Freedom Metal Boot Code

Version 1.1

Freedom Metal Boot Code

Proprietary Notice

Release Information

Version	Date	Changes
V1.0	January 29, 2020	• Initial release
V1.1	February 4, 2020	• Minor clarifications

Contents

Chapter 1

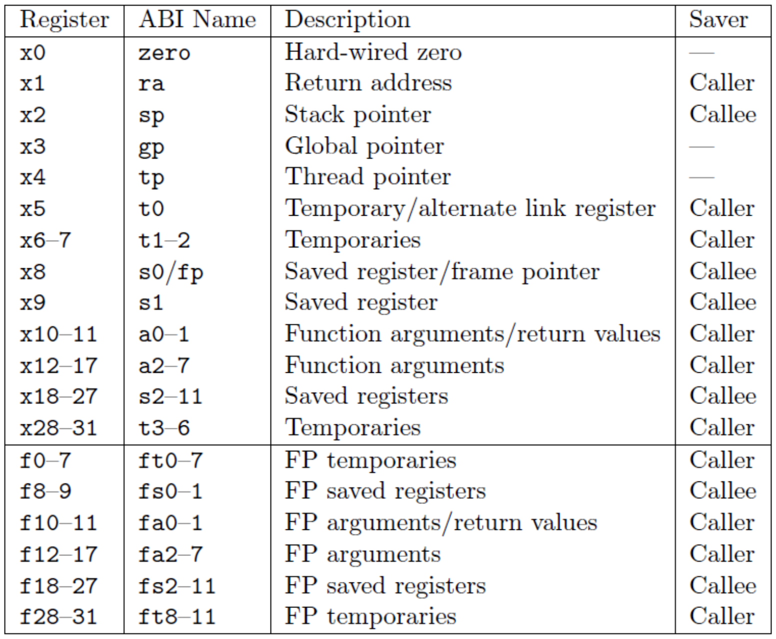
Freedom Metal Boot Code

1.1 Introduction

All software examples in the SiFive freedom-e-sdk github repository utilize low-level startup code which resides in the freedom-metal github repository. The freedom-metal repo is an API submodule that is integrated within freedom-e-sdk, along with software examples, which are distributed as part of all custom core IP tarball deliveries. This document describes the low level startup code contained in freedom-metal.

1.1.1 RISC-V User Registers

Prior to describing the details of the boot flow, it is worthwhile to understand the RISC-V user registers, and their representation within the defined Application Binary Interface (ABI). The ABI calling convention (link) describes how registers are used in functions and library calls within the software application.



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Figure 1: RISC-V Assembler mnemonics and ABI convention

A few notable descriptions include:

- Function arguments and return values start in x10
- `x0` is always hardwired to 0x0
- The `gp` is initialized at startup and should always be preserved, as it may be used to calculate jump offsets

Referencing each user register in assembly code can be accomplished by the register or ABI name, as the assembler understands both.

The full RISC-V Assembly Manual can be found here.

1.1.2 RISC-V Control & Status Registers (CSRs)

The CSR registers are only accessible using variations of the `csrr` (Read) and `csrw` write instructions. Only the CPU executing the `csr` instruction can read or write these registers, and they are not visible by software outside of the core they reside on. A few examples of assembler CSR instructions are shown below.

```
; Read mhartid CSR value into user register a0
csrr a0, mhartid

; load a0 with a predefined value of BITS_TO_SET
li a0, BITS_TO_SET;
; use CSR Set instruction to set these bits in mstatus
csrs mstatus, a0
```

```
; Write immediate data to floating point control and status register
csrwi fcsr, 0
```

A C code representation showing how to clear all bits in `medeleg` regsiter is shown below.

```
// The Compiler will select the register to use for the `%0` field.
__asm__ volatile ("csrc medeleg, %0" :: "r"(-1));
```

For more information on CSR registers, refer to the *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*, available at riscv.org.

Chapter 2

Boot Flow Components

The freedom-metal boot code supports single core boot or multi-core boot, and contains all the necessary initialization code to enable every core in the system. All SiFive core designs contain a single hardware thread, or **hart**, which may be used interchangeably for **core** or **cpu** in this document.

2.1 Linker File

The linker file generates important symbols that are used in the boot code. The linker file options are found in the freedom-e-sdk/bsp path. There are usually three different linker file options:

- metal.default.lds — Use flash and RAM sections
- metal.ramrodata.lds — Place read only data in RAM for better performance
- metal.scratchpad.lds — Places all code + data sections into available RAM location

Each linker option can be selected by selecting LINK_TARGET on the command line. For example:

```
make PROGRAM=hello TARGET=design-rtl CONFIGURATION=release LINK_TARGET=scratchpad
software
```

The metal.default.lds linker file is selected by default when LINK_TARGET is not specified. If there is a scenario where a custom linker is required, one of the supplied linker files can be copied and renamed and used for the build. For example, if a new linker file named metal.newmap.lds was generated, this can be used at build time by specifying `LINK_TARGET=newmap` on the command line.

Note

The linker files describe `heap` sections but they may not be needed if the application does not use library calls like `malloc` which require heap sections to be defined. There is no requirement to initialize the `heap` section in the startup code.

2.1.1 Linker File Symbols

The linker file generates symbols that are used by the startup code, so that software can use these symbols to assign the stack pointer, initialize or copy certain RAM sections, and provide the boot hart information. These symbols are made visible to software using the `PROVIDE` keyword. For example:

```
        __stack_size = DEFINED(__stack_size) ? __stack_size : 0x400;
        PROVIDE(__stack_size = __stack_size);
```

Generated Linker Symbols

A list and description of the generated linker symbols is shown below.

- `__metal_boot_hart`
  - This is an integer number to describe which hart runs the main init flow. The `mhartid` CSR contains the integer value for each hart. For example, hart 0 has mhartid==0, hart 1 has mhartid==1, and so on. An assembly example is shown below, where `a0` already contains the `mhartid` value.

```
/* If we're not hart 0, skip the initialization work */
la t0, __metal_boot_hart
bne a0, t0, _skip_init
```

An example on how to use this symbol in C code is shown below.

```
extern int __metal_boot_hart;
int boot_hart = (int)&__metal_boot_hart;
```

Additional linker file generated symbols, along with descriptions are shown below.

- `__metal_chicken_bit`
  - Status bit to tell startup code to zero out the Feature Disable CSR. Details of this register are internal use only.
- `__global_pointer$`
  - Static value used to write the `gp` register at startup.
- `_sp`

  - Address of the end of stack for hart 0, used to initialize the beginning of the stack since the stack grows lower in memory. On a multi-hart system, the start address of the stack for each hart is calculated using (`_sp + __stack_size * mhartid`)
- `metal_segment_bss_target_start` & `metal_segment_bss_target_end`
  - Used to zero out global data mapped to `.bss` section.
    - Only `__metal_boot_hart` runs this code.
- `metal_segment_data_source_start`, `metal_segment_data_target_start`, `metal_segment_data_target_end`
  - Used to copy data from image to its destination in RAM.
    - Only `__metal_boot_hart` runs this code.
- `metal_segment_itim_source_start`, `metal_segment_itim_target_start`, `metal_segment_itim_target_end`
  - Code or data can be placed in itim sections using the __attribute__section(".itim")
    - When this attribute is applied to code or data, the `metal_segment_itim_source_start`, `metal_segment_itim_target_start`, and `metal_segment_itim_target_end` symbols get updated accordingly, and these symbols allow the startup code to copy code and data into the ITIM area.
      - Only `__metal_boot_hart` runs this code.

Note

At the time of this writing, the boot flow does not support C++ projects

Chapter 3

Boot Flow Details

A high level overview of the boot flow is shown below, which is based on the 2019.08 tagged release of Freedom Metal.

1. **ENTRY POINT: File: freedom-metal/src/entry.S, label: `_enter`**
2. Initialize global pointer `gp` register using the generated symbol `__global_pointer$`.
3. Write `mtvec` register with `early_trap_vector` as default exception handler
4. Clear chicken bits (usage for this register is not made public)
5. Read `mhartid` into register `a0` and call `_start`, which exists in **crt0.S**
6. **We now transition to File: freedom-metal/gloss/crt0.S, label `_start`**
7. Initialize stack pointer, `sp`, with `_sp` generated symbol. Harts with `mhartid` of one or larger are offset by (`_sp + __stack_size * mhartid`). The `__stack_size` field is generated in the linker file.
8. Check if `mhartid == __metal_boot_hart` and run the init code if they are equal. All other harts skip init and go to the Post Init Flow, step #15.
9. **Boot Hart Init Flow Begins Here**
10. Init `data` section to destination in defined RAM space
11. Copy `ITIM` section, if ITIM code exists, to destination
12. Zero out `bss` section
13. Call `atexit` library function which registers the libc and freedom-metal destructors to run after `main` returns.
14. Call `__libc_init_array` library function, which runs all functions marked with `__attribute__((constructor))`.
    a. For example, PLL, UART, L2 if they exist in the design. This method provides full early initialization prior to entering the main application.

15. **Post Init Flow Begins Here**
16. Call the C routine `__metal_synchronize_harts`, where hart 0 will release all harts once their individual `msip` bits are set. The `msip` bit is typically used to assert a software interrupt on individual harts, however interrupts are not yet enabled, so `msip` in this case is used as a gatekeeping mechanism.
17. Check `misa` register to see if floating point hardware is part of the design, and set up `mstatus` accordingly.
18. Single or multi-hart design redirection step

    a. If design is a single hart only, or a multi-hart design without a C-implemented function `secondary_main`, ONLY the boot hart will continue to `main()`.

    b. For multi-hart designs, all other CPUs will enter sleep via `WFI` instruction via the weak `secondary_main` label in crt0.S, while boot hart runs the application program.

    c. In a multi-hart design which includes a C-defined `secondary_main` function, all harts will enter `secondary_main` as the primary C function.

Note

As described, the above flow is based on the 2019.08 tagged release of Freedom Metal, located here. Future released may include new initialization files `freedom-metal/src/init.c` and `freedom-metal/metal/init.h`, which may change the low level details described above. The new methodology is designed to provide additional user control and flexibility in the startup flow.

Chapter 4

Conclusion

The freedom-metal startup code works together with the linker generated symbols to setup the appropriate boot flow based on the hardware configuration and software requirements. Required steps such as stack setup, global pointer assignment, and specific linker section initialization (bss, itim, data) are fundamental to any application working correctly. Other features like heap sections depend on whether the end application requires it, and may be optional.