**# Title**:

Update 2.3.7 (RISC-V Platforms) for tighter/concise language (v3)

**# Status:**

Draft

**# Document**:

UEFI Specification 2.10+

**# License**:

SPDX-License-Identifier: CC-BY-4.0

**# Submitter**:

Intel (Andrei Warkentin)

**# Summary of the change**

Revise 2.3.7 (RISC-V Platforms) to avoid conflating specific implementation choices and platform firmware requirements with generic UEFI requirements on all RISC-V implementations (e.g removing language about Machine mode behavior). Augment with language consistent with existing implementations and other accepted calling convention norms for other architectures. Defer to external documentation where possible (e.g. ABI, privileged spec, etc) and avoid overly verbose sections that aren’t requirements and that with time will amount to a maintenance burden (e.g. the language today prevents VM implementations of UEFI).

**# Benefits of the change**

Bring spec to date with current UEFI implementations (U-Boot, Tiano) with the flexibility necessary for appropriate bare metal and virtual machine implementations, by removing language around UEFI and Machine mode interop (some of this language could move to the PI spec as one of the possible implementation choices)

**# References**

<https://bugzilla.tianocore.org/show_bug.cgi?id=4232>

**# Impact of the change**

This ECR is based on UEFI 2.10. The changes relax the requirements on how UEFI-compliant firmware for RISC-V is implemented. They also remove unnecessary descriptions that don’t add anything to the description of the requirements.

**# Detailed description of the change [normative updates]**

GREEN add, RED DELETE, YELLOW MODIFY

**2.3.7 RISC-V Platforms**

UEFI implementations may target RV32 (32-bit), RV64 (64-bit) and RV128 (128-bit) processors, supporting code execution in native bitness mode only (e.g. an RV64 UEFI implementation will not support RV32 UEFI images).

All functions are called with the C language calling convention. See *Detailed Calling Convention* for more detail. During boot services only a single processor is used for execution. All secondary processors must be either powered off or held in a quiescent state.

On RISC-V platform, three privileged levels are currently introduced in RISC-V architecture. Beyond the User privilege, Supervisor privilege and Machine privileges cover all aspects of RISC-V system. The privileged instructions are also defined in each privilege level.

Table 2.7: RISC-V Platforms

|  |  |  |  |
| --- | --- | --- | --- |
| **Level** | **Encoding** | **Name** | **Abbreviation** |
| 0 | 0 | User/Application | U |
| 1 | 1 | Supervisor | S |
| 2 | 10 | Reserved |  |
| 3 | 11 | Machine | M |

A RISC-V platform can contain one or more RISC-V cores and other components such as physical memory, fixed function accelerators and I/O devices. The term RISC-V core refers to a component which contains an independent instruction fetch unit. A RISC-V core may have multiple RISC-V-compatible hardware threads, or hart.

RISC-V UEFI firmware could be executed in either Machine mode or Supervisor mode during the entire POST, according to the hart capability and the platform design. However, RISC-V UEFI firmware has to switch the boot hart to Supervisor mode at either early or late POST if the platform is designed to boot a Supervisor mode OS or OS loader.

The machine mode has the highest privilege and this mode is the only mandatory privilege level for RISC-V platforms; all other privilege levels are optional depending on the platform requirements. Machine mode is the initial privilege mode entered at the power-on reset. This level is used in UEFI for low-level access to a hardware platform.

UEFI firmware implementation may provide the Supervisor Binary Interface (SBI) to allow the Supervisor mode execution environment to invoke privileged functions or access privileged hardware.

The processor is in the following execution mode during boot service time:

* The processor must be in little-endian Supervisor mode and running with native (XLEN) bitness. If the processor implements the hypervisor extension, the processor must be in HS mode.
* The processor must support the following extensions:
  + A - Atomic extension
  + C - Compressed extension
  + I – base (integer) ISA
  + M - Integer Multiplication and Division extension
  + Standard privileged architecture (Zicsr, Zifencei)
* Implementations of boot services will enable architecturally manageable caches and TLBs i.e., those that can be managed directly using implementation independent registers using mechanisms and procedures defined in the RISC-V Volume 2, Privileged Spec and ratified extension specifications. They should not enable caches requiring platform information to manage or invoke non-architectural cache/TLB lockdown mechanisms.
* Interrupts are enabled, though no interrupt services are supported other than the UEFI boot services timer functions (All loaded device drivers are serviced synchronously by “polling”).
* A timer is enabled and configured for Supervisor interrupt delivery, e.g. machine timer or supervisor timer if the Sstc extension is present.
* 128 KiB or more of available stack space.
* Total 32 general-purpose registers x1-x31. Register x0 is hardwired to 0. Each register has its ABI (Application Binary Interface) name. See *Detailed Calling Convention* for more detail.
* The width of the native base integer depends on the RISC-V privileged mode implementation. XLEN is a general term which used to refer the width of base integer in bits.  
  — For the Base Integer ISA in 32-bit width, XLEN = 32  
  — For the Base Integer ISA in 64-bit width, XLEN = 64  
  — For the Base Integer ISA in 128-bit width, XLEN = 128
* The width of processor registers could be determined by placing the immediate 4 in a register then shifting the register left by 31 bits at a time. If zero after one shift, then the machine is RV32. If zero after two shifts, then the machine is RV64, else RV128.
* Processor reset vector is platform specified. In UEFI, it is configured to the platform implementation-defined reset vector. The reset vector address is the first instruction which fetched by RISC-V processor when the power-on reset.
* The mcause value after reset have implementation-specific interpretation, value 0 should be returned on implementations that do not distinguish different reset conditions. Implementations that distinguish different reset conditions should only use 0 to indicate the most complete reset (e.g., hard reset). The causes of reset could be power-on reset, external hard reset, brownout detected, watchdog timer elapse, sleep-mode wakeup, etc., which machine-mode UEFI system firmware has to distinguish.
* The mstatus.xIE indicates the current processor interrupt activation in current privilege mode.  
  — mstatus.MIE is set to one while mstatus.SIE and mstatus.UIE are set to zero during early UEFI POST stage
* The machine mode interrupt is enabled during boot service in UEFI. Two kinds of interrupts are enabled, one is for timer interrupt and another is software interrupt.
* mie.MSIE = 1
* mie.MTIE = 1
* The memory is in physical addressing mode. Page is disabled in RISC-V machine mode during UEFI boot service.
* I/O access is through memory map I/O.
* Only support Machine level Control and Status Registers (CSRs) in UEFI.
* Machine ISA (misa) register contains the information regarding to the capabilities of CPU implementation. The misa.MXL field encodes the native base integer ISA width in machine mode. MXLEN (Machine XLEN) is given by setting of misa.MXL.  
  — misa.MXL = 1, MXLEN is 32 bit  
  — misa.MXL = 2, MXLEN is 64 bit  
  — misa.MXL = 3, MXLEN is 128 bit
* RISC-V processor supports extensive customization and specialization instruction sets. RISC-V variations provide various purposes of processor implementations and the processor capability is reported in the extension bits in misa register. UEFI drivers will need to know the capabilities of processor before executing the specified RISC-V extension instructions. The extensions fields encodes the presence of the standard extensions, with a single bit per letter of the alphabet. (Bit 0 encodes presence of extension “A”, Bit 1 encodes presence of extension “B” and so on. Currently the single letter extension mnemonics are as below,  
  — A - Atomic extension  
  — B - Tentatively reserved for Bit operations extension  
  — C - Compressed extension  
  — D - Double-Precision Floating-Point extension  
  — E - Reduced Register Set Indicator RV32E (16 registers)  
  — F - Single-Precision Floating-Point extension  
  — G - Additional standard extensions present  
  — H - Hypervisor extension  
  — I - RV32I/64I/128I base ISA  
  — J - Tentatively reserved for Dynamically Translated Languages extension  
  — K - Reserved  
  — L - Tentatively reserved for Decimal Floating-Point extension  
  — M - Integer Multiplication and Division extension  
  — N - User-level interrupts supported  
  — O - Reserved  
  — P - Tentatively reserved for Packed-SIMD extension  
  — Q - Quad-Precision Floating-Point extension  
  — S - Supervisor mode implemented  
  — T - Tentatively reserved for Transactional Memory extension  
  — U - User mode implemented  
  — V - Tentatively reserved for Vector extension  
  — W - Reserved  
  — X - Non-standard extension present  
  — Y - Reserved  
  — Z - Reserved  
  — Zifenci - Instruction-Fetch Fence  
  — Zicsr - Control and Status Register Access
* Machine Vendor ID Register  
  — The mvendorid is a 32-bit read-only register encoding the manufacture of the part. Value of 0 indicates this field is not implemented or this is a non-commercial implementation.
* Machine Architecture ID Register  
  — The marchid is an MXLEN-bit read-only register encoding the base microarchitecture of the hart. The combination of mvendorid and marchid should uniquely identify the type of hart microarchitecture that is implemented.
* Machine Implementation ID Register  
  — This provides a unique encoding of the version of processor implementation.

Runtime services are permitted to make ECALLs into higher privilege modes.

For an operating system to use any runtime services, it must:

* Preserve all memory in the memory map marked as runtime code and runtime data
* Call the runtime service functions, with the following conditions:
  + Call runtime services consistently from the same privilege mode (either HS/S or VS mode)
  + Runtime services must only be assigned to a single operating system or hypervisor. They must not be shared between multiple guest operating systems.
  + The system address regions described by all the entries in the EFI memory map that have the EFI\_MEMORY\_RUNTIME bit set must be identity mapped as they were for the EFI boot environment. If the OS Loader or OS used SetVirtualAddressMap() to relocate the runtime services in a virtual address space, then this condition does not have to be met. See description of SetVirtualAddressMap() for details of memory map after this function has been called.
  + The processor must be in a mode in which it has access to the syst  
    memory map with the EFI\_MEMORY\_RUNTIME bit set
  + 8 KiB, or more, of available stack space.
  + The stack must be 16-byte aligned (128-bit)
  + Interrupts may be disabled or enabled at the discretion of the caller.

An application written to this specification may alter the processor execution mode, but the UEFI image must ensure firmware boot services and runtime services are executed with the prescribed execution environment.  
After an Operating System calls ExitBootServices (), firmware boot services are no longer available and it is illegal to call any boot service. After ExitBootServices, firmware runtime services are still available and may be called with paging enabled and virtual address pointers if SetVirtualAddressMap () has been called describing all virtual address ranges used by the firmware runtime service.

If ACPI is supported:

* ACPI Tables loaded at boot time can be contained in memory of type *EfiACPIReclaimMemory* (recommended) or *EfiACPIMemoryNVS*. ACPI FACS must be contained in memory of type *EfiACPIMemoryNVS*
* The system firmware must not request a virtual mapping for any memory descriptor of type *EfiACPIReclaimMemory* or *EfiACPIMemoryNVS*.
* EFI memory descriptors of type *EfiACPIReclaimMemory* and *EfiACPIMemoryNVS* must be aligned on a 4 KiB boundary and must be a multiple of 4 KiB in size.
* Any UEFI memory descriptor that requests a virtual mapping via the *EFI\_MEMORY\_DESCRIPTOR* having the *EFI\_MEMORY\_RUNTIME* bit set must be aligned on a 4 KiB boundary and must be a multiple of 4 KiB in size.
* An ACPI Memory Op-region must inherit cacheability attributes from the UEFI memory map. If the system memory map does not contain cacheability attributes, the ACPI Memory Op-region must inherit its cacheability attributes from the ACPI name space. If no cacheability attributes exist in the system memory map or the ACPI name space, then the region must be assumed to be non-cacheable.
* ACPI tables loaded at runtime must be contained in memory of type *EfiACPIMemoryNVS*.  
  The cacheability attributes for ACPI tables loaded at runtime should be defined in the UEFI memory map. If no information about the table location exists in the UEFI memory map, cacheability attributes may be obtained from ACPI memory descriptors. If no information about the table location exists in the UEFI memory map or ACPI memory descriptors, the table is assumed to be non-cached.
* In general, UEFI Configuration Tables loaded at boot time (e.g., SMBIOS table) can be contained in memory of type *EfiRuntimeServicesData* (recommended), *EfiBootServicesData*, *EfiACPIReclaimMemory* or *EfiACPIMemoryNVS*. Tables loaded at runtime must be contained in memory of type *EfiRuntimeServicesData* (recommended) or *EfiACPIMemoryNVS*.

**Note**: *Previous EFI specifications allowed ACPI tables loaded at runtime to be in the* EfiReservedMemoryType *and there was no guidance provided for other EFI Configuration Tables*. EfiReservedMemoryType *is not intended to be used by firmware. The UEFI Specification intends to clarify the situation moving forward. Also, only OSes conforming to the UEFI Specification are guaranteed to handle SMBIOS table in memory of type* EfiBootServicesData

**2.3.7.1 Handoff State**When UEFI firmware hands off control to Supervisor mode OS, RISC-V boot hart must be operated in Supervisor mode, and the memory addressing must be operated in Bare mode which is no memory address translation or protection through the virtual page table entry.

In order to describe the heterogeneous RISC-V cores and harts for the next boot stage after POST, UEFI firmware must build up the information of core and hart hardware capabilities in the firmware data structure if the target bootable image requires this information. (e.g. If the platform supports SMBIOS structure, SMBIOS record type 44 record, see “Link to UEFI Specification-Related Document” on https://uefi.org/uefi under the heading “RISC-V Processor SMBIOS Specification”).

UEFI based RISC-V platform firmware must implement the RISCV\_EFI\_BOOT\_PROTOCOL for the boot image  
that requires the boot information during the handoff from the firmware boot stage. Launched EFI binaries should use the RISCV\_EFI\_BOOT\_PROTOCOL.GetBootHartId() to obtain the boot hart ID. The boot hart ID information provided by either SMBIOS or Device Tree (as described in the UEFI 2.9 specification) is ignored by the boot image. See “Links to UEFI Specification-Related Document” on https://uefi.org/uefi under the heading “RISC-V EFI Boot Protocol.”

If the platform supports Device Tree structure to describe the system configurations, the Flattened Device Blob (DTB) must be installed in the EFI Configuration Table (*EFI Configuration Table & Properties Table* for details)

All UEFI images takes two parameters: the UEFI image handle and the pointer to EFI System Table. According to the RISC-V calling convention, EFI\_HANDLE is passed through the a0 register and EFI\_SYSTEM\_TABLE is passed through the a1 register.

* x10 - EFI\_HANDLE (ABI name: a0)
* x11 - EFI\_SYSTEM\_TABLE (ABI name: a1)
* x1 - Return Address (ABI name: ra)

**2.3.7.2 Data Alignment**In the RV32I and RV64I, the datatypes must be aligned at its natural size when stored in memory. The following table describes the datatype and its alignment in RV32I and RV64I in UEFI.

Table 2.8: RV32 datatype alignment

|  |  |  |
| --- | --- | --- |
| **Datatype** | **Description** | **Alignment** |
| BOOLEAN | Logical Boolean | 1 |
| INTN | Signed value in native width. | 4 |
| UINTN | Unsigned value in native width. | 4 |
| INT8 | 1-byte signed value | 1 |
| UINT8 | 1-byte unsigned value | 1 |
| INT16 | 2-byte signed value | 2 |
| UINT16 | 2-byte unsigned value | 2 |
| INT32 | 4-byte signed value | 4 |
| UINT32 | 4-byte unsigned value | 4 |
| INT64 | 8-byte signed value | 8 |
| UINT64 | 8-byte unsigned value | 8 |
| CHAR8 | 1-byte character | 1 |
| CHAR16 | 2-byte character | 2 |
| VOID | Undeclared type | 4 |

Table 2.9: RV64 datatype alignment

|  |  |  |
| --- | --- | --- |
| **Datatype** | **Description** | **Alignment** |
| BOOLEAN | Logical Boolean | 1 |
| INTN | Signed value in native width. | 8 |
| UINTN | Unsigned value in native width. | 8 |
| INT8 | 1-byte signed value | 1 |
| UINT8 | 1-byte unsigned value | 1 |
| INT16 | 2-byte signed value | 2 |
| UINT16 | 2-byte unsigned value | 2 |
| INT32 | 4-byte signed value | 4 |
| UINT32 | 4-byte unsigned value | 4 |
| INT64 | 8-byte signed value | 8 |
| UINT64 | 8-byte unsigned value | 8 |
| CHAR8 | 1-byte character | 1 |
| CHAR16 | 2-byte character | 2 |
| VOID | Undeclared type | 8 |

**2.3.7.2 Enabling Paging or Alternate Translations in an Application**Boot Services define a specific execution environment. This section will describe how to write an application that creates an alternative execution environment. Some Operating Systems require the OS Loader to be able to enable OS required translations at Boot Services time, and make other changes to the UEFI defined execution environment.

If a UEFI application uses its own page tables, or other processor state, the application must ensure that the firmware executes with each supplanted functionality. There are two ways that firmware conforming to this specification can execute in this alternate execution environment:  
• Explicit firmware call  
• Firmware preemption of application via timer event

An application with an alternate execution environment can restore the firmware environment before each UEFI call. However the possibility of preemption may require the alternate execution-enabled application to disable interrupts while the alternate execution environment is active. It’s legal for the alternate execution environment enabled application to enable interrupts if the application catches the interrupt and restores the EFI firmware environment prior to calling the UEFI interrupt ISR. After the UEFI ISR context is executed it will return to the alternate execution environment enabled application context. An alternate execution environment created by a UEFI application must not change the semantics or behavior of the MMU configuration created by the UEFI firmware prior to invoking ExitBootServices(), including the bit layout of  
the page table entries.

After an OS loader calls ExitBootServices() it should immediately configure the exception vector to point to appropriate code.

**2.3.7.3 Detailed Calling Convention**

The base calling convention is defined in the *RISC-V ELF psABI Specification*. See Links to UEFI Specification-Related Documents (https://uefi.org/uefi) under the heading “RISC-V EFL psABI Specification”, and the RISC-V assembly programmer’s handbook section in the RISC-V Unprivileged ISA specification.

This binding further constrains the calling convention (EFIAPI) between UEFI-compliant images and firmware in the following manner:

* Datatypes must be aligned at its natural size when stored in memory (code shall make no assumptions on support for unaligned memory accesses)
* Calls will conform to LP64 ABI (make no use of floating point registers)
* Code may use RVC (compressed instructions).
* Optional floating point, vector and other extensions may be only used:
  + After dynamically checking for their existence
  + Saving and then later restoring any additional execution state, hiding use of the additional functionality from other components (incl. OS for EFI Runtime Service calls).
* Only little-endian operation is supported.
* The stack will maintain 16 byte alignment.
* UEFI firmware must neither trust the values of *tp* (x4) and *gp* (x3) nor make an assumption of owning the write access to these register in any circumstances. (Such as in EFI Boot service, EFI Runtime service, EFI Management Mode service and any UEFI firmware interfaces which may invoked by the EFI drivers, OS or external firmware payload.) Preserve the values in *gp* or *tp* register if UEFI firmware needs to change them, and never touch them after ExitBootServices(). Whether and how to preserve *gp* and *tp* in the UEFI firmware environment is implementation-specific

The RISC-V calling convention passes arguments in register when necessary. In RISC-V, total 32 general registers are declared, each register has its corresponding ABI name.

Table 2.10: Register name and ABI name

|  |  |  |
| --- | --- | --- |
| **Register** | **ABI Name** | **Description** |
| x0 | zero | Hardwired to zero |
| x1 | ra | Return address |
| x2 | sp | Stack pointer |
| x3 | gp | Global pointer |
| x4 | tp | Thread pointer |
| x5-7 | t0-2 | Temporaries |
| x8 | s0/fp | Saved register/frame pointer |
| x9 | s1 | Saved register |
| x10-11 | a0-1 | Function arguments/Return values |
| x12-17 | a2-7 | Function arguments |
| x18-27 | s2-11 | Saved registers |
| x28-31 | t3-6 | Temporaries |

In the RISC-V calling convention, up to eight integer registers are used for passing argument, a0-a7. a0-a7 are the ABI names and the corresponding registers are x10-x17. Values are returned from functions in integer registers a0 and a1, those are register x10 and x11. In the standard RISC-V calling convention, the stack grows downward and the stack pointer is always kept 16-byte aligned. Five integer register t0-t6 are temporary registers that are volatile across calls and must be saved by the caller if later used. Twelve integer registers s0-s11 are preserved across calls and must be saved by the callee if used.

In view of the following statement:  
*“In the standard ABI, procedures should not modify the integer registers tp and gp, because signal handlers may rely upon their values”*

mentioned in the RISC-V EFL psABI Specification, and the RISC-V calling convention that gp and tp registers are not assigned a specific owner to save and restore their values (see links below), UEFI firmware must neither trust the values of tp and gp nor make an assumption of owning the write access to these register in any circumstances. (Such as in EFI Boot service, EFI Runtime service, EFI Management Mode service and any UEFI firmware interfaces which may invoked by the EFI drivers, OS or external firmware payload.) Preserve the values in gp or tp register if UEFI firmware needs to change them, and never touch them after ExitBootServices(). Whether and how to preserve gp and tp in the UEFI firmware environment is implementation-specific. See Links to UEFI Specification-Related Documents (https://uefi.org/uefi) under the heading “RISC-V EFL psABI Specification”, and the RISC-V assembly programmer’s handbook section in the RISC-V Unprivileged ISA specification

**2.6.2 Platform-Specific Elements**

There are a number of elements that can be added or removed depending on the specific features that a platform requires. Platform firmware developers are required to implement UEFI elements based upon the features included. The following is a list of potential platform features and the elements that are required for each feature type:

1. If a platform includes console devices …  
     
   …
2. RISC-V platform firmware must implement the RISCV\_EFI\_BOOT\_PROTOCOL for the boot image  
   that requires the boot information during the handoff from the firmware boot stage. Launched EFI binaries should use the RISCV\_EFI\_BOOT\_PROTOCOL.GetBootHartId() to obtain the boot hart ID. The boot hart ID information provided by either SMBIOS or Device Tree (as described in the UEFI 2.9 specification) is ignored by the boot image. See “Links to UEFI Specification-Related Document” on https://uefi.org/uefi under the heading “RISC-V EFI Boot Protocol.”

**# Special Instructions**

Add links to referenced specs at uefi.org/uefi