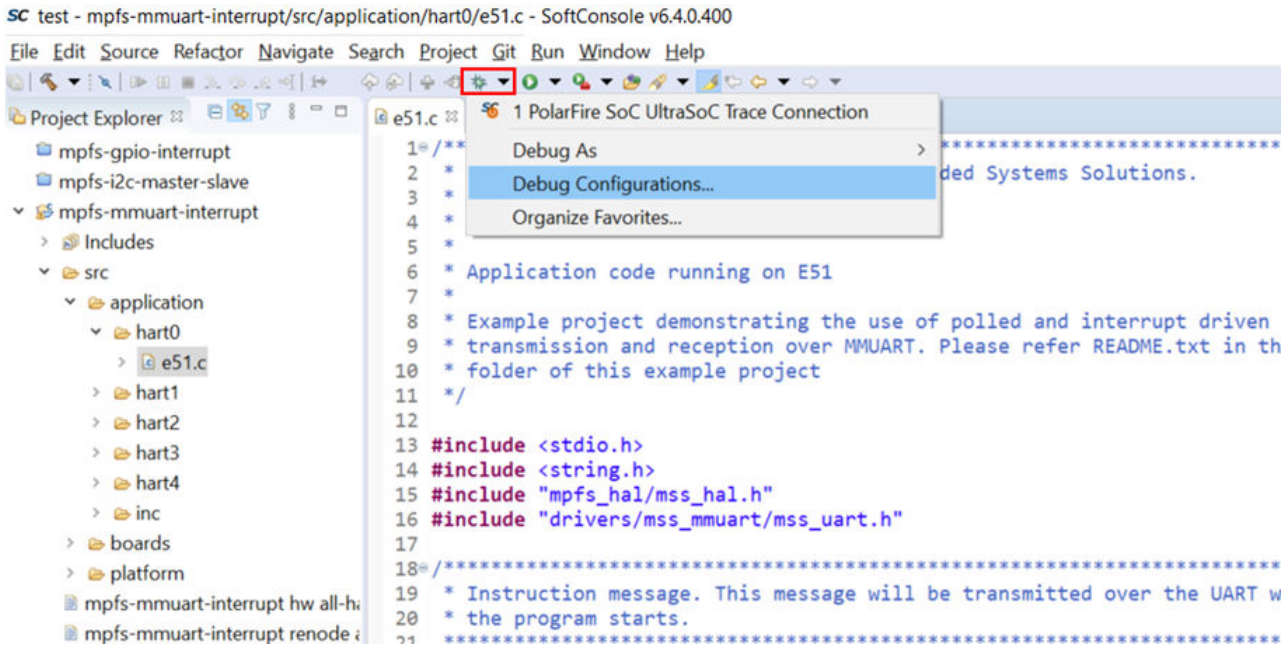
See the SoftConsole section for information on configuring builds and setting up debug configurations using SoftConsole.

### 1.3.2.1 Launching a Debug Configuration (Ask a Question)

Follow these steps to launch the debug configuration.

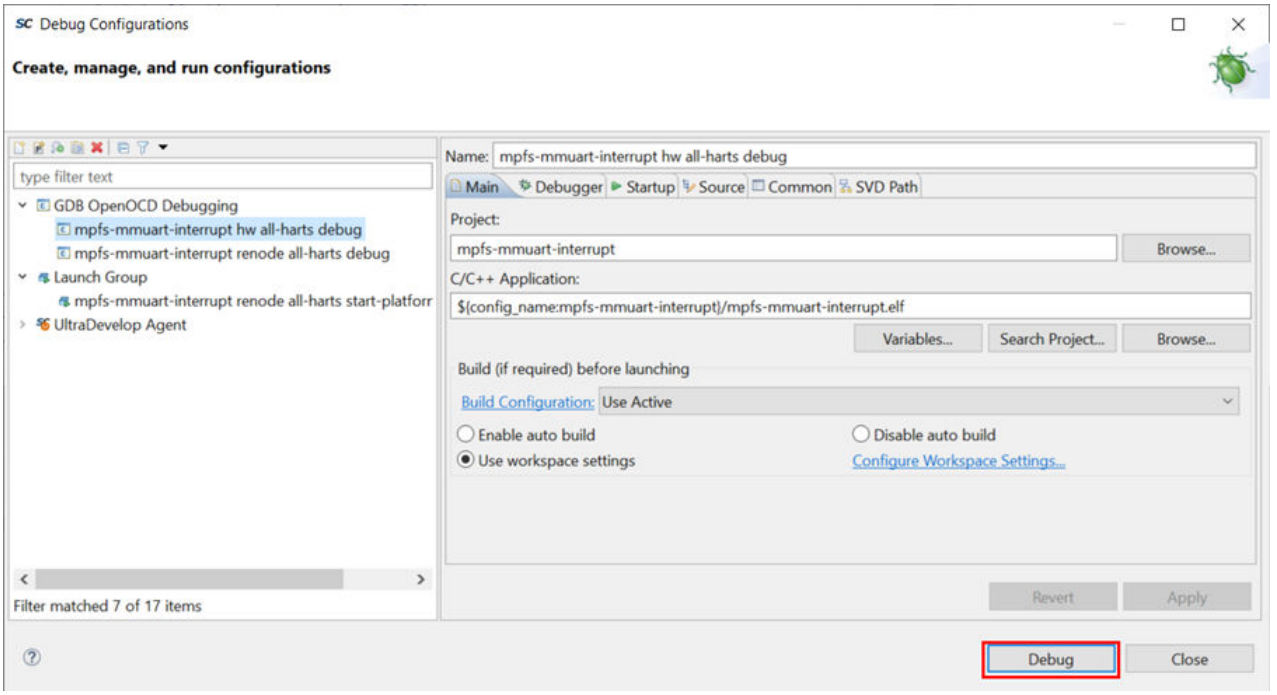
1. To launch a debug session, click the drop down arrow beside the debug icon and then, click the **Debug Configurations** option.

**Figure 1-12.** Debug Configurations




2. Select the debug session to be launched and click **Debug**.

**Figure 1-13.** Debug Configurations

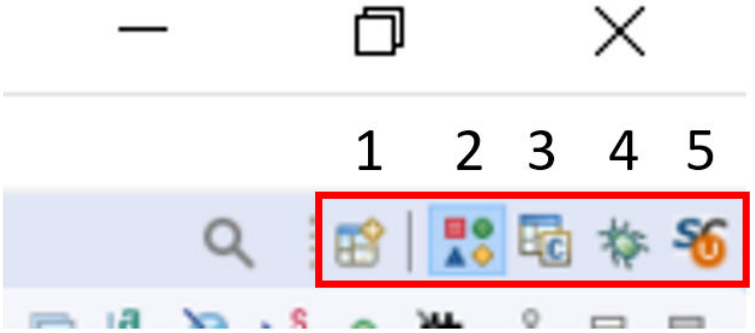



The debug session is launched and connects to the target.

### 1.3.2.2 Perspectives (Ask a Question)

There are several perspectives to choose from, each has a different layout optimized for different tasks. Perspectives can be chosen using icons at the top-right corner of the SoftConsole window. The following figure and table show icons for different perspectives and their description.

**Figure 1-14.** Choosing Different Perspectives

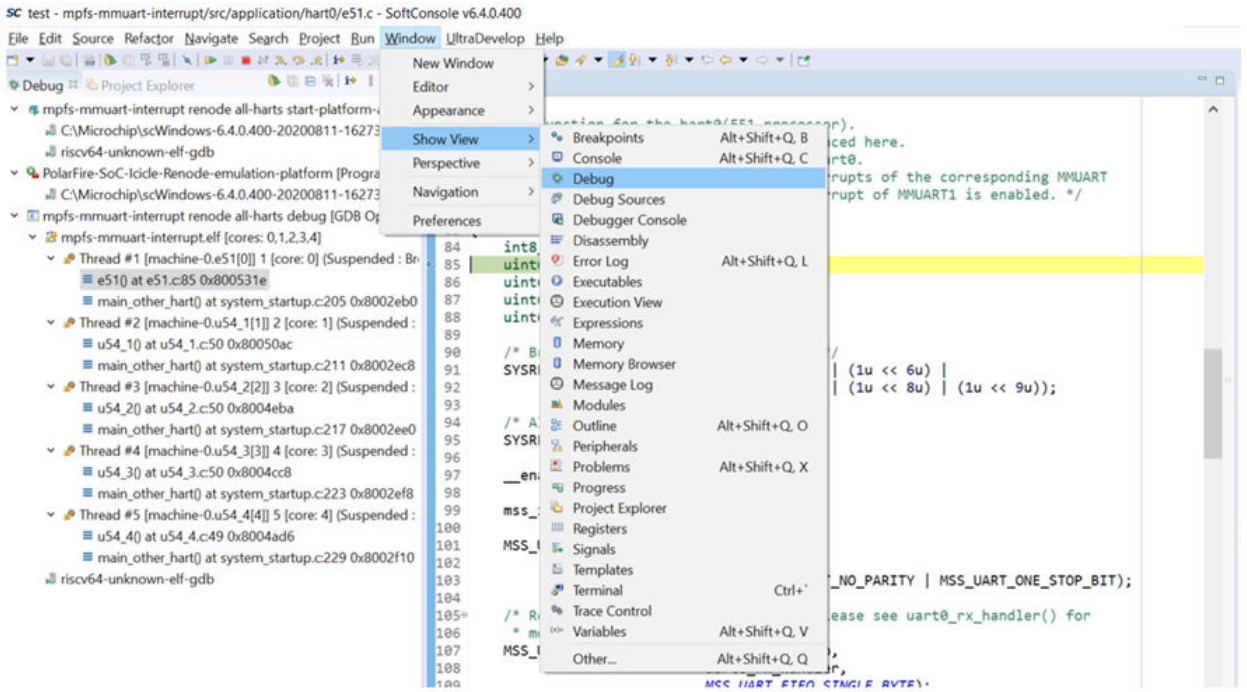



The description for each number is provided in the following table.

| | |
|---|---|
| **1** | View all available perspectives |
| **2** | Develop and debug perspective |
| **3** | C/C++ perspective |
| **4** | Debug perspective |
| **5** | UltraDevelop perspective |

To add windows or tools to the perspective, select **Window > Show View** and choose the required window. In the following figure, the **Debug** window is chosen.

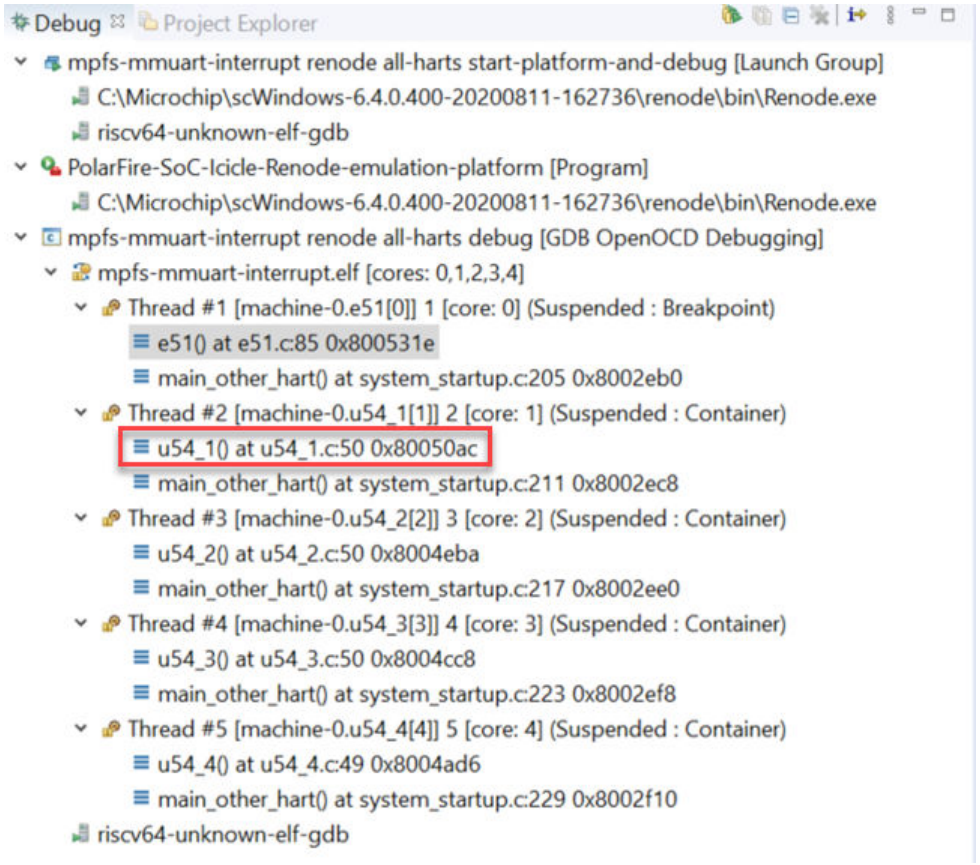
**Figure 1-15.** Adding Windows or Tools for Perspectives



### 1.3.2.3 Debugging a Hart (Ask a Question)

Each hart in the system appears as a different thread in the **Debug** window. This window is automatically shown in the **Debug** perspective and can be added to an active perspective. All the threads appear under the project name as shown in the following figure.

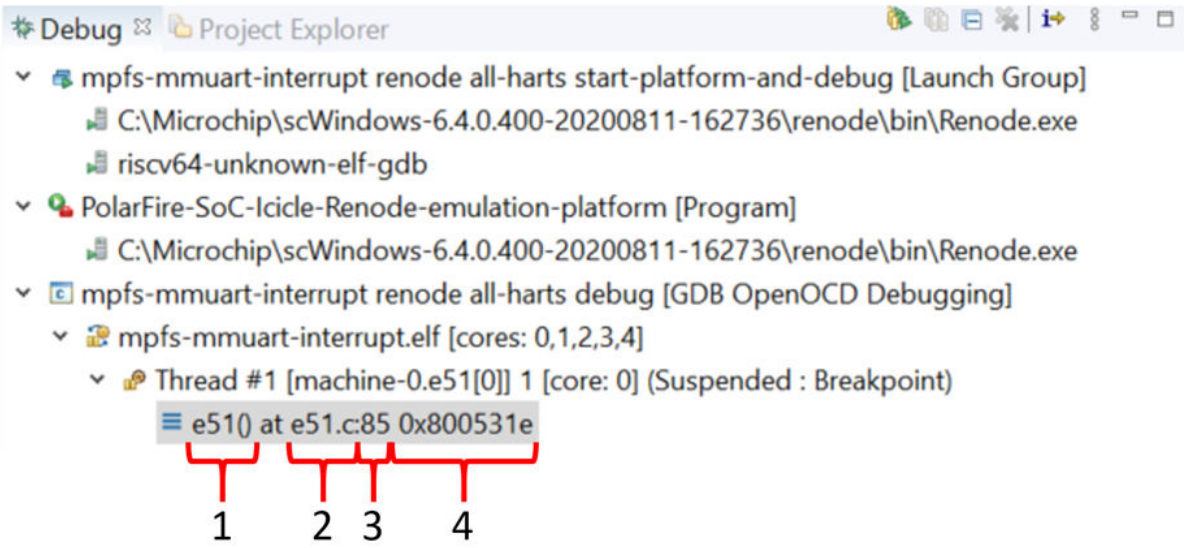
**Figure 1-16.** Debug Window Showing Harts as Threads




Each thread in the preceding figure represents a hart, and the function listed below each thread is the code being executed.

The following figure describes how to interpret the current function that is executed on a hart.

**Figure 1-17.** Current Function Executing on a Hart




The description for each number is provided in the following table.

| | |
|---|---|
| **1** | Shows the function that is currently being executed. |
| **2** | Shows the file in which function is found. |
| **3** | Shows the line of the file where the code is currently being executed |
| **4** | Memory address of the code that is executed |

### 1.3.2.4 Debug Session Controls (Ask a Question)

The following figure shows the buttons that are used to control the execution of the debug session.

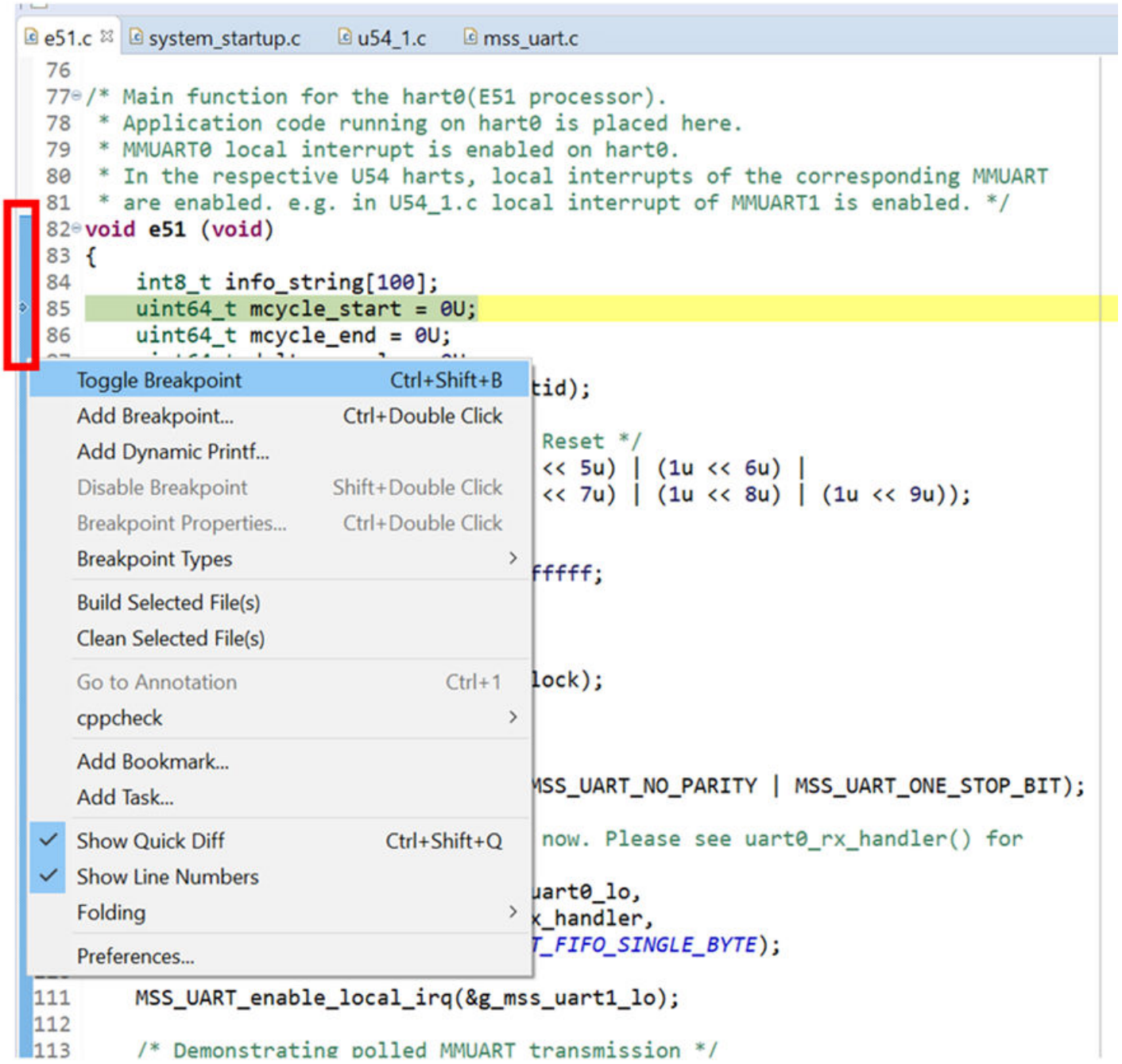
**Figure 1-18.** Buttons to Control Debug Session


1. Disable breakpoints 2. Resume 3. Halt 4. Stop 5. Step into 6. Step over 7. Step return 8. Instruction Stepping Mode

### 1.3.2.5 Setting Breakpoints (Ask a Question)

To add Breakpoints, right click beside the line number where the Breakpoint is required and select **Toggle Breakpoint**. Alternately, the same can be achieved by double clicking on the same location.

**Figure 1-19.** Toggle Breakpoint

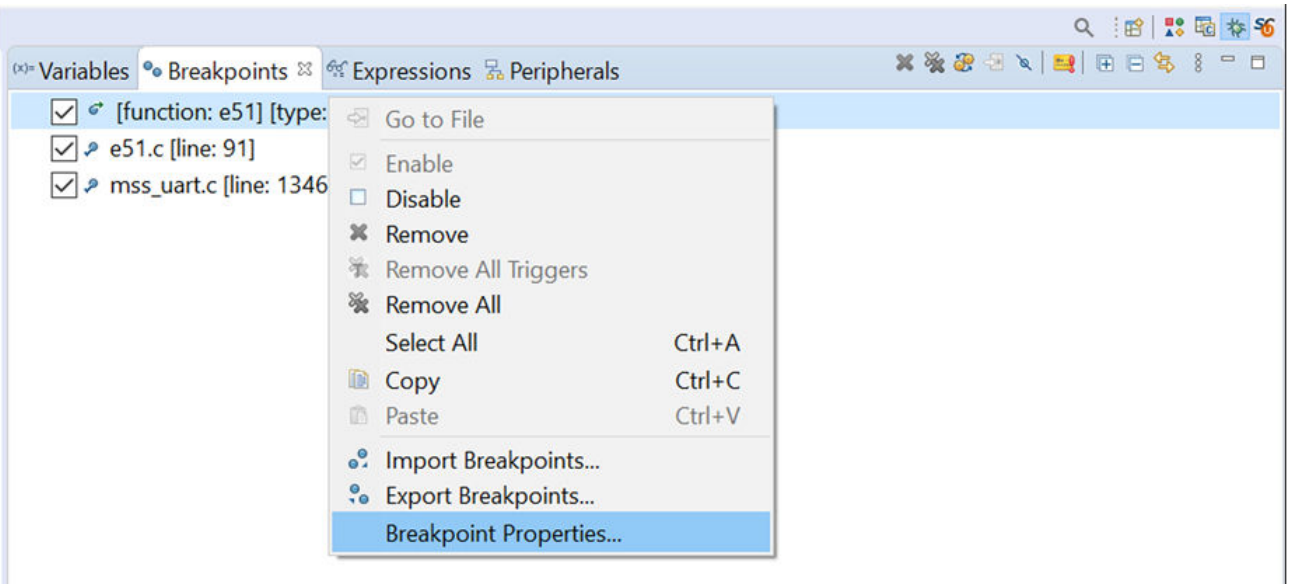



These breakpoints are set for all harts.

If a breakpoint is required only for a single hart and shared code is being run, the breakpoint can be filtered using the following steps:

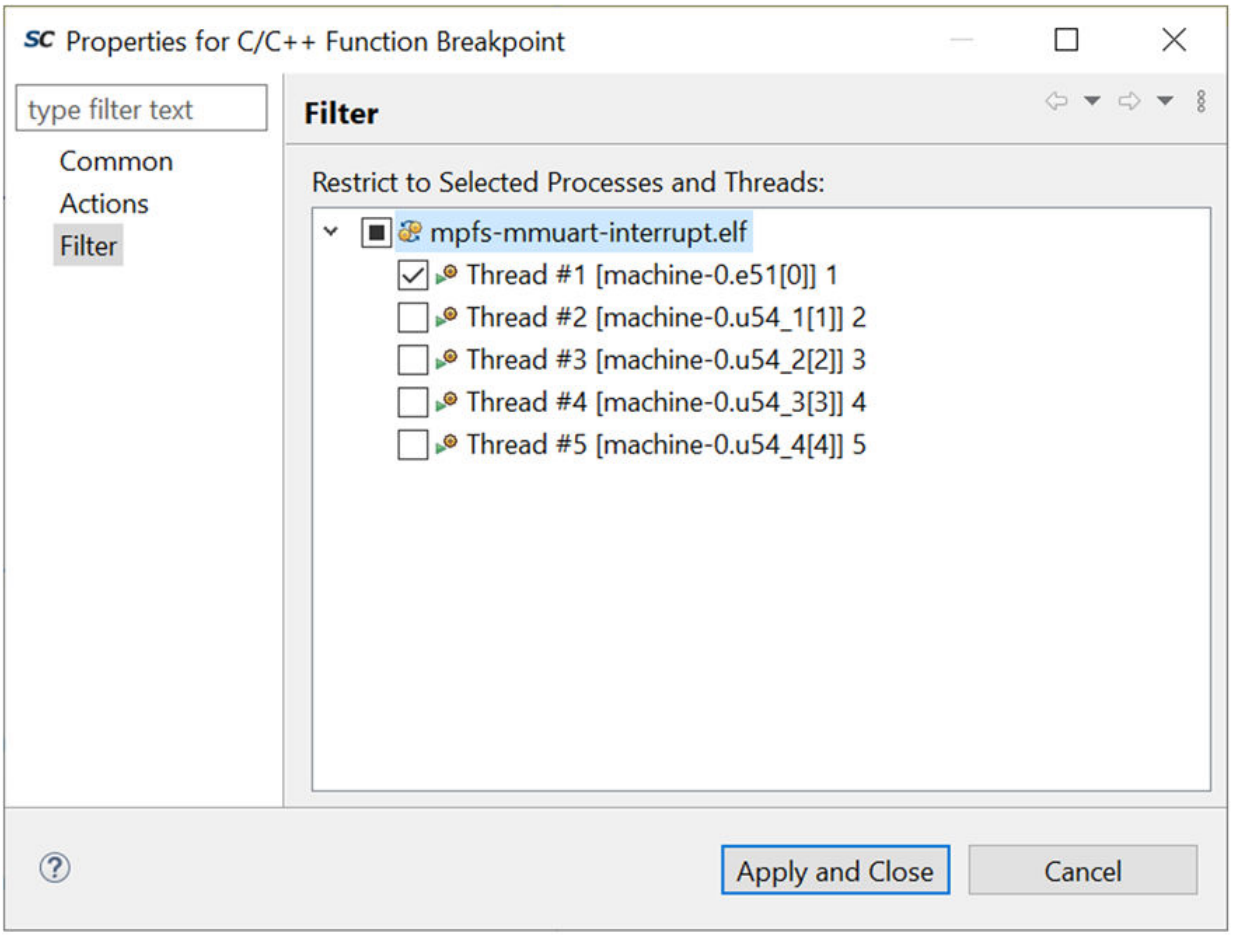
1. Click the **Breakpoints** window.
2. Right click the breakpoint to be filtered and select **Breakpoint Properties**.

**Figure 1-20.** Breakpoint Properties




3. Select the **Filter** option and enable the breakpoint for the hart(s) required.

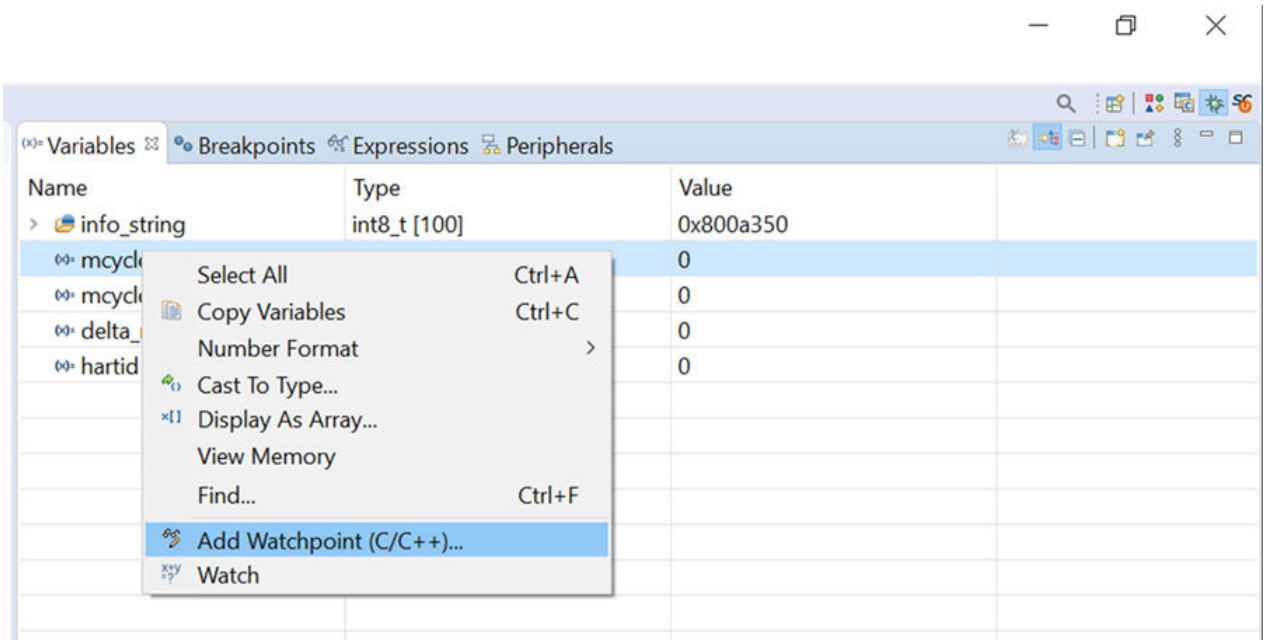
**Figure 1-21.** Filter Breakpoints



### 1.3.2.6 Setting Watchpoints (Ask a Question)

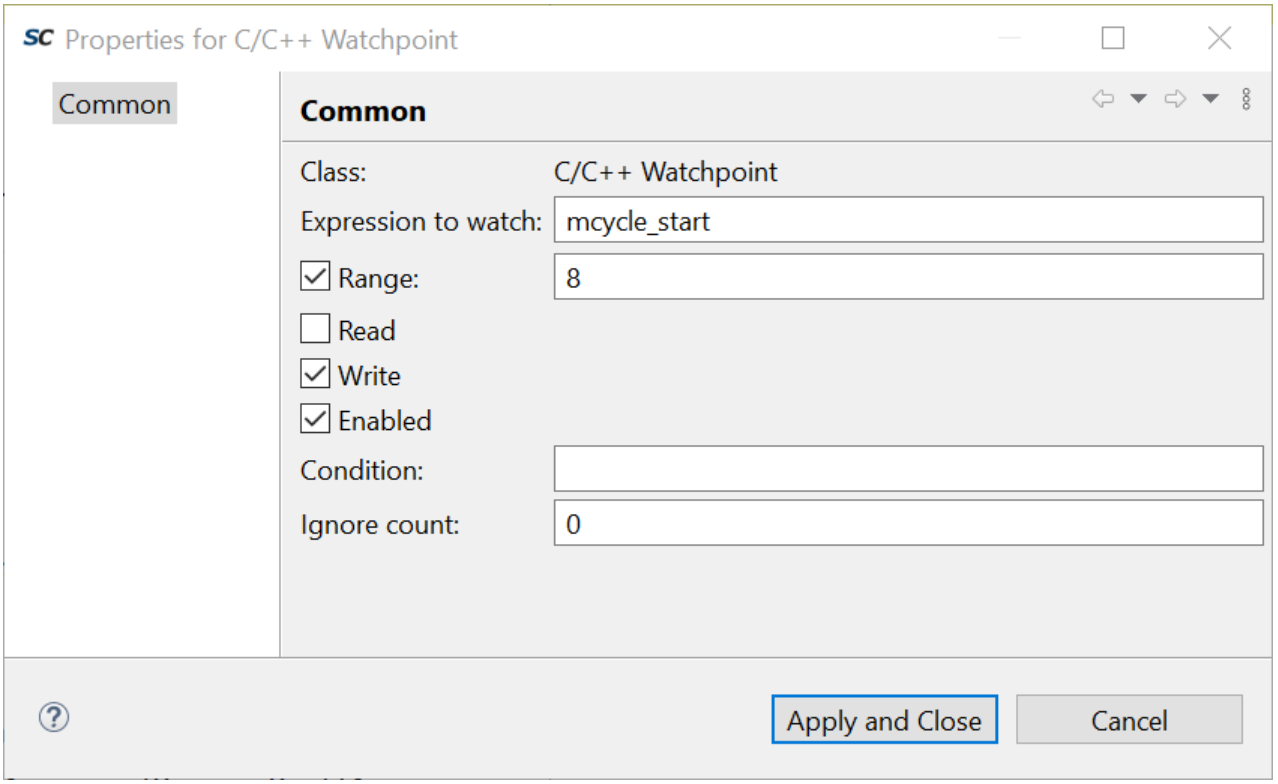
Watchpoints can be set on variables while running a debug session. To set a Watchpoint, open the **Variables** window, right click on the variable and select **Add Watchpoint (C/C++)...**.

**Figure 1-22.** Add Watchpoint



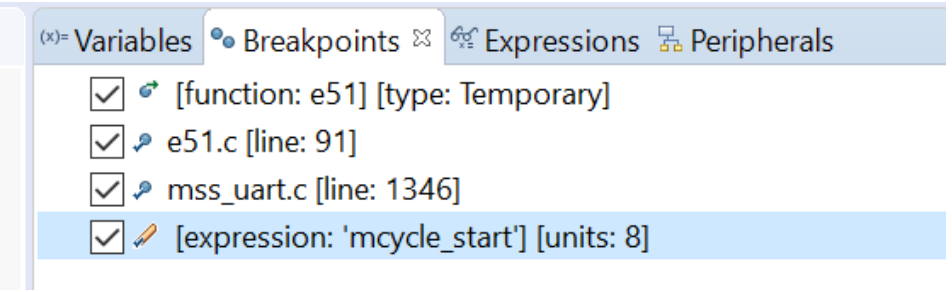

This Watchpoint can be configured using the **Properties for C/C++ Watchpoint** window.

**Figure 1-23.** Watchpoint Properties Window



The added Watchpoint appears in the **Breakpoints** window as an expression.

**Figure 1-24.** Breakpoints Window with Watchpoint




### 1.3.3 Renode™ (Ask a Question)

Renode is an open-source software development framework with commercial support from Antmicro that lets you develop, debug and test multi-node device systems reliably, scalably, and effectively.

For more information, see: https://renode.io/ and https://github.com/renode/renode.

## 1.4 FlashPro Express (Ask a Question)

FlashPro Express is the software tool for programming PolarFire SoC using the FlashPro Programmer hardware.

For the latest version of FlashPro Express User Guide, see www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/fpga/libero-software-later-versions#Documentation.

## 1.5 RISC-V GCC Bare Metal (Ask a Question)

The GNU Compiler Collection (GCC) is a compiler system produced by the GNU Project supporting various programming languages. GCC is a key component of the GNU toolchain and the standard compiler for most projects related to GNU and Linux, including the Linux kernel. The Free Software Foundation (FSF) distributes GCC under the GNU General Public License (GNU GPL). GCC has played an important role in the growth of free software, as both a tool and an example.

SoftConsole is shipped with Bare Metal riscv-gcc-toolchain with `newlib` and `newlib.nano` for 40 abi/arch multilib combinations that allow a single toolchain to target various different target architectures (see SoftConsole Release Notes for more details).

For GCC documentation, see Using the GNU Compiler Collection.

For GCC RISC-V specific options, see RISC-V Options.

For more details about march, mabi, and mtune arguments, see www.sifive.com/blog/all-aboard-part-1-compiler-args and The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2.

## 1.6 RISC-V Linux Toolchain (Ask a Question)

The RISC-V GNU Linux-ELF/glibc toolchain is used by the Linux build tools (Yocto and Buildroot) to build a Linux image.

The source used for the toolchain is available on GitHub at the following location: github.com/riscv/riscv-gnu-toolchain.

## 1.7 Yocto (Ask a Question)

Yocto is an open-source development build environment for Linux. It can be used to customize a Linux image for embedded or IoT application deployment. The Yocto framework is modular in nature and designed as a software stack with layers managing different tasks and functions. The

BSP layer provides machine configurations. The distro layer includes the top level polices for a distribution. The OpenEmbedded Build system (referred to as "the build system") is the build system used by Yocto based on "Poky". BitBake is used by the build system for image generation.

The Microchip Yocto BSP can be found at the following location: mi-v-ecosystem.github.io/redirects/repo-meta-polarfire-soc-yocto-bsp. It contains predefined recipes to build different PolarFire SoC targets and build images with different bundles of tools. For a full list of available builds, see the readme or the console output after successfully configuring a build.

**Important:** Microchip peripheral drivers are currently added as patches with the Microchip PCIe driver being up streamed to the Linux kernel version 5.8, it is planned to upstream remaining drivers for other system peripherals in later releases.

Yocto builds supported on Linux and Yocto are not currently supported by the Windows Subsystem for Linux.

More information about Yocto is available on the Yocto project website.

**Related Links**

https://www.yoctoproject.org/

https://docs.yoctoproject.org/current/ref-manual/index.html

## 1.8 Buildroot (Ask a Question)

Buildroot is a tool that simplifies and automates the process of building a complete Linux system for an embedded system, using cross-compilation. It is able to generate a cross-compilation toolchain, a root filesystem, a Linux kernel image, and a bootloader for the target.

The Microchip PolarFire SoC Buildroot External is available at the following location: mi-v-ecosystem.github.io/redirects/repo-polarfire-soc-buildroot-sdk. It contains configured builds for different PolarFire SoC targets. For a full list of available builds, see the readme.

**Important:** Microchip peripheral drivers are currently added as patches with the Microchip PCIe driver being up streamed as to the Linux kernel version 5.8, it is planned to upstream remaining drivers for other system peripherals in later releases.

Buildroot is supported on Linux and supported by the Windows Subsystem for Linux.

More information about Buildroot is available on the Buildroot website.

**Related Links**

https://buildroot.org/

https://buildroot.org/downloads/manual/manual.html#:~:text=Buildroot%20is%20a%20tool%20that,a%20bootloader%20for%20your%20target

## 1.9 SmartDebug (Ask a Question)

SmartDebug is a tool that enables verification and troubleshooting at the hardware level. It provides access to sNVM, SRAM, transceiver, uPROM, and fabric probe capabilities.

SmartDebug accesses the built-in probe points through the Active Probe and Live Probe features that enable designers to check the state of inputs and outputs in real-time without modification of the design.

SmartDebug can be run in the following modes:

- Integrated mode from the Libero Design Flow
- Standalone mode
- Demo mode (without target hardware connected)

For the latest version of SmartDebug User Guides, see www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/fpga/libero-software-later-versions#Documentation.

## 1.10 Identify (Ask a Question)

Identify is a tool to find and correct functional design bugs by probing internal signals of the design directly from the FPGA at the system speed.

For more information, see Identify ME webpage.

# 2. Software Stack (Ask a Question)

RISC-V is a large ecosystem with a variety of compilers, software libraries, examples, and tools available for application development. This section outlines the open source RISC-V libraries available, along with Microchip applications and examples to aid development for PolarFire SoC.

## 2.1 RISC-V Libraries (Ask a Question)

Standard libraries provide generic type definitions, functions, and macros for tasks that will be undertaken on a system. This section provides information on the Newlib C standard library implementation and the GNU Binary Utilities toolset bundled with SoftConsole.

### 2.1.1 Newlib (Ask a Question)

Newlib is a C standard library implementation intended for use on embedded systems. It is a conglomeration of several library parts, all under free software licenses that make them easily usable for embedded products.

SoftConsole's RISC-V GCC toolchain and its multilibs come with pre-compiled and ready-to-use “newlib” and “newlib-nano” C library.

Microchip’s toolchains come with generic basic Newlib implementation.

More information about Newlib and FAQ is available on the Sourceware website.

**Related Links**

https://www.sourceware.org/newlib/faq.html

https://www.sourceware.org/newlib/docs.html

### 2.1.2 Binutils (Ask a Question)

The GNU Binary Utilities, or binutils, are a set of programming tools for creating and managing binary programs, object files, libraries, profile data, and assembly source code.

SoftConsole comes bundled with a ready-to-use Bare Metal binutils. The SoftConsole Release Notes show some use cases of how to use nm and objcopy; however, most of the tools are used and invoked automatically by the IDE. These tools are generic and can be used to target the 32-bit Mi-V RISC-V cores and the 64-bit PolarFire SoC targets.

**Table 2-1.** GNU Binary Utilities

| Tool Name | What it does | Documentation |
|---|---|---|
| as | Assembler | sourceware.org/binutils/docs/as/ |
| ld | Linker | sourceware.org/binutils/docs/ld/ |
| gprof | Profiler | sourceware.org/binutils/docs/gprof/ |
| addr2line | Convert address to file and line | sourceware.org/binutils/docs/binutils/addr2line.html |
| ar | Create, modify, and extract from archives | sourceware.org/binutils/docs/binutils/ar.html |
| c++filt | Demangling filter for C++ symbols | sourceware.org/binutils/docs/binutils/c_002b_002bfilt.html |
| nm | List symbols in object files | sourceware.org/binutils/docs/binutils/nm.html |
| objcopy | Copy object files, possibly making changes | sourceware.org/binutils/docs/binutils/objcopy.html |
| objdump | Dump information about object files | sourceware.org/binutils/docs/binutils/objdump.html |
| ranlib | Generate indices for archives (for compatibility, same as ar -s) | sourceware.org/binutils/docs/binutils/ranlib.html |
| readelf | Display content of ELF files | sourceware.org/binutils/docs/binutils/readelf.html |
| size | List total and section sizes | sourceware.org/binutils/docs/binutils/size.html |

**Table 2-1.** GNU Binary Utilities (continued)

| Tool Name | What it does | Documentation |
|---|---|---|
| strings | List printable strings | sourceware.org/binutils/docs/binutils/strings.html |
| strip | Remove symbols from an object file | sourceware.org/binutils/docs/binutils/strip.html |

## 2.2 Hart Software Services (HSS) (Ask a Question)

Hart Software Services, commonly referred to as "HSS", is a collection of services that run on the E51 monitor core. HSS is used for the following:

- Program memory using USB mass storage or YMODEM transfer.
- Copy a program (Linux or Bare Metal) from a non-volatile storage (for example, eMMC or SD card) to the LIM or DDR.
- Create a payload containing multiple applications to be booted and run.
- Pass messages between cores in the MSS.

### Operation

The HSS uses Bare Metal drivers to initialize the system, which are found in the PolarFire SoC Bare Metal Library. It also relies on XML generated by the PolarFire SoC MSS Configurator to configure the system on boot.

The HSS comprises of the following:

- A superloop monitor running on the E51 processor, which receives requests from the individual U54 application processors to perform certain services on their behalf.
- A Machine-Mode software interrupt trap handler, which allows the E51 to send messages to the U54s, and requests them to perform certain functions for it related to rebooting the U54.

### HSS as a ZSBL

The HSS can function as a Zero Stage Boot Loader (ZSBL) to boot Linux. In this case, the HSS loads U-Boot acting as a ZSBL with U-Boot subsequently loading an OS. U-Boot is a First Stage Boot Loader (FSBL) and a Second Stage Boot Loader (SSBL).

### HSS as a FSBL

The HSS can be used to boot Linux directly like the Berkeley Boot Loader (BBL) acting as an FSBL and SSBL.

### Licenses

This software is released under an MIT license. It also uses other open source tools. RISC-V OpenSBI is released under a BSD-2-Clause and FastLZ compression is released under an MIT license. More information on licensing can be found at: mi-v-ecosystem.github.io/redirects/repo-hart-software-services.

### Building

The HSS can be built as a standalone image. The source is published on GitHub and build instructions for different targets can be found in its readme: mi-v-ecosystem.github.io/redirects/repo-hart-software-services.

### Releases

The HSS GitHub repository is the most up-to-date location to retrieve the source files and build instructions for the HSS.

## 2.3 Bare Metal Library (Ask a Question)

The PolarFire SoC Bare Metal Library contains the most recent version of the PolarFire SoC HAL source code with a pre-populated platform folder for a PolarFire SoC Bare Metal project with all drivers. It also contains Bare Metal examples for each driver available for PolarFire SoC. These

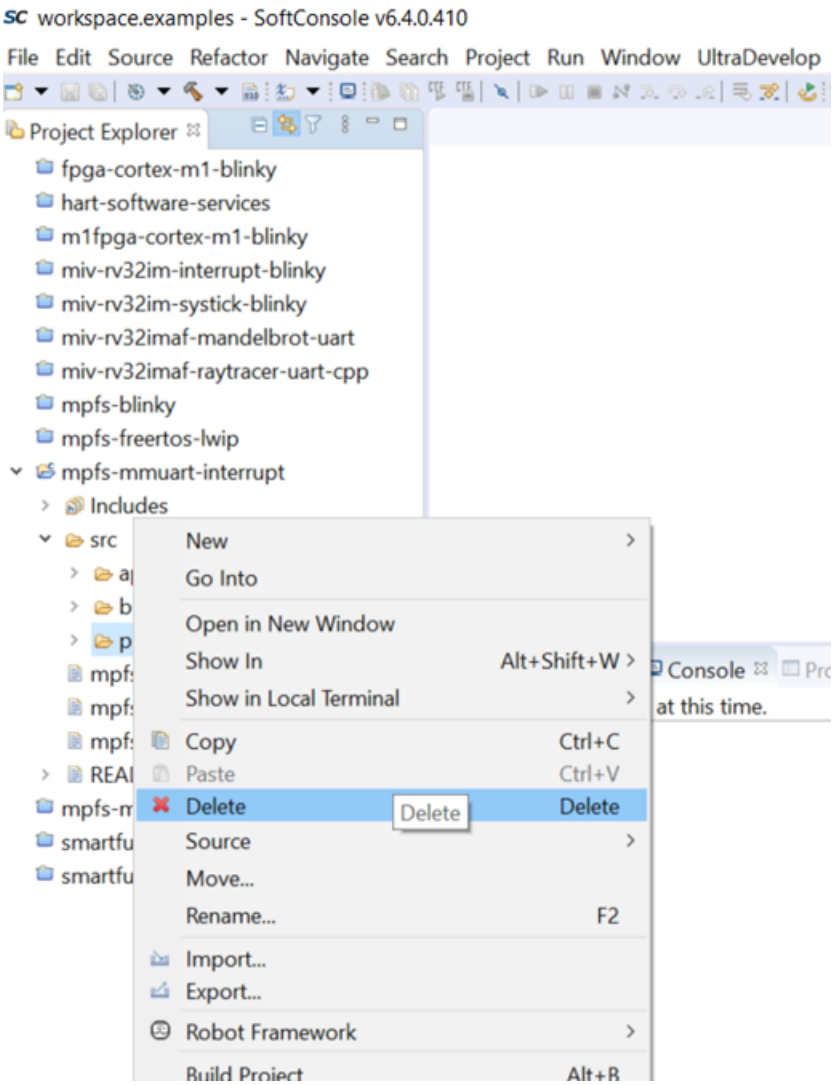
examples show how to use different functions available in the drivers and how they are configured for PolarFire SoC.

To use the Bare Metal Library examples, follow the instructions in the `polarfire-soc-bare-metal-library/examples readme.md` file available at: mi-v-ecosystem.github.io/redirects/repo-polarfire-soc-bare-metal-library

To use the pre-populated platform folder in a Bare Metal project, follow these steps:

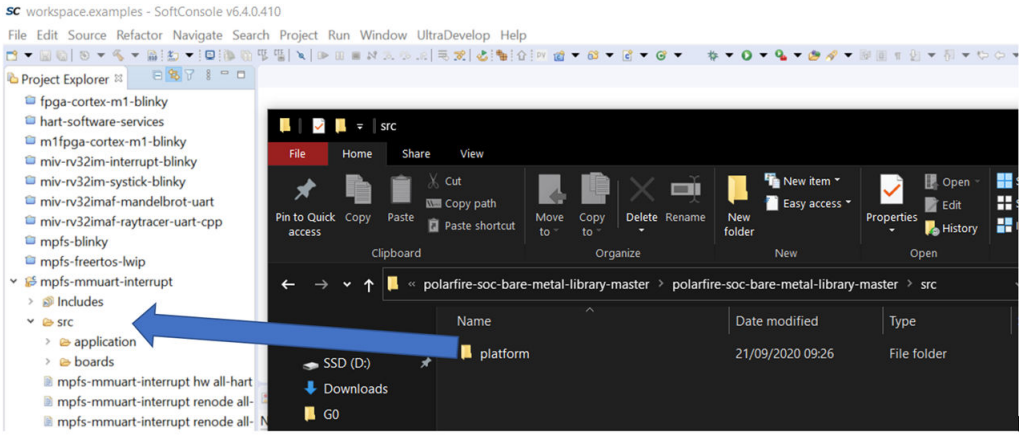
1. Download the Bare Metal Library repository and extract it.
2. Delete the existing platform folder in the project (back up any changes such as, Linker script updates).

**Figure 2-1.** Removing Platform Folder



3. Copy the platform folder extracted from the Bare Metal Library repository into the SoftConsole project and re-implement any changes that were made.

**Figure 2-2.** Adding Updated Platform Folder




The only folder that might be modified by the user is the `platform/config` folder. The drivers, hal, and mpfs_hal do not need user modification.

mpfs_hal contains the part of the HAL specific to PolarFire SoC. It contains startup code, MSS register descriptions, and performs DDR training. The content of this folder is not intended to be modified. It also contains the code for interrupt and exception handling, and hardware access methods.

## 2.4 Linker Scripts (Ask a Question)

The main purpose of the Linker script is to describe how the sections in the input files must be mapped into the output file, and to control the memory layout of the output file.

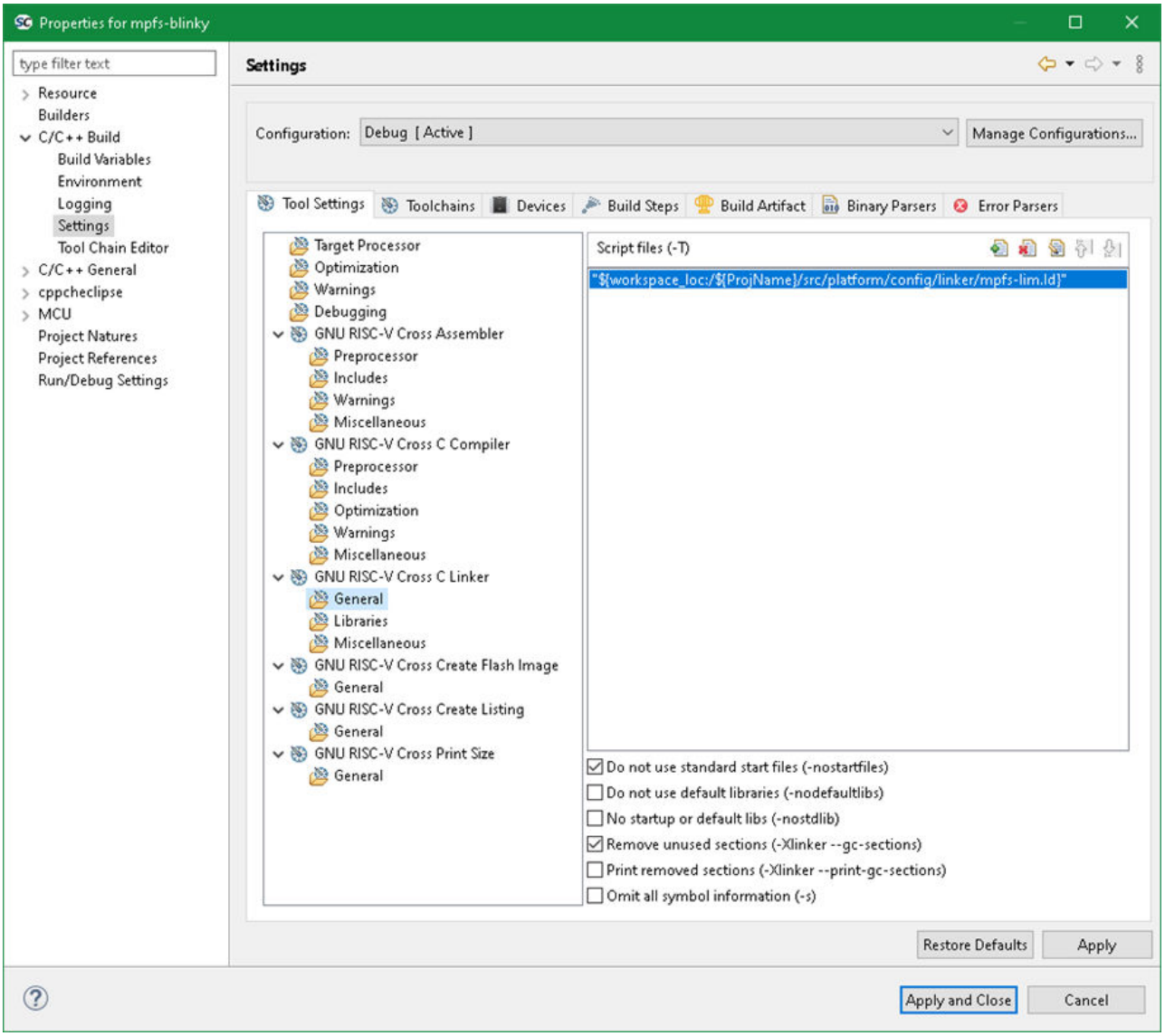
Each SoftConsole project comes with at least one Linker script. Specify the memory location where the application will be deployed. The sample Linker scripts provided include:

- mpfs-ddr-e51
- mpfs-dtim
- mpfs-envm
- mpfs-lim
- mpfs-lim-lma-scratchpad-vma

To switch between them, open **Settings** and navigate to the **Script files**:

**Project's Properties > C/C++ Build > Settings > Tool Settings > GNU RISC-V Cross C Linker > General > Script files**

**Figure 2-3.** Tool Settings




Microchip recommends users to use the supplied Linker scripts (located in the `src/platform/config/linker` folder) and use these as a base script for their custom Linker scripts (when the supplied Linker scripts are not sufficient).

For the Linker script manual, see: sourceware.org/binutils/docs/ld/Scripts.html

**Related Links**

https://www.sifive.com/blog/all-aboard-part-2-relocations

https://www.sifive.com/blog/all-aboard-part-3-linker-relaxation-in-riscv-toolchain

## 2.5 Linux Images (Ask a Question)

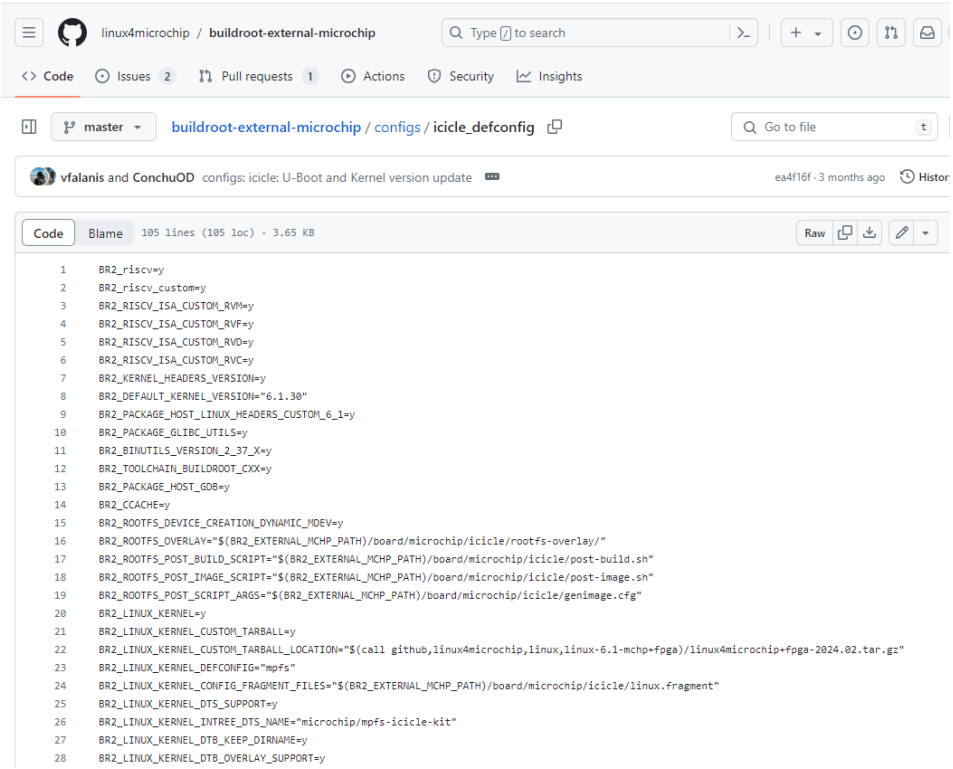
The current Linux kernel version (at the time of publishing this document) is 5.6.16. The next update is for moving to Linux kernel version 5.8. Beyond version 5.8, it is planned to use the latest long-term support kernel in all builds. Information on long-term support for kernels is available on: www.kernel.org/.

In Yocto, the kernel version used in the build is specified in the `recipes-kernel/linux/*.bb` file as shown in the following figure.

**Figure 2-4.** Example of Yocto Linux Kernel Version


For Buildroot, the specific kernel version used can be found in `defconfig`, the following figure shows an example `defconfig`, which specifies the Linux kernel version used for the Icicle kit.

**Figure 2-5.** Example of Buildroot Linux Kernel Version



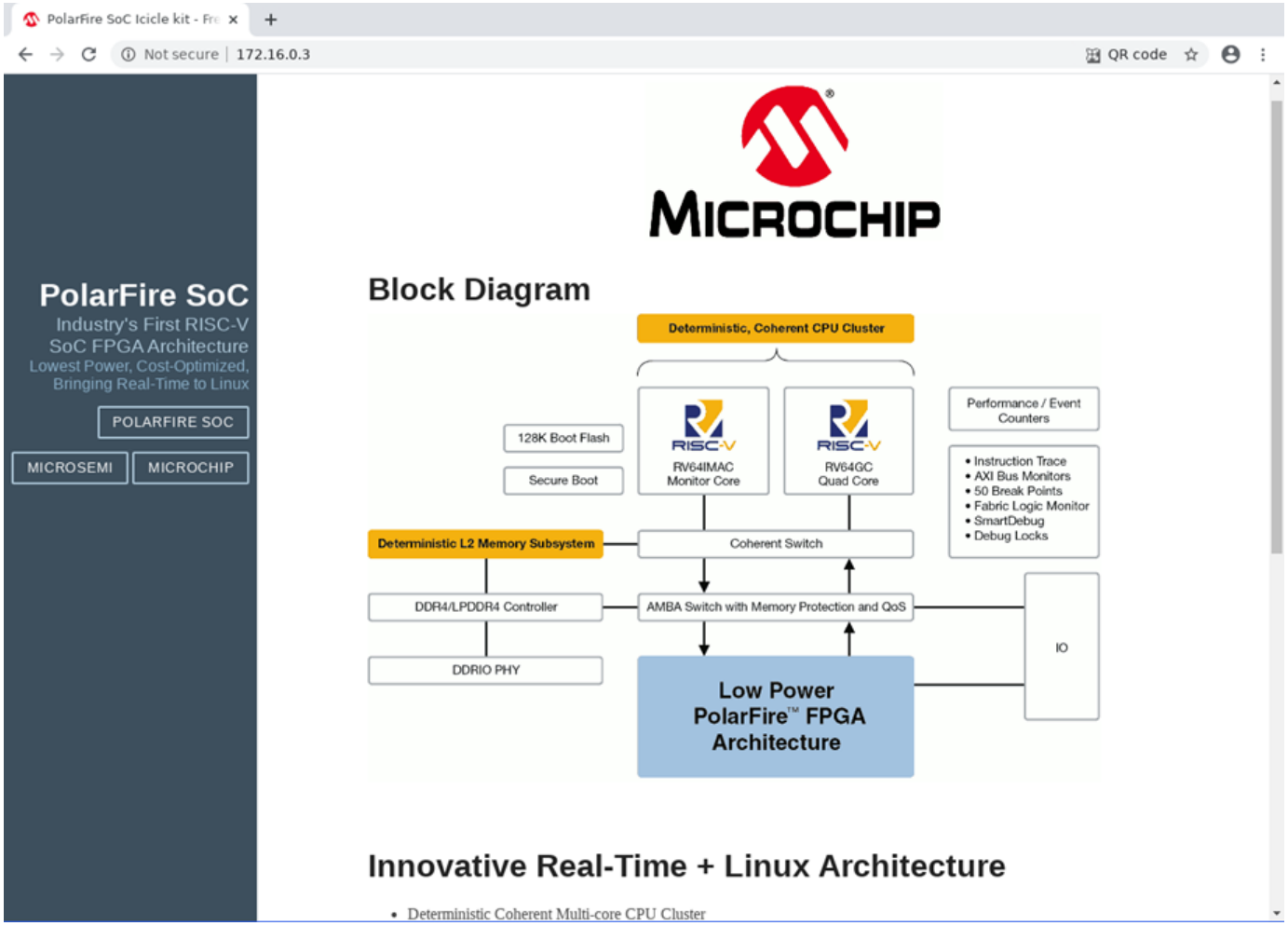

## 2.6 FreeRTOS™ (Ask a Question)

FreeRTOS is a real-time operating system kernel for embedded devices that has been ported to many microcontroller platforms. It is distributed under the MIT license.

SoftConsole is shipped with a bundled FreeRTOS example, see the `mpfs-freertos-lwip` example. The example can target the bundled Renode emulation and users can access the webserver running on it (See the *mpfs-freertos-lwip's* readme).

The following figure shows a webserver running on top of FreeRTOS and lwip.

**Figure 2-6.** Webserver Running on Top of FreeRTOS and lwip




The latest version of FreeRTOS targeting PolarFire SoC hardware is available from the Bare Metal Examples repository in the `examples/mss/mss-ethernet-mac/` directory.

More information is available on the FreeRTOS website.

**Related Links**

https://www.freertos.org/a00104.html#getting-started

https://www.freertos.org/Documentation/RTOS_book.html

## 2.7 Third Party Tools (Ask a Question)

The third party tool Renode is used for emulating RISC-V subsystem.

# 3. Application Development (Ask a Question)

PolarFire SoC supports Bare Metal, Linux, and RTOS. This section describes the device boot process, boot modes, and development flow to build user applications for these types of embedded systems. For example, the following can be executed on the application cores.

- Bare Metal applications
- Linux user applications
- RTOS
- Combination of the above (AMP)

Bare Metal applications for PolarFire SoC devices can be developed using SoftConsole. The PolarFire SoC Bare Metal firmware drivers and source files for Linux user application development are available on the PolarFire SoC GitHub.

PolarFire SoC MSS comprises of one E51 monitor core and four U54 application cores. The E51 core executes the Hart System Services (HSS), which configures the MSS and responds to runtime events. The U54 cores execute any of the following:

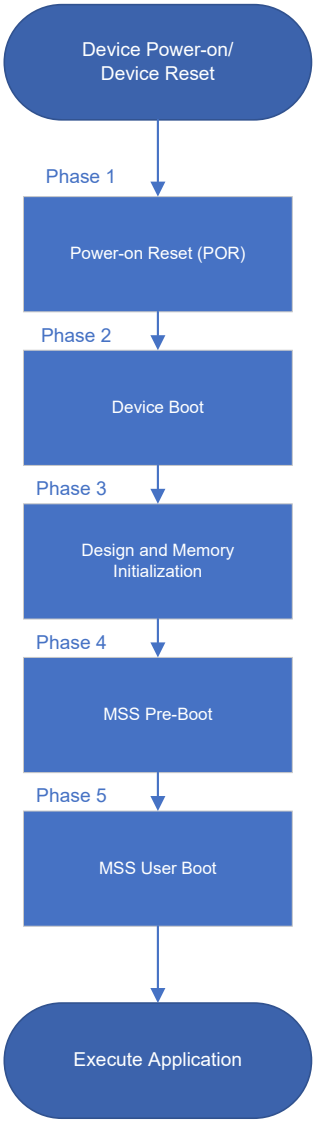
- Bare Metal user applications
- Operating Systems

## 3.1 Device Boot and Configuration Process (Ask a Question)

The boot-up sequence starts when the PolarFire SoC FPGA is powered-up or the device is reset. It ends when the processor is ready to execute user applications. The booting sequence runs through several stages before it begins execution of user application code. A set of operations are performed during the boot-up process that includes Power-on Reset of the hardware, peripheral initialization, memory initialization, and loading a user-defined application from non-volatile memory to volatile memory for execution.

The following figure shows the different phases of the boot-up sequence.

**Figure 3-1.** Boot-Up Sequence




For more information about the booting process, see PolarFire FPGA and PolarFire SoC FPGA Power-Up and Resets User Guide.

PolarFire SoC MSS supports the following boot modes:

- Boot Mode 0—boot mode 0 is used by blank devices or when debugging embedded software, where code must not be executed on power-up.
- Boot Mode 1—boot mode 1 is used where the MSS harts start executing non-secured code from eNVM on power-up.
- Boot Mode 2—boot mode 2 is intended for implementing user-defined secure boot authentication.

- Boot Mode 3—boot mode 3 implements Microchip supplied factory secure boot authentication of the eNVM content. It uses the Elliptic Curve Digital Signature Algorithm (ECDSA) to authenticate the signature of a Secure Boot Image Certificate (SBIC) as part of booting the system.

## 3.2 Boot Mode 0-Idle Boot (Ask a Question)

Boot mode 0 puts the MSS in a mode where all harts execute a loop waiting for the debugger to connect through JTAG or for the device to be programmed.

## 3.3 Boot Mode 1-Direct Boot from eNVM (Ask a Question)

In this mode, the MSS executes from a specified eNVM address without authentication. It is the fastest boot option, but there is no authentication of the code image. Boot Mode 1 includes the following steps:

1. The user application image needs to be programmed into eNVM using SoftConsole and the Boot mode is set.
2. If the eNVM content is a boot loader, it fetches the final user application from non-volatile storage and loads it to the desired memory location specific by the application, the harts then execute the application.

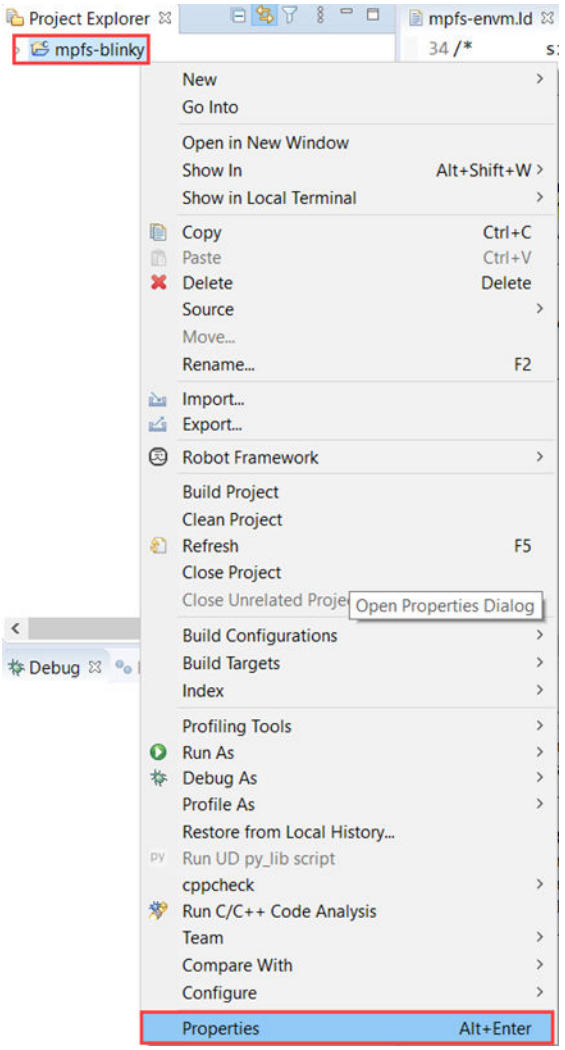
Boot vector addresses for all five processors are absolute addresses in eNVM.

### 3.3.1 Programming the eNVM (Ask a Question)

Launch SoftConsole and create an application project. Ensure that the SoftConsole application project contains latest **mpfs_hal** and firmware drivers from GitHub. For illustration purposes, **mpfs_blinky** is used as an example application project. To configure the project's build tool settings, follow these steps:

1. Right click **mpfs_blinky** and select **Properties** as shown in the following figure.

**Figure 3-2.** Properties for mpfs-blinky



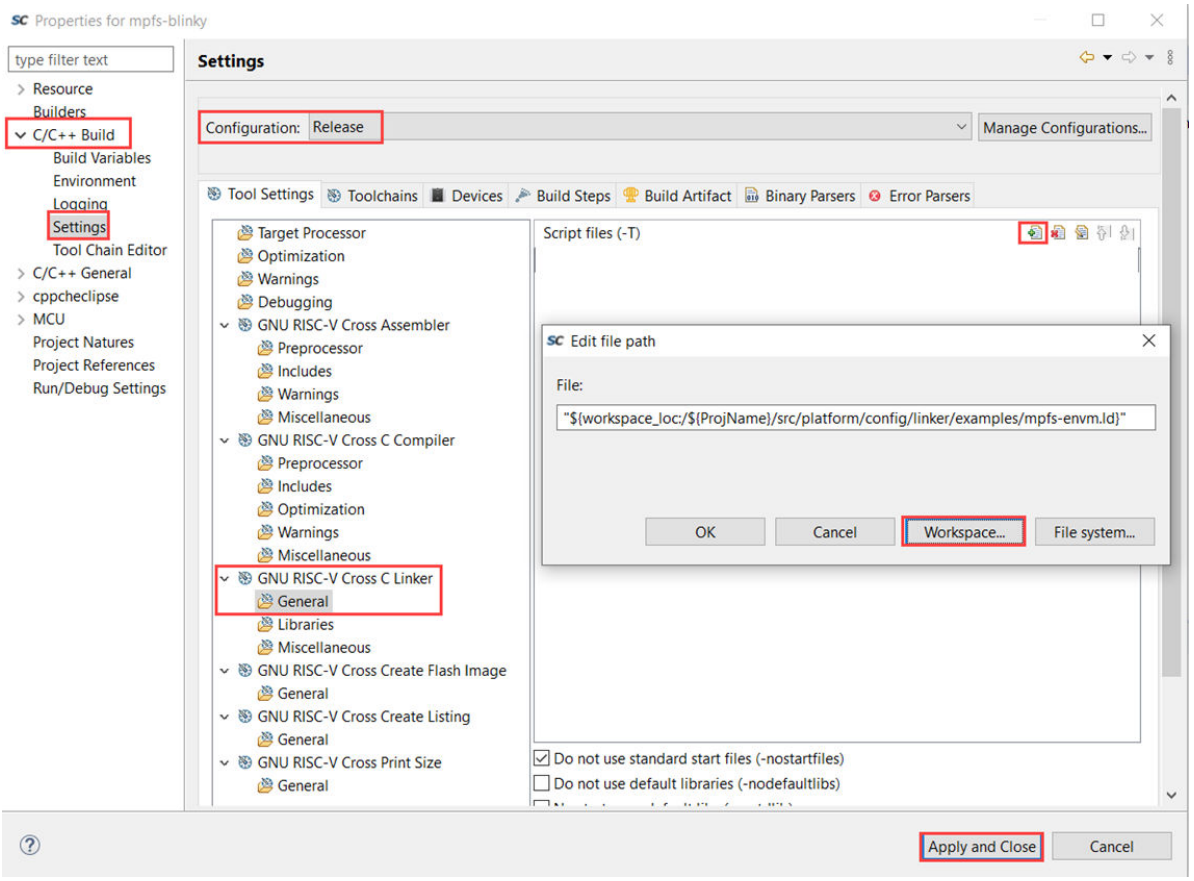

The **Properties** window appears.

2. Expand **C/C++ Build**, and select **Settings**.
3. Set the **Configuration** to **Release**.
4. Expand **GNU RISC-V Cross C Linker**, select **General** and perform the following actions to select the appropriate Linker script:
   a. Click **Add...**.
   b. Select **Workspace** on the **Add File path** window.
5. In the **File Selection** tab, expand the **mpfs-blinky** and browse to: `mpfs-blinky > src > platform >Config > linker > examples > mpfs-envm.ld` file. Then, click **OK**.

   The Linker is a script file which provides the information about the memory from where the code must be executed and how that memory must be used for heap and stack.

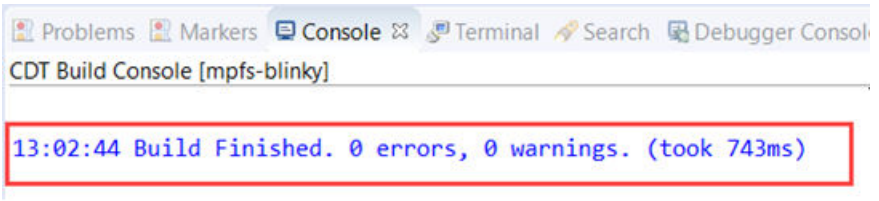
   For Release mode, the Linker script `mpfs-envm.ld` is selected to build the application that executes the code from eNVM with the stack and heap in LIM. Other Linker script files are also available to execute the code out of LIM (`mpfs-lim.ld`), DDR memory (`mpfs-ddr-e51.ld`), and Data Tightly Integrated Memory (`mpfs-dtim.ld`).

**Figure 3-3.** mpfs-blinky Release Window



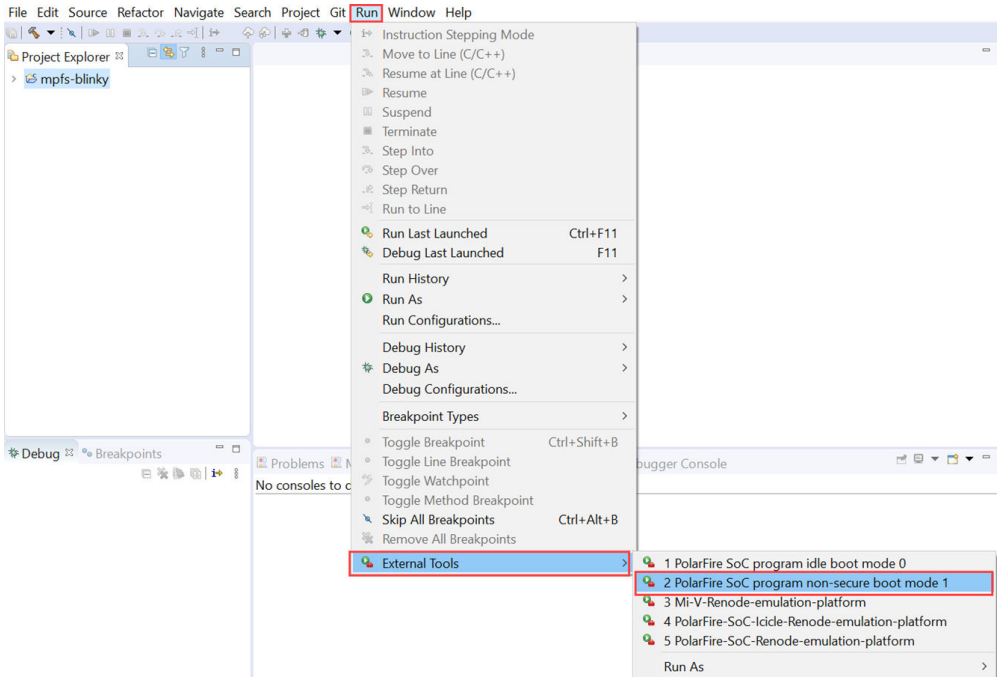

6. Click **Apply and Close**.
7. Select **Project > Clean** to clean the project.
   Now, the project build settings are completed and ready for building.
8. Select **Project > Build All**.
   The project is built successfully as shown in the following figure.

**Figure 3-4.** Project Successful Message



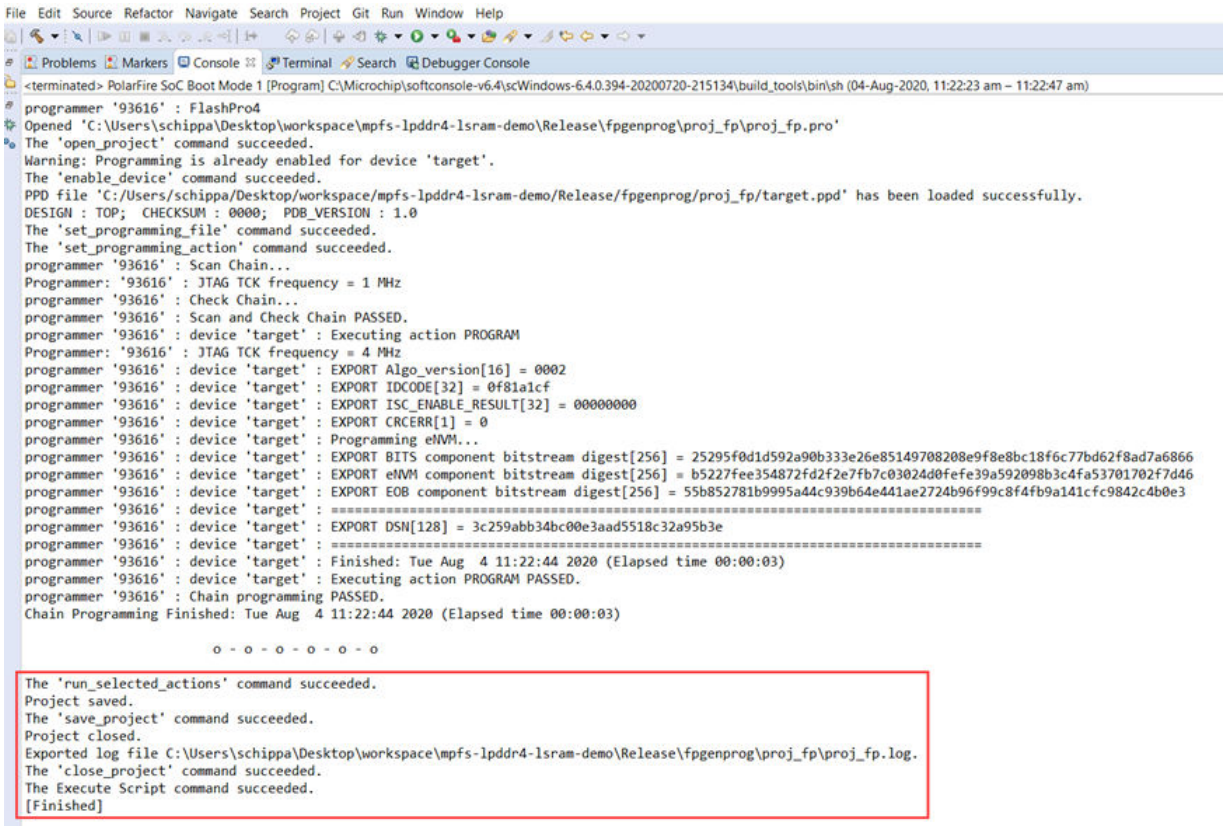

9. The `mpfs-blinky.elf` file is generated in the Release folder. This ELF file is programmed to the eNVM storage using SoftConsole so that at device power-up the MSS executes the application from eNVM.
   To program the Boot mode settings and eNVM using SoftConsole, select the desired project and click **Run > External Tools > PolarFire SoC program non-secure boot mode 1** as shown in the following figure.
   **Note:** This step requires Libero SoC or the Program/Debug tool to be installed on the host PC. It also requires pre-configuration in SoftConsole. For information on pre-configuration and installation, see the `readme.txt file` in (SoftConsole install)/extras/mpfs.

**Figure 3-5.** PolarFire SoC Boot Mode 1




This sets the Boot mode to 1 and programs the application to eNVM as shown in the following figure. After power-cycling the board, the application gets executed from eNVM.

**Figure 3-6.** Programming the Application

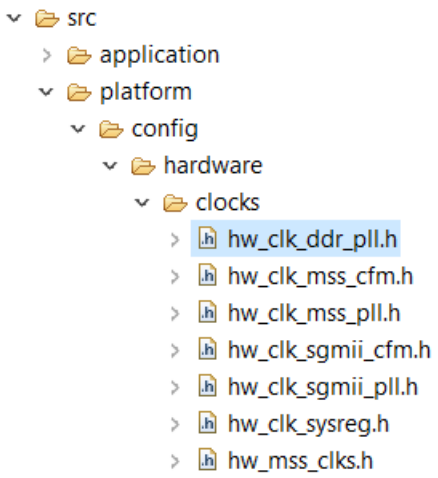


**Important:** For more information on all the boot modes and their sequences, see PolarFire SoC MSS Technical Reference Manual.

## 3.4 Clock Management (Ask a Question)

The clock configuration made in the **Clocks** tab of the PolarFire SoC MSS Configurator is stored in the XML file. The SoftConsole build process converts the XML file into hardware configuration files that include the clock configuration as shown in the following figure. If any changes to the clock configuration are required, these changes need to be made in the PolarFire SoC MSS Configurator and the updated XML file needs to be imported into the SoftConsole project. The system start-up code uses these configuration files to configure the PLLs and clocking related system registers.

**Figure 3-7.** Clocks Folder




## 3.5 Physical Memory Protection (PMP) (Ask a Question)

### 3.5.1 Using the PMPs in Bare Metal (Ask a Question)

To support secure execution of application code, it is required to limit the physical addresses accessible by the software running on a Hardware Thread (Hart). The access to physical addresses can be restricted using the PMP unit in each hart. The PMP defines a finite number of regions that can be individually configured by setting the PMP registers using the user application code. It is applied on harts to allow physical memory access privileges (read, write, and execute). A PMP is configured to achieve the following:

- Security of the system is improved as there is no possibility of code injection attacks. The memory of one hart is not accessible to other harts.
- If there is any overflow in the stack usage, PMP detects it.
- It is less expensive to get safety certification for the product.

### 3.5.2 Using the PMPs in Linux (Ask a Question)

When booting Linux using HSS, the PMPs are automatically configured by the HSS for the system configuration based on the payload created.

**Important:** Configuring the PMPs using Libero is currently not supported and will be available in the next release.

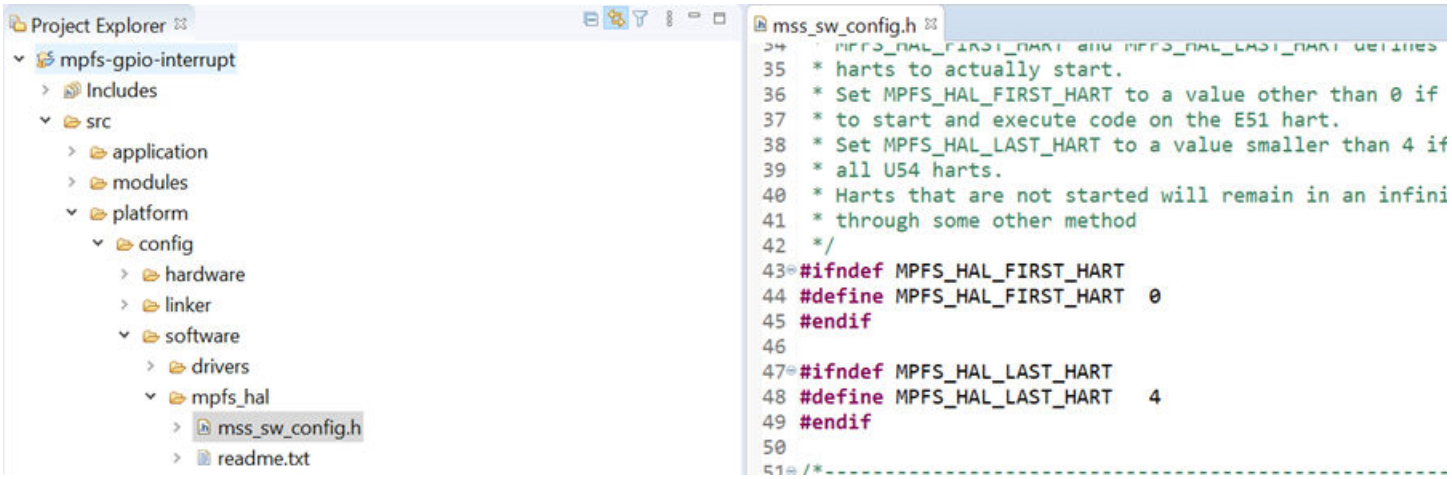
## 3.6 Generating Boot Images (Ask a Question)

This section outlines the software configuration for Bare Metal project(s) that targets different harts for user applications. The steps to configure a Bare Metal application as the sole application to run on the system are provided along with how to configure multiple independent Bare Metal applications. This section also outlines the steps required to configure an application to be executed directly from eNVM and how to configure an application to be stored in the external memory and load it using the HSS. The process of programming the eNVM is also introduced.

### 3.6.1 Targeting Harts (Ask a Question)

The MPFS HAL (PolarFire SoC HAL) can be used to target multiple or single cores in a Bare Metal application. In the Bare Metal application in **src > platform > config > software > mpfs_hal the mss_sw_config.h** file can be used to target harts.

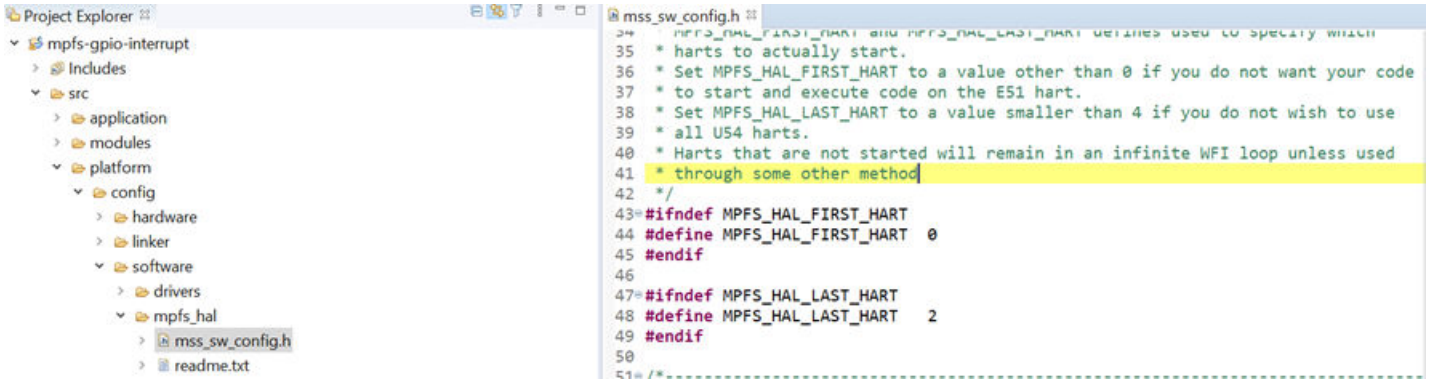
**Figure 3-8.** Targeting Harts




The `MPFS_HAL_FIRST_HART` define selects the hart that will boot up and configure the system. The `MPFS_HAL_LAST_HART` define selects the hart that will be the last to start up. In the preceding image, the e51 is the hart selected to start up first and will be used to wake all of the U54 harts from WFI (Wait For Interrupt) mode.

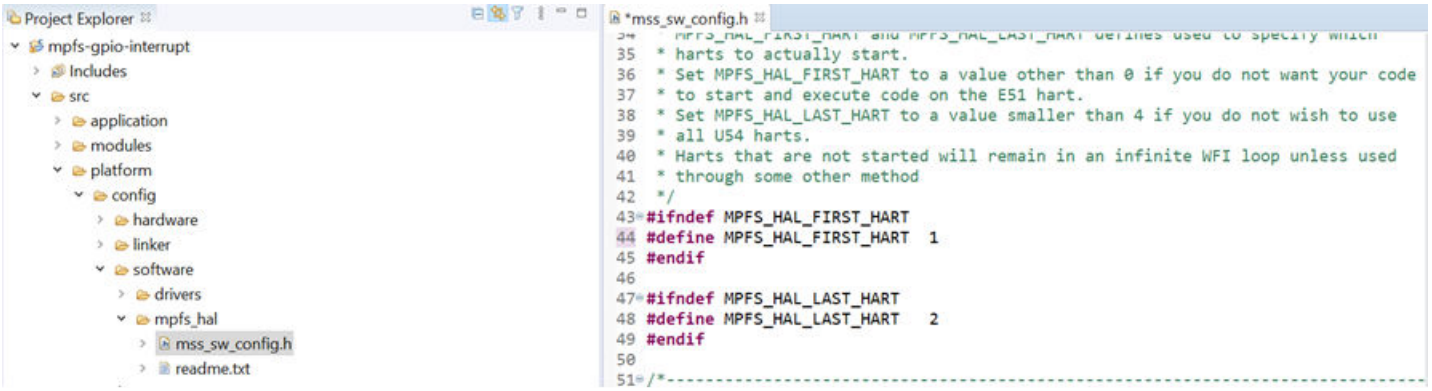
If the system was configured as shown in the following figure, the e51 is still the main hart but only the U54_1 and U54_2 are taken out of the WFI mode (the e51 is hart 0).

**Figure 3-9.** MPFS_HAL_LAST_HART Configuration




If the system is configured as shown in the following figure, the e51 is held in WFI mode and the U54_1 becomes the main hart to wake the other U54 harts from WFI mode, in this case only the U54_2 is taken out of the WFI mode.

**Figure 3-10.** MPFS_HAL_FIRST_HART and MPFS_HAL_LAST_HART Configuration




The projects must be built targeting the memory they will be executed from (for example, DDR) and not in the location they will be stored in (for example, eMMC). When built in SoftConsole, the resulting files created using the HSS tools are called a payload. The HSS readme contains information on the steps for creating a payload, and the tools required are stored in tools folder of the repository. This payload can then be programmed to the intended memory (for example, SD card or eMMC) using the HSS. The HSS then boots on the e51 and unpacks the payload into the relevant memory locations they are targeted for and wakes the harts they will run on.

### 3.6.2 Storing a Single Bare Metal Application in an eNVM (Ask a Question)

If a single Bare Metal application targeting one or more harts is to be used and no other application is running on the system, the application can be stored directly in the eNVM, provided it is less than 128 kB, and executed from memory. The e51 can be used as the main hart of the system (MPFS_HAL_FIRST_HART 0) and wakes any of the required U54 harts from WFI.

### 3.6.3 Storing Bare Metal Application(s) to an External Memory (Ask a Question)

If a single Bare Metal application or multiple independent applications are being used in the system and cannot be stored in the eNVM (for example, they are greater than 128 kB), then the HSS must

be used to program the external non-volatile storage with the applications and copy their code to the relevant memory location on boot.

The HSS must be programmed into the eNVM to be executed on boot, it uses a small portion of the LIM for stack and heap when running on the E51. The Bare Metal application(s) can then be developed—they must not target the e51 as this will be running the HSS and must not overlap with the LIM memory locations used by the HSS. The area of LIM used by the HSS can be identified using the `.map` file generated when the HSS is built.

#### 3.6.3.1 Single Bare Metal Application (Ask a Question)

If a single Bare Metal application is created, the first hart must target one of the U54s (for example, MPFS_HAL_FIRST_HART 1) that wakes the other U54 harts in the system. The project must be built in the memory it will be executed from (for example, DDR) and not in the location it will be stored in (for example, eMMC). When built in SoftConsole, the resulting files created using the HSS tools are called payload. The HSS readme contains information on steps for creating a payload, and the tools required are stored in tools folder of the repository. This payload can then be programmed to the memory intended to be used for storing the payload (for example, SD card or eMMC) using the HSS. The HSS then boots on the e51 and unpacks the payload into the relevant memory location it is targeted for and wakes the harts it will run on.

#### 3.6.3.2 Multiple Bare Metal Applications (Ask a Question)

If multiple Bare Metal applications are created, the first hart for each project must target separate U54s and the last hart in the project (that is, the final U54 this project will run on) must not overlap with the first hart of a subsequent project.

See the following example:

Project 1:

```
MPFS_HAL_FIRST_HART 1
MPFS_HAL_LAST_HART 2
```

This project runs on U54_1 and U54_2.

Project 2:

```
MPFS_HAL_FIRST_HART 3
MPFS_HAL_LAST_HART 4
```

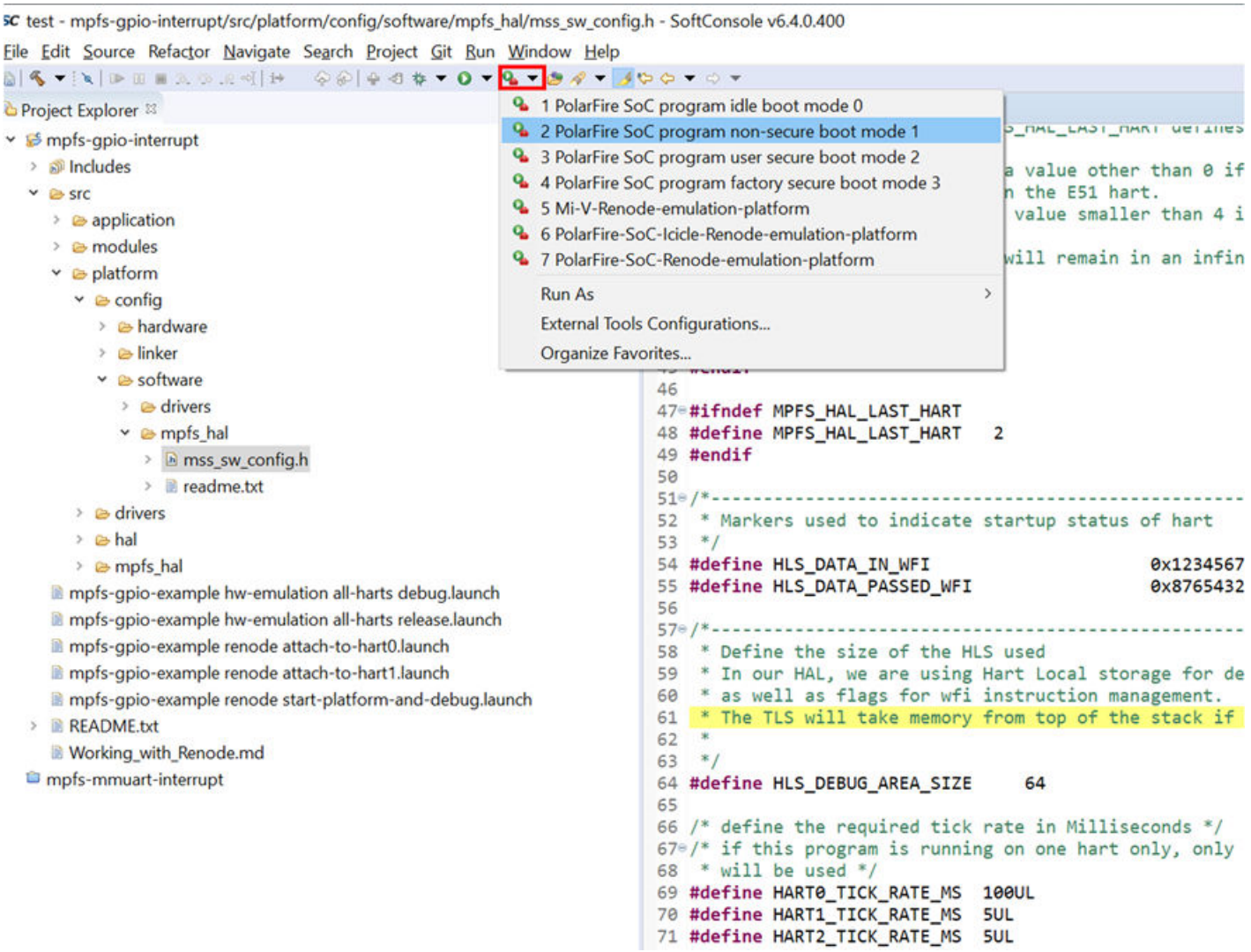
This project runs on U54_3 and U54_4.

For more information, see the Targeting Harts section.

### 3.6.4 Programming the eNVM (Ask a Question)

SoftConsole is capable of programming the eNVM and setting the boot modes for the harts used in the system. If a single Bare Metal project is used, it can be programmed directly into the eNVM. If multiple projects are used (that is, as a payload), the HSS must be programmed into the eNVM. This programming is achieved using an external tool configuration provided with SoftConsole.

**Figure 3-11.** External Tools Setting Boot Modes



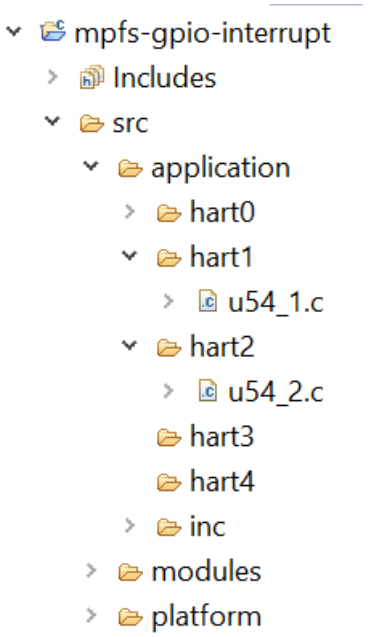

To use this tool, you must have the project to be programmed selected in the project explorer and the build configuration is used to locate programming files. This implies, if a project is built in a Debug configuration, SoftConsole programs it using the files from the Debug folder in the project, whereas if the project is built in a Release configuration, SoftConsole uses the files from the Release folder.

The external tool generates a bitstream containing only the eNVM programming files (that is, it does not overwrite FPGA programming) and sets the Boot mode accordingly.

### 3.6.5 Unused Harts (Ask a Question)

If harts are not going to be used (for example, only U54_1 and U54_2 are used), no main function(s) (for example, `u54_3.c`) need to be provided for unused harts.

**Figure 3-12.** Unused Harts




The mpfs_hal contains weakly linked functions that are used in place of strongly linked functions if no main function is found, which implies, if no u54_3() function is found in the project, the function shown in the following figure "`system_startup.c`" file is used.

The mpfs_hal contains weakly linked functions for all main functions available for each hart. A weakly linked function is used as a fallback or default function if a strongly linked function is not defined. By default, all functions are strongly linked without any modification and if present overrides the weakly linked functions. The mpfs_hal weakly linked functions can be found in `platform/mpfs_hal/system_startup.c`.

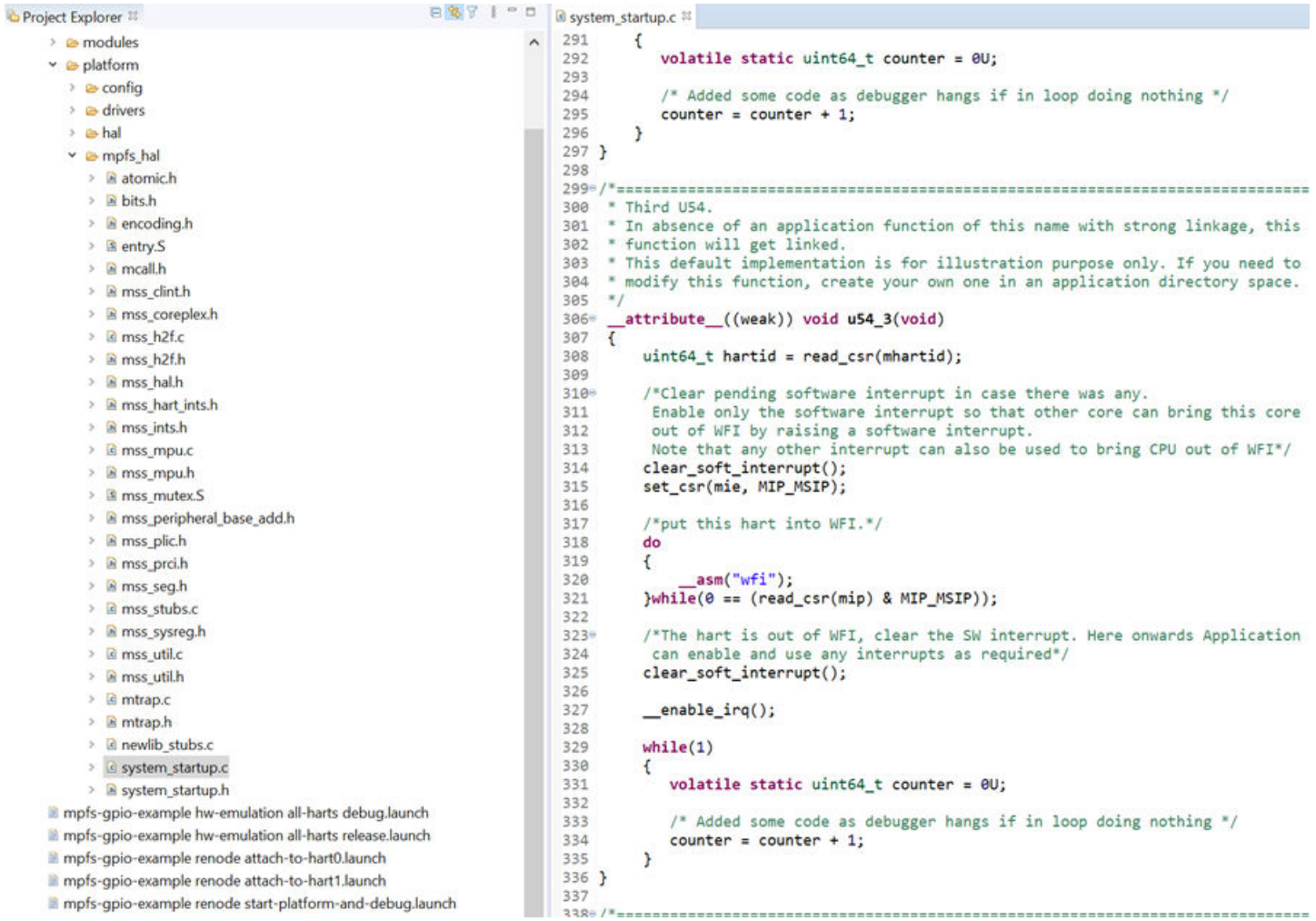
A function is defined as weakly linked by adding the following attribute to its definition: `__attribute__((weak))`.

Consider the following examples for U54_1:

- Weakly linked function: `__attribute__((weak)) void u54_1(void)`
- Strongly linked function: `void u54_1(void)`

**Attention:** If a strongly linked function and a weakly linked function are defined, only the strongly linked function will be included in the build. If two strongly linked functions are defined with the same name a symbol link error occurs at build time.

**Figure 3-13.** system_startup.c File Showing Weakly Linked Functions




This function is entered if the hart is taken out of WFI (if the **MPFS_HAL_LAST_HART** value still includes this hart, it is included in the system) and simply puts the hart back into WFI as no code is found for it to run.

## 3.7 Bare Metal Development (Ask a Question)

The firmware drivers and associated platform specific files are available on GitHub Bare Metal Library. The Bare Metal application(s) can be executed from one of the memories—LIM, eNVM, DDR memory and so on. Linker script files to execute applications from corresponding memories are also available on GitHub Bare Metal Library. If the application size is more than eNVM size (128 kB), it is recommended to store the application in an external flash memory. The HSS that runs on E51 fetches the application from the external memory and buffers it in DDR, it then copies to the address in LIM or DDR that the application runs from and executes the application. The E51 core releases the U54 cores from WFI depending upon the application requirements. The SoftConsole tool provides an environment to develop Bare Metal applications.

### 3.7.1 Single U54 (Ask a Question)

The Bare Metal start-up code (mpfs_hal) initializes the system clocks and external memory. For single U54 Bare Metal development, the E51 wakes the U54 hart from WFI mode by raising the software interrupt. The U54 executes the application tasks. The remaining harts are kept in WFI mode by the default weakly linked functions defined in the start-up code.

### 3.7.2 Multiple U54s (Ask a Question)

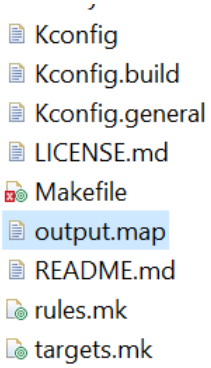
For multiple U54 Bare Metal development, the E51 wakes any combination of U54 harts from WFI mode by raising the software interrupt on each hart. The unused harts are held in the WFI mode. All of the harts run the same start-up code, read their hart ID, and enter WFI. When the U54s receive a software interrupt from E51, they exit the WFI mode and execute their application(s). The U54 harts can execute the same application or different individual applications depending upon application requirements. The HSS provides the necessary functions for the E51 hart to communicate with individual U54 harts to perform certain services on their behalf.

### 3.7.3 Initializing the Application Execution Space (LIM or DDR) (Ask a Question)

The E51 core runs the HSS to fetch a payload containing an application or applications from an external memory. It buffers the application to DDR and then copies the application to the destination memory and releases the U54(s) from WFI. The U54 cores run the application from the destination memory. Linker script files to execute applications from different memory locations are available from the PolarFire SoC Bare Metal Library .

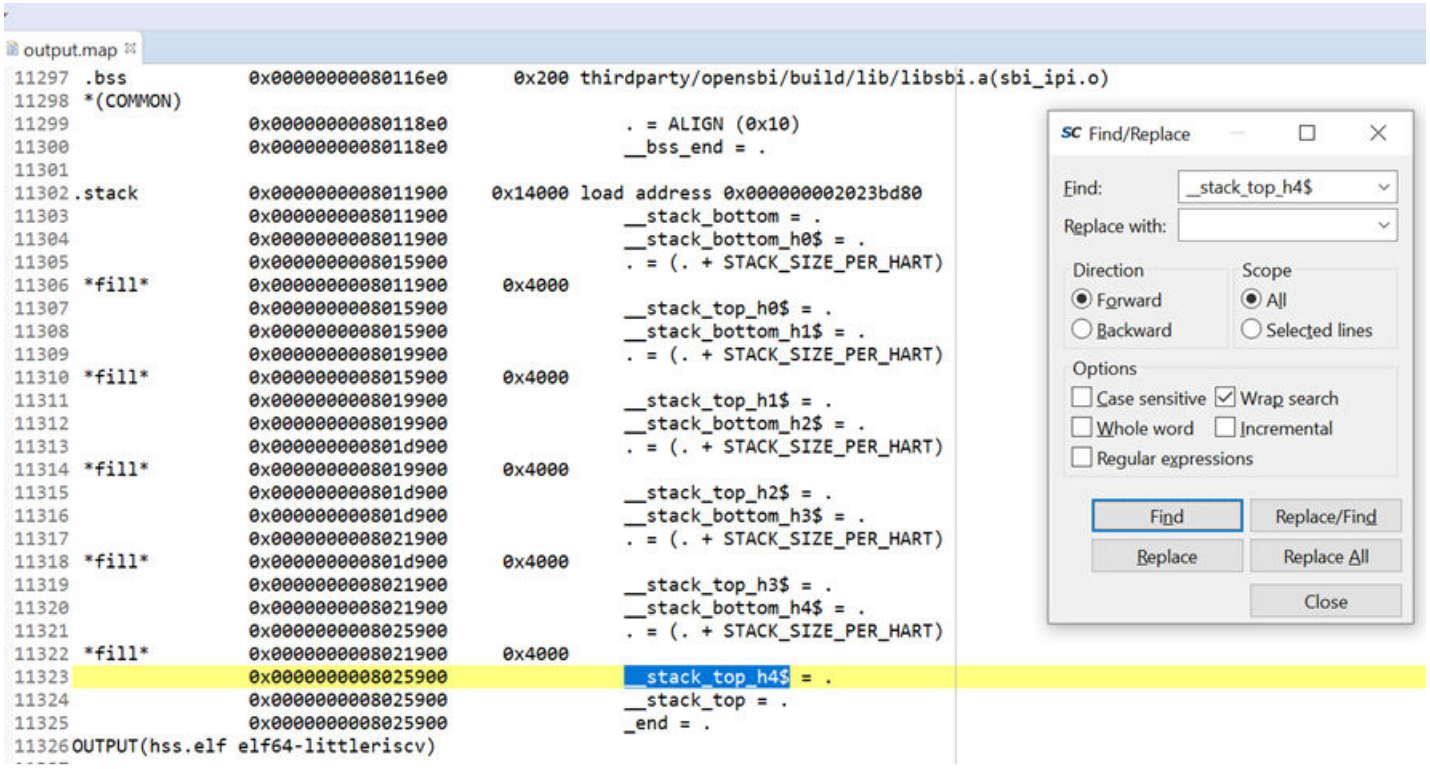
The HSS uses a portion of the LIM while running. The Bare Metal application start addresses must be greater than the end address of memory used by HSS. When built a `.map` file is produced for the HSS outlining memory locations used for functions, variables, and so on. The following image shows a `output.map` file.

**Figure 3-14.** HSS output.map File



This file can be opened in the SoftConsole text editor or an external editor. The last region of memory used by HSS is the stack, searching the map file for `__stack_top_h4$` shows the last address used for data in the stack, as shown in the following figure.

**Figure 3-15.** Stack Location HSS output.map File




In the preceding example, the last address used in the HSS is 0x8025900 and the target Bare Metal application must use an address greater than this as its start address.

The standard Bare Metal Library applications use E51 as the main hart of the system. When the E51 hart runs the HSS, a different hart, one of the U54s, must be used to wake harts in use from the WFI mode. The main hart defined in the `mss_sw_config.h` file in `platform/mpfs_hal_config` must be changed to reflect that the U54 core is the main hart.

The following figure shows a standard `mss_sw_config.h` configuration.

**Figure 3-16.** Standard mss_sw_config.h File

mss_sw_config.h

```
#ifndef USER_CONFIG_MSS_USER_CONFIG_H_
#define USER_CONFIG_MSS_USER_CONFIG_H_

/*-------------------------------------------------------------------------
 * MPFS_HAL_FIRST_HART and MPFS_HAL_LAST_HART defines used to specify which
 * harts to actually start.
 * Set MPFS_HAL_FIRST_HART to a value other than 0 if you do not want your code
 * to start and execute code on the E51 hart.
 * Set MPFS_HAL_LAST_HART to a value smaller than 4 if you do not wish to use
 * all U54 harts.
 * Harts that are not started will remain in an infinite WFI loop unless used
 * through some other method
 */
#ifndef MPFS_HAL_FIRST_HART
#define MPFS_HAL_FIRST_HART  0
#endif

#ifndef MPFS_HAL_LAST_HART
#define MPFS_HAL_LAST_HART   4
#endif

/*
```

The MPFS_HAL_FIRST_HART is set to 0 and the MPFS_HAL_LAST_HART is set to 4. The hart values are as follows:

E51: Hart 0

U54_1: Hart 1

U54_2: Hart 2

U54_3: Hart 3

U54_4: Hart 4

Changing the MPFS_HAL_FIRST_HART value to 1 sets U54_1 as the main hart of the system.

**Figure 3-17.** mss_sw_config.h File Using U54_1 as the Main Hart

*mss_sw_config.h

```
#ifndef USER_CONFIG_MSS_USER_CONFIG_H_
#define USER_CONFIG_MSS_USER_CONFIG_H_

/*------------------------------------------------------------------------------
 * MPFS_HAL_FIRST_HART and MPFS_HAL_LAST_HART defines used to specify which
 * harts to actually start.
 * Set MPFS_HAL_FIRST_HART to a value other than 0 if you do not want your code
 * to start and execute code on the E51 hart.
 * Set MPFS_HAL_LAST_HART to a value smaller than 4 if you do not wish to use
 * all U54 harts.
 * Harts that are not started will remain in an infinite WFI loop unless used
 * through some other method
 */
#ifndef MPFS_HAL_FIRST_HART
#define MPFS_HAL_FIRST_HART  1
#endif

#ifndef MPFS_HAL_LAST_HART
#define MPFS_HAL_LAST_HART   4
#endif

/*
```

This implies that on system start up U54_1 wakes the other three U54 harts from WFI. If less harts are used by this application (for example, U54_1 and U54_2 only), then the last hart value can be changed.

**Figure 3-18.** mss_sw_config.h File

*mss_sw_config.h

```
#ifndef USER_CONFIG_MSS_USER_CONFIG_H_
#define USER_CONFIG_MSS_USER_CONFIG_H_

/*------------------------------------------------------------------------------
 * MPFS_HAL_FIRST_HART and MPFS_HAL_LAST_HART defines used to specify which
 * harts to actually start.
 * Set MPFS_HAL_FIRST_HART to a value other than 0 if you do not want your code
 * to start and execute code on the E51 hart.
 * Set MPFS_HAL_LAST_HART to a value smaller than 4 if you do not wish to use
 * all U54 harts.
 * Harts that are not started will remain in an infinite WFI loop unless used
 * through some other method
 */
#ifndef MPFS_HAL_FIRST_HART
#define MPFS_HAL_FIRST_HART  1
#endif

#ifndef MPFS_HAL_LAST_HART
#define MPFS_HAL_LAST_HART   2
#endif

/*------------------------------------------------------------------------------
```

This uses U54_1 to wake U54_2 and then both harts continue on to the application.

### 3.7.4 Merging Multiple Bare Metal Applications (Ask a Question)

To merge Bare Metal applications, see the HSS readme in its GitHub repository.

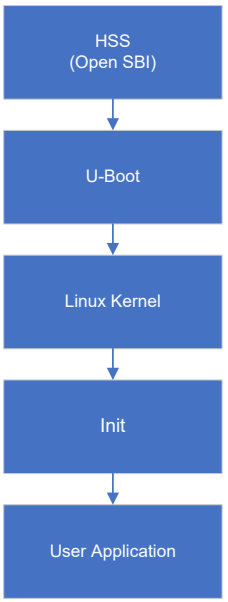
## 3.8 Linux Application Development (Ask a Question)

A typical boot process consists of multiple stages.

1. The HSS is executed from the eNVM. The HSS acts as a ZSBL that includes the Open Source Supervisor Binary Interface (OpenSBI). OpenSBI is a platform-specific firmware running in M-mode. It acts as an interface between the HSS and U-boot. The HSS loads the first stage boot loader (U-Boot) from a boot device to an external RAM. The HSS uses the OpenSBI functions to switch the execution mode from M-mode to S-mode when transferring execution control to U-Boot.
2. U-Boot initializes the peripherals and loads the kernel. The boot device can either be an embedded memory microcontroller (eMMC) or an SD card. U-Boot loads the Linux kernel from the boot device to DDR.
3. In the next stage, Linux is executed (from DDR).
4. Init is the first process executed by the Linux kernel and it is the parent of all processes.
5. In the final stage, user applications are executed in Linux.

The following figure shows the boot process flow.

**Figure 3-19.** Linux Boot Process Flow




### 3.8.1 Building Linux Images (Ask a Question)

Linux images can be built using the Yocto or Buildroot build systems. Both of these systems come with a readme that lists the required packages and build steps.

The Microchip Yocto BSP can be found at: mi-v-ecosystem.github.io/redirects/repo-meta-polarfire-soc-yocto-bsp.

It supports several board targets and build configurations for different images (for example, minimal/development tools).

The Microchip Buildroot External can be found at: mi-v-ecosystem.github.io/redirects/repo-polarfire-soc-buildroot-sdk.

It supports several board targets and is pre-configured to generate a minimal image.

### 3.8.2 Integrating Linux Applications in Yocto (Ask a Question)

#### 3.8.2.1 Existing Linux Applications (Ask a Question)

To integrate an existing Linux application (package) into Yocto, ensure that the package is a part of the Yocto source and add the package to the final image. An example on integrating Linux package using apache2 is shown in the following steps.

1. Find the package `.bb` file in the Yocto repository `apache2` (web server), which is shown as an example in the following code snippet.

```
microchip@microchip-OptiPlex-9010:~/riscv/icicle/yocto-dev$
microchip@microchip-OptiPlex-9010:~/riscv/icicle/yocto-dev$ find ./meta-* -name apache2
./meta-openembedded/meta-webserver/recipes-httpd/apache2
./meta-openembedded/meta-webserver/recipes-httpd/apache2/apache2
```

2. Ensure that the package meta layer directory is present in the `bblayers.conf` file.

```
microchip@microchip-OptiPlex-9010:~/riscv/icicle/yocto-dev$
microchip@microchip-OptiPlex-9010:~/riscv/icicle/yocto-dev$ ls build/conf/bblayers.conf
build/conf/bblayers.conf
```

3. If the package meta layer directory is not part of `bblayers.conf` file, add the directory path as highlighted in the following code snippet.

```
# LAYER_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
LCONF_VERSION = "7"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
     /home/microchip/riscv/icicle/yocto-dev/openembedded-core/meta \
     /home/microchip/riscv/icicle/yocto-dev/meta-openembedded/meta-oe \
     /home/microchip/riscv/icicle/yocto-dev/meta-openembedded/meta-python \
     /home/microchip/riscv/icicle/yocto-dev/meta-openembedded/meta-multimedia \
     /home/microchip/riscv/icicle/yocto-dev/meta-openembedded/meta-networking \
     /home/microchip/riscv/icicle/yocto-dev/meta-openembedded/meta-webserver \
     /home/microchip/riscv/icicle/yocto-dev/meta-riscv \
     /home/microchip/riscv/icicle/yocto-dev/meta-polarfire-soc-yocto-bsp \
     "
```

4. The following code snippet shows the `mpfs-dev-cli.bb` files folder in Yocto source for PolarFire SoC, open the `mpfs-dev-cli.bb` file.

```
microchip@microchip-OptiPlex-9010:~/riscv/icicle/yocto-dev$
microchip@microchip-OptiPlex-9010:~/riscv/icicle/yocto-dev$ ls meta-polarfire-soc-yocto-
bsp/recipes-core/images/
mpfs-dev-cli.bb  riscv-initramfs-image.bb
```

5. The `mpfs-dev-cli.bb` files show a list of packages added in the PolarFire SoC device. Add `apache2` package in the existing list as shown in the following code snippet.

```
DESCRIPTION = "Microchip MPFS Development CLI Linux image"

inherit image-buildinfo core-image extrausers
EXTRA_USERS_PARAMS = "usermod -P microchip root;"

IMAGE_FEATURES += " ssh-server-openssh \
                    tools-debug tools-sdk debug-tweaks \
                    dev-pkgs dbg-pkgs \
                    "

IMAGE_INSTALL = "\
    packagegroup-core-boot \
    packagegroup-core-full-cmdline \
    perl-modules \
    alsa-utils \
    i2c-tools \
    apache2 \
    screen \
    apps \
    vim vim-vimrc \
    dhcp-client \
    nbd-client \
    mpfr-dev \
    gmp-dev \
    libmpc-dev \
    zlib-dev \
    flex \
    bison \
    dejagnu \
    gettext \
    texinfo \
    procps \
    glibc-dev \
    elfutils \
    elfutils-dev \
    pciutils \
    usbutils \
    mtd-utils \
    sysfsutils \
    htop \
    iw \
    python3 \
    git \
    swig \
    boost \
    orc \
    libudev \
    glib-2.0 \
    evtest devmem2 iperf3 memtester lmbench \
    tcpdump \
    iw \
    libudev \
    nano \
    nfs-utils-client \
    cifs-utils \
    openssh-sftp \
    openssh-sftp-server \
    procps \
    protobuf \
    ntp ntpdate ntp-utils \
    linux-firmware \
    libsodium \
    sqlite3 \
    tar \
    wget \
    zip \
    unzip \
    rsync \
    kernel-modules kernel-devsrc kernel-dev \
    ${CORE_IMAGE_EXTRA_INSTALL} \
```

Upon successful addition of the `apache2` package as shown in the preceding steps, Yocto source can be built.

### 3.8.2.2 Custom Linux Applications (Ask a Question)

To integrate an existing Linux application into Yocto, ensure that the particular package is part of the Yocto source and add the package to the final image. An example of integrating Linux application using apache2 is shown in the following steps.

**Note:** The user package recipe must be placed under the `apps` folder to be part of the Yocto build process.

The following code snippet shows a sample application `.bb` file.

```
#
# This file was derived from the 'Hello World!' example recipe in the
# Yocto Project Development Manual.
#

DESCRIPTION = "Simple application to blink LEDs"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"
PR = "r0"
SRC_URI = "file://led_blinky.c \
          "

S = "${WORKDIR}/"

do_compile() {
                ${CC} led_blinky.c ${LDFLAGS} -o led_blinky
}

FILES_${PN} += "/microchip-apps"
do_install() {
             install -d ${D}/microchip-apps
             install -m 0755 led_blinky.c ${D}/microchip-apps
             install -m 0755 led_blinky ${D}/microchip-apps
}
```

The `mpfs-dev-cli.bb` file shows the list of packages added in the PolarFire SoC device. Add `apps` package in the existing list as shown in the following code snippet.

```
DESCRIPTION = "Microchip MPFS Development CLI Linux image"

inherit image-buildinfo core-image extrausers
EXTRA_USERS_PARAMS = "usermod -P microchip root;"

IMAGE_FEATURES += " ssh-server-openssh \
                    tools-debug tools-sdk debug-tweaks \
                    dev-pkgs dbg-pkgs \
                    "

IMAGE_INSTALL = "\
    packagegroup-core-boot \
    packagegroup-core-full-cmdline \
    perl-modules \
    alsa-utils \
    i2c-tools \
    screen \
    apps \
    vim vim-vimrc \
    dhcp-client \
    nbd-client \
    mpfr-dev \
    gmp-dev \
    libmpc-dev \
    zlib-dev \
    flex \
    bison \
    dejagnu \
    gettext \
    texinfo \
    procps \
    glibc-dev \
    elfutils \
    elfutils-dev \
    pciutils \
    usbutils \
    mtd-utils \
    sysfsutils \
    htop \
    iw \
    python3 \
    git \
    swig \
    boost \
    orc \
    libudev \
    glib-2.0 \
    evtest devmem2 iperf3 memtester lmbench \
    tcpdump \
    iw \
    libudev \
    nano \
    nfs-utils-client \
    cifs-utils \
    openssh-sftp \
    openssh-sftp-server \
    procps \
    protobuf \
    ntp ntpdate ntp-utils \
    linux-firmware \
    libsodium \
    sqlite3 \
    tar \
    wget \
    zip \
    unzip \
    rsync \
    kernel-modules kernel-devsrc kernel-dev \
    ${CORE_IMAGE_EXTRA_INSTALL} \
    "
```

Upon successful addition of the apps package as shown in the preceding code snippet, Yocto source can be built.

For more information about Yocto, see Yocto Reference Manual.

Microchip offers a complete Yocto source image for PolarFire SoC Icicle Kit that can be used for customized Linux images including application development. See GitHub page for more information about the directory structure, source files, and documentation.

### 3.8.3 Integrating Linux Application in Buildroot (Ask a Question)

#### 3.8.3.1 Existing Linux Applications (Ask a Question)

1. To integrate existing Linux application (package) into Buildroot, go to the PolarFire SoC Buildroot External path and execute the following commands:
   - To find all defconfigs available for Icicle kit, execute the command:

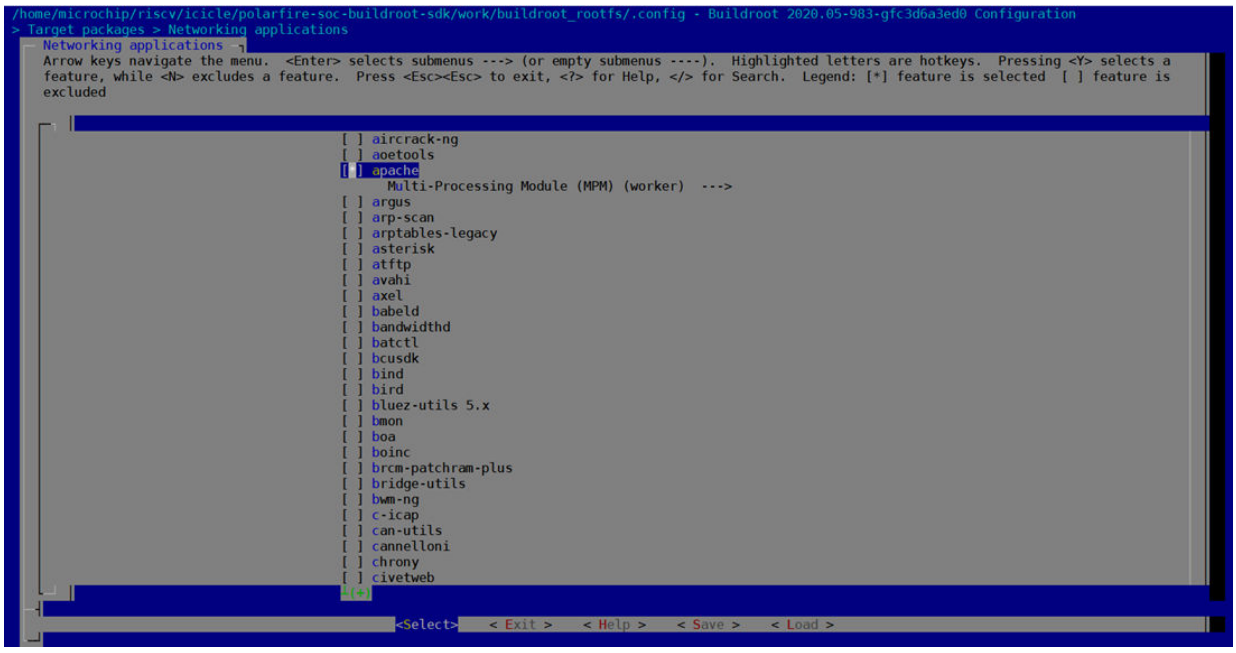
```
microchip@microchip-OptiPlex-9010:~/work/icicle$ ls
buildroot  buildroot-external-microchip
microchip@microchip-OptiPlex-9010:~/work/icicle$ ls ./buildroot-external-microchip/
configs/icicle_*
icicle_amp_defconfig     icicle_defconfig         icicle_nand_defconfig
icicle_nor_defconfig     icicle_rootfs_defconfig
```

   - To configure for the `icicle_defconfig`, execute the command:

```
microchip@microchip-OptiPlex-9010:~/work/icicle$ ls
buildroot  buildroot-external-microchip
microchip@microchip-OptiPlex-9010:~/work/icicle$ cd buildroot
microchip@microchip-OptiPlex-9010:~/work/icicle/buildroot$ BR2_EXTERNAL=../buildroot-
external-microchip/ make icicle_defconfig
BR2_EXTERNAL=../buildroot-external-microchip/ make menuconfig
```

2. When the make command is successfully executed, the Config menu appears as shown in the following figure. Search for the package that needs to be added. Select the package and **Save**. Apache is used as an example package in the following figure.

**Figure 3-20.** Apache Buildroot




Upon successful addition of the apache package as shown in the preceding image, the Buildroot source can be built.

### 3.8.3.2 Custom Linux Applications (Ask a Question)

To integrate a custom Linux application (package) into buildroot-external-microchip, follow these steps:

1. Create a new directory in the package folder of buildroot-external-microchip source. For example, `microchip-apps` package folder is used as an example, which consists of two files —`Config.in` and `microchip-apps.mk`. If you have any C Source files and corresponding makefile, create a directory `files` to store the source files.
   The following code snippet shows the `microchip-apps` folder.

```
microchip@microchip-OptiPlex-9020:~/work/icicle/buildroot-external-microchip/package/
microchip-apps$ ls
Config.in files microchip-apps.mk
microchip@microchip-OptiPlex-9020:~/work/icicle/buildroot-external-microchip/package/
microchip-apps$
```

2. Create a `Config.in` file with the following code snippet.

```
# ====================================
# package/microchip-apps/Config.in
# ====================================
config BR2_PACKAGE_MICROCHIP_APPS
   bool "microchip-apps"
   help
   Microchip applications for blinking LEDs
```

3. Create a `microchip-apps.mk` file with the following code snippet.

```
# ======================================
# package/microchip-apps/microchip-apps.mk
# ======================================
MICROCHIP_APPS_VERSION = 1.0
MICROCHIP_APPS_SITE = $(BR2_EXTERNAL_MCHP_PATH)/package/microchip-apps/files
MICROCHIP_APPS_SITE_METHOD = local


define MICROCHIP_APPS_BUILD_CMDS
        $(MAKE) $(TARGET_CONFIGURE_OPTS) -C $(@D) all
endef

define MICROCHIP_APPS_INSTALL_TARGET_CMDS
        $(INSTALL) -D -m 0755 $(@D)/led_blinky $(TARGET_DIR)/microchip-apps/led_blinky
endef

$(eval $(generic-package))
```

4. Edit the parent package config file (`Config.in`) to include the `microchip-apps` package.

```
source "$BR2_EXTERNAL_MCHP_PATH/package/microchip-apps/Config.in"
```

5. Add the `microchip-apps` package to the `configs/icicle_defconfig` file to build the `microchip-apps` package as part of the icicle_defconfig Linux image.

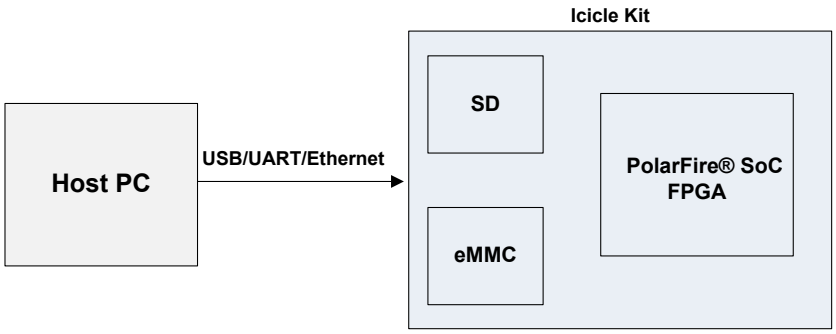
```
BR2_LINUX_KERNEL_DTB_KEEP_DIRNAME=y
BR2_LINUX_KERNEL_DTB_OVERLAY_SUPPORT=y
BR2_PACKAGE_LINUX_TOOLS_GPIO=y
BR2_PACKAGE_LINUX_TOOLS_IIO=y
BR2_PACKAGE_BUSYBOX_SHOW_OTHERS=y
BR2_PACKAGE_DHRYSTONE=y
BR2_PACKAGE_LMBENCH=y
BR2_PACKAGE_DIFFUTILS=y
BR2_PACKAGE_FINDUTILS=y
BR2_PACKAGE_GAWK=y
BR2_PACKAGE_E2FSPROGS=y
BR2_PACKAGE_MTD=y
BR2_PACKAGE_MICROCHIP_APPS=y
BR2_PACKAGE_GPTFDISK=y
BR2_PACKAGE_GPTFDISK_GDISK=y
BR2_PACKAGE_GPTFDISK_SGDISK=y
BR2_PACKAGE_I2C_TOOLS=y
BR2_PACKAGE_PCIUTILS=y
BR2_PACKAGE_RNG_TOOLS=y
```

You can build the Buildroot source on adding the `apps` package, successfully.

### 3.8.4 Different Sources of Booting (Ask a Question)

The PolarFire SoC target hardware runs the HSS from eNVM to load the Linux image either from eMMC or SD card depending on its configuration when built as shown in the following figure.

**Figure 3-21.** Different Sources of Booting




When the customized Linux images are available, USB, UART or Ethernet can be used to transfer the image to eMMC or SD card.

To boot Linux on Icicle kit using eMMC, see GitHub.

### 3.8.5 Device Tree Source (DTS) (Ask a Question)

A Device Tree is a data structure for describing the hierarchy of hardware subsystems within a hardware platform, or an add-on peripheral to that platform. It is used to select and configure the device drivers for embedded Linux platform during the boot process. It can be represented in different formats:

- Device Tree Source format (`.dts`)
- Compiled binary Device Tree binary format (`.dtb`)
- For example, `mpfs-icicle-kit.dts and mpfs-icicle-kit.dtb`

Additional peripherals like GPIO, LSRAM, and so on, can be added to a kernel by including them in the DTS.

#### 3.8.5.1 Adding a Sample Device Node for GPIO (Ask a Question)

The following example device node can be added to a Device Tree.

```
gpio@20122000 {
        compatible = "microchip,mpfs-gpio";
        reg = <0x20122000 0x1000>;
        clocks = <&clkcfg 25>;
        interrupt-parent = <&irqmux>;
        gpio-controller;
        #gpio-cells = <2>;
        ngpios = <32>;
        interrupt-controller;
        #interrupt-cells = <2>;
        interrupts = <64>, <65>, <66>, <67>,
                     <68>, <69>, <70>, <71>,
                     <72>, <73>, <74>, <75>,
                     <76>, <77>, <78>, <79>,
                     <80>, <81>, <82>, <83>,
                     <84>, <85>, <86>, <87>,
                     <88>, <89>, <90>, <91>,
                     <92>, <93>, <94>, <95>;
    };
```

### 3.8.5.2 Adding a Sample Device node for LSRAM (UIO Framework) (Ask a Question)

The following example device node can be added to device tree (mpfs-icicle-kit-fabric.dtsi).

```
uio_lsram@0x60000000 {
compatible = "generic-uio";
reg = < 0x0 0x60000000 0x0 0x00010000 // LSRAM0 Memory
0x0 0x60010000 0x0 0x00010000 >; //LSRAM1 Memory
status = "okay";
};
```

Hardware that is ideally suited for an UIO driver fulfills all the following:

- The device has memory that can be mapped. The device can be controlled completely by writing to this memory.
- The device usually generates interrupts.
- The device does not fit into one of the standard kernel subsystems.

# 4. Appendix (Ask a Question)

### HSS

The HSS is a ZSBL that is stored in the eNVM of PolarFire SoC. It can be used to program memories and boot applications running on different harts in the system. It runs in a super loop executing on the E51 core and provides a machine mode trap handler to pass messages between the U54 harts.

The HSS can be found on GitHub at: mi-v-ecosystem.github.io/redirects/repo-hart-software-services.

### HAL

The MPFS HAL provides the initial boot code, interrupt handling, hardware access methods for the PolarFire SoC MSS and DDR training code. The terms PolarFire-SoC HAL and MPFS HAL are used interchangeably but the term MPFS HAL is preferred. The MPFS HAL is a combination of C and assembly source code.

Location of the repository: mi-v-ecosystem.github.io/redirects/repo-platform.

### Peripheral Driver Library

The PolarFire SoC Bare Metal Library includes:

- Source code for start-up code and HAL the PolarFire SoC MSS.
- Source code for the PolarFire SoC MSS peripheral drivers.
- Documentation for the HAL and peripheral drivers.
- SoftConsole example projects demonstrating the use of the various PolarFire SoC peripherals.

You can access the PolarFire SoC Bare Metal library related resources at mi-v-ecosystem.github.io/redirects/repo-polarfire-soc-bare-metal-library.

# 5. Revision History (Ask a Question)

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

| Revision | Date | Description |
|---|---|---|
| E | 04/2025 | The following is a summary of changes made in this revision:<br>• Updated the boot modes information in Device Boot and Configuration Process.<br>• Replaced “Buildroot SDK” with “Buildroot External” throughout the document.<br>• Updated file paths and commands with respect to Buildroot External in Existing Linux Applications and Custom Linux Applications.<br>• Updated the device tree source information, see Device Tree Source (DTS), Adding a Sample Device Node for GPIO, and Adding a Sample Device node for LSRAM (UIO Framework). |
| D | 09/2022 | Added information about a new script made available for building bitstreams, see Version Controlling Bitstreams. |
| C | 05/2022 | Enabled ‘Ask A Question’ hyperlink for each section in the document. |
| B | 05/2021 | Updated the References section |
| A | 09/2020 | Initial Revision |

# Microchip FPGA Support

Microchip FPGA products group backs its products with various support services, including Customer Service, Customer Technical Support Center, a website, and worldwide sales offices. Customers are suggested to visit Microchip online resources prior to contacting support as it is very likely that their queries have been already answered.

Contact Technical Support Center through the website at www.microchip.com/support. Mention the FPGA Device Part number, select appropriate case category, and upload design files while creating a technical support case.

Contact Customer Service for non-technical product support, such as product pricing, product upgrades, update information, order status, and authorization.

- From North America, call **800.262.1060**
- From the rest of the world, call **650.318.4460**
- Fax, from anywhere in the world, **650.318.8044**

# Microchip Information

## Trademarks

The “Microchip” name and logo, the “M” logo, and other names, logos, and brands are registered and unregistered trademarks of Microchip Technology Incorporated or its affiliates and/or subsidiaries in the United States and/or other countries (“Microchip Trademarks”). Information regarding Microchip Trademarks can be found at https://www.microchip.com/en-us/about/legal-information/microchip-trademarks.

ISBN: 979-8-3371-0946-6

## Legal Notice

This publication and the information herein may be used only with Microchip products, including to design, test, and integrate Microchip products with your application. Use of this information in any other manner violates these terms. Information regarding device applications is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. Contact your local Microchip sales office for additional support or, obtain additional support at www.microchip.com/en-us/support/design-help/client-support-services.

THIS INFORMATION IS PROVIDED BY MICROCHIP “AS IS”. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE, OR WARRANTIES RELATED TO ITS CONDITION, QUALITY, OR PERFORMANCE.

IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY INDIRECT, SPECIAL, PUNITIVE, INCIDENTAL, OR CONSEQUENTIAL LOSS, DAMAGE, COST, OR EXPENSE OF ANY KIND WHATSOEVER RELATED TO THE INFORMATION OR ITS USE, HOWEVER CAUSED, EVEN IF MICROCHIP HAS BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE. TO THE FULLEST EXTENT ALLOWED BY LAW, MICROCHIP’S TOTAL LIABILITY ON ALL CLAIMS IN ANY WAY RELATED TO THE INFORMATION OR ITS USE WILL NOT EXCEED THE AMOUNT OF FEES, IF ANY, THAT YOU HAVE PAID DIRECTLY TO MICROCHIP FOR THE INFORMATION.

Use of Microchip devices in life support and/or safety applications is entirely at the buyer’s risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

## Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip products:

- Microchip products meet the specifications contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is secure when used in the intended manner, within operating specifications, and under normal conditions.
- Microchip values and aggressively protects its intellectual property rights. Attempts to breach the code protection features of Microchip products are strictly prohibited and may violate the Digital Millennium Copyright Act.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of its code. Code protection does not mean that we are guaranteeing the product is “unbreakable”. Code protection is constantly evolving. Microchip is committed to continuously improving the code protection features of our products.