# RISC-V Supervisor Binary Interface Specification

RISC-V Platform Specification Task Group
Version 0.3-rc0

# Table of Contents

# Copyright and license information

# Change Log

## Version 0.3-rc0

- Improved document styling and naming conventions
- Added SBI system reset extension
- Improved SBI introduction secion
- Improved documentation of SBI hart state managment extension
- Added suspend function to SBI hart state managment extension

## Version 0.2

- The entire v0.1 SBI has been moved to the legacy extension, which is now an optional extension. This is technically a backwards-incompatible change because the legacy extension is optional and v0.1 of the SBI doesn’t allow probing, but it’s as good as we can do.

# Chapter 1. Introduction

This specification describes the RISC-V Supervisor Binary Interface, known from here on as SBI. The SBI allows supervisor-mode (S-mode or VS-mode) software to be portable across all RISC-V implementations by defining an abstraction for platform (or hypervisor) specific functionality. The design of the SBI follows the general RISC-V philosophy of having a small core along with a set of optional modular extensions.

The higher privilege software providing SBI interface to the supervisor-mode software is referred to as a SBI implemenation or Supervisor Execution Environment (SEE). A SBI implementation (or SEE) can be platform runtime firmware executing in machine-mode (M-mode) (see below Figure 1) or it can be some hypervisor executing in hypervisor-mode (HS-mode) (see below Figure 2).

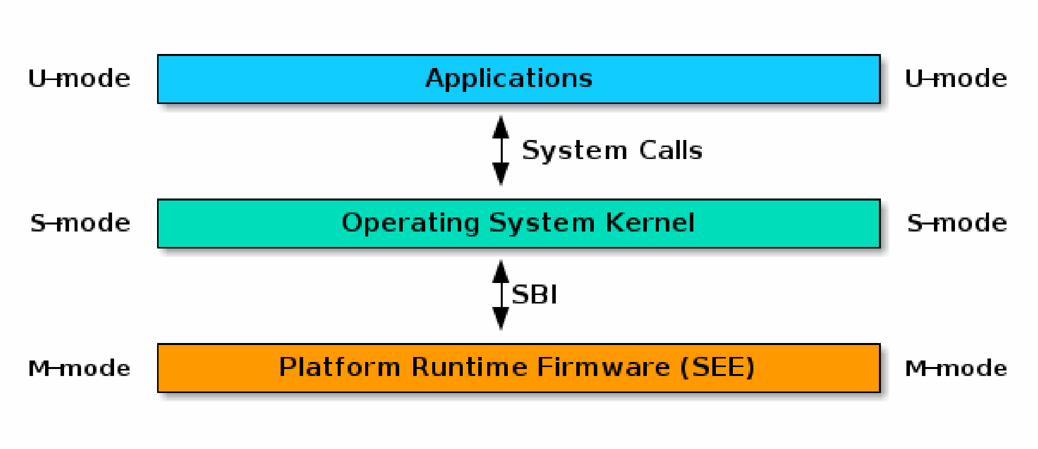

*Figure 1. RISC-V System without H-extension*

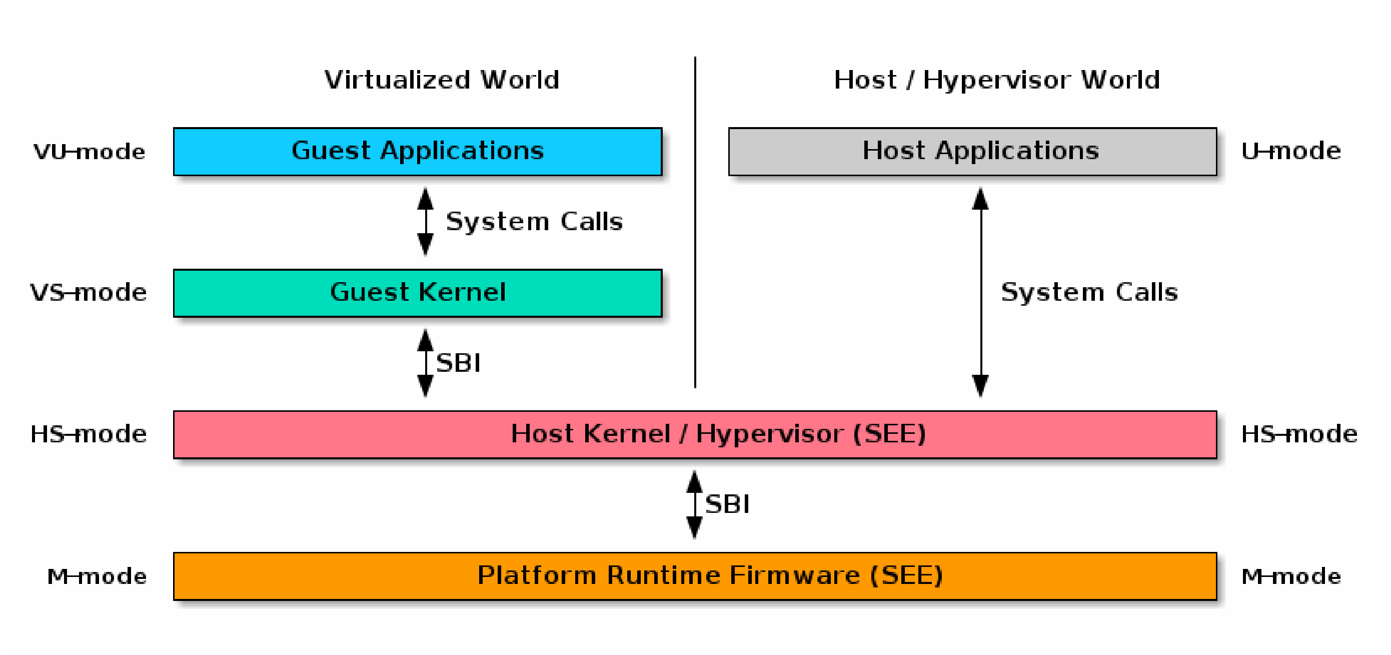

*Figure 2. RISC-V System with H-extension*

# Chapter 2. Binary Encoding

All SBI functions share a single binary encoding, which facilitates the mixing of SBI extensions. This binary encoding matches the RISC-V Linux syscall ABI, which itself is based on the calling convention defined in the RISC-V ELF psABI. In other words, SBI calls are exactly the same as standard RISC-V function calls except that:

- An `ECALL` is used as the control transfer instruction instead of a `CALL` instruction.
- `a7` (or `t0` on RV32E-based systems) encodes the SBI extension ID (**EID**), which matches how the system call ID is encoded in Linux system call ABI.

Many SBI extensions also chose to encode an additional SBI function ID (**FID**) in `a6`, a scheme similar to the `ioctl()` system call on many UNIX operating systems. This allows SBI extensions to encode multiple functions within the space of a single extension.

In the name of compatibility, SBI extension IDs (**EIDs**) and SBI function IDs (**FIDs**) are encoded as signed 32-bit integers. When passed in registers these follow the standard RISC-V calling convention rules.

SBI functions must return a pair of values in `a0` and `a1`, with `a0` returning an error code. This is analogous to returning the C structure

```
    struct sbiret {
        long error;
        long value;
    };
```

Standard SBI error codes are listed below

| Error Type | Value |
|---|---|
| SBI_SUCCESS | 0 |
| SBI_ERR_FAILED | -1 |
| SBI_ERR_NOT_SUPPORTED | -2 |
| SBI_ERR_INVALID_PARAM | -3 |
| SBI_ERR_DENIED | -4 |
| SBI_ERR_INVALID_ADDRESS | -5 |
| SBI_ERR_ALREADY_AVAILABLE | -6 |

An `ECALL` with an unsupported SBI extension ID (**EID**) or an unsupported SBI function ID (**FID**) must return the error code `SBI_ERR_NOT_SUPPORTED`.

Every SBI function should prefer `unsigned long` as the data type. It keeps the specification simple and easily adaptable for all RISC-V ISA types (i.e. RV32, RV64 and RV128). In case the data is defined as 32bit wide, higher privilege software must ensure that it only uses 32 bit data only.

If a SBI function needs to pass a list of harts to the higher privilege mode, it must use a hart mask as defined below. This is applicable to any extensions defined in or after v0.2.

Any function, requiring a hart mask, need to pass following two arguments.

- `unsigned long hart_mask` is a scalar bit-vector containing hartids
- `unsigned long hart_mask_base` is the starting hartid from which bit-vector must be computed.

In a single SBI function call, maximum number harts that can be set is always XLEN. If a lower privilege mode needs to pass information about more than XLEN harts, it should invoke multiple instances of the SBI function call. `hart_mask_base` can be set to `-1` to indicate that `hart_mask` can be ignored and all available harts must be considered.

Any function using hart mask may return following possible error value in addition to function specific error values.

| **Error code** | **Description** |
|---|---|
| SBI_ERR_INVALID_PARAM | Either `hart_mask_base` or any of the hartid from `hart_mask` is not valid i.e. either the hartid is not enabled by the platform or is not available to the supervisor. |

# Chapter 3. Base Extension (EID #0x10)

The base extension is designed to be as small as possible. As such, it only contains functionality for probing which SBI extensions are available and for querying the version of the SBI. All functions in the base extension must be supported by all SBI implementations, so there are no error returns defined.

## 3.1. Function: Get SBI specification version (FID #0)

```
struct sbiret sbi_get_spec_version(void);
```

Returns the current SBI specification version. This function must always succeed. The minor number of the SBI specification is encoded in the low 24 bits, with the major number encoded in the next 7 bits. Bit 31 must be 0 and is reserved for future expansion.

## 3.2. Function: Get SBI implementation ID (FID #1)

```
struct sbiret sbi_get_impl_id(void);
```

Returns the current SBI implementation ID, which is different for every SBI implementation. It is intended that this implementation ID allows software to probe for SBI implementation quirks.

## 3.3. Function: Get SBI implementation version (FID #2)

```
struct sbiret sbi_get_impl_version(void);
```

Returns the current SBI implementation version. The encoding of this version number is specific to the SBI implementation.

## 3.4. Function: Probe SBI extension (FID #3)

```
struct sbiret sbi_probe_extension(long extension_id);
```

Returns 0 if the given SBI extension ID (**EID**) is not available, or an extension-specific non-zero value if it is available.

## 3.5. Function: Get machine vendor ID (FID #4)

```
struct sbiret sbi_get_mvendorid(void);
```

Return a value that is legal for the `mvendorid` CSR and 0 is always a legal value for this CSR.

## 3.6. Function: Get machine architecture ID (FID #5)

```
struct sbiret sbi_get_marchid(void);
```

Return a value that is legal for the `marchid` CSR and 0 is always a legal value for this CSR.

## 3.7. Function: Get machine implementation ID (FID #6)

```
struct sbiret sbi_get_mimpid(void);
```

Return a value that is legal for the `mimpid` CSR and 0 is always a legal value for this CSR.

## 3.8. Function Listing

| **Function Name** | **FID** | **EID** |
|---|---|---|
| sbi_get_sbi_spec_version | 0 | 0x10 |
| sbi_get_sbi_impl_id | 1 | 0x10 |
| sbi_get_sbi_impl_version | 2 | 0x10 |
| sbi_probe_extension | 3 | 0x10 |
| sbi_get_mvendorid | 4 | 0x10 |
| sbi_get_marchid | 5 | 0x10 |
| sbi_get_mimpid | 6 | 0x10 |

## 3.9. SBI Implementation IDs

| **Implementation ID** | **Name** |
|---|---|
| 0 | Berkeley Boot Loader (BBL) |
| 1 | OpenSBI |
| 2 | Xvisor |
| 3 | KVM |
| 4 | RustSBI |
| 5 | Diosix |

# Chapter 4. Legacy Extensions (EIDs #0x00 - #0x0F)

The legacy SBI extensions ignores the SBI function ID field, instead being encoded as multiple SBI extension IDs. Each of these extension IDs must be probed for directly.

The legacy SBI extensions is deprecated in favor of the other extensions listed below. The legacy console SBI functions (`sbi_console_getchar()` and `sbi_console_putchar()`) are expected to be deprecated; they have no replacement.

## 4.1. Extension: Set Timer (EID #0x00)

```
void sbi_set_timer(uint64_t stime_value)
```

Programs the clock for next event after **stime_value** time. This function also clears the pending timer interrupt bit.

If the supervisor wishes to clear the timer interrupt without scheduling the next timer event, it can either request a timer interrupt infinitely far into the future (i.e., (uint64_t)-1), or it can instead mask the timer interrupt by clearing `sie.STIE` CSR bit.

## 4.2. Extension: Console Putchar (EID #0x01)

```
void sbi_console_putchar(int ch)
```

Write data present in **ch** to debug console.

Unlike `sbi_console_getchar()`, this SBI call **will block** if there remain any pending characters to be transmitted or if the receiving terminal is not yet ready to receive the byte. However, if the console doesn’t exist at all, then the character is thrown away.

## 4.3. Extension: Console Getchar (EID #0x02)

```
int sbi_console_getchar(void)
```

Read a byte from debug console; returns the byte on success, or -1 for failure. Note. This is the only SBI call in the legacy extension that has a non-void return type.

## 4.4. Extension: Clear IPI (EID #0x03)

```
void sbi_clear_ipi(void)
```

Clears the pending IPIs if any. The IPI is cleared only in the hart for which this SBI call is invoked. `sbi_clear_ipi()` is deprecated because S-mode code can clear `sip.SSIP` CSR bit directly.

## 4.5. Extension: Send IPI (EID #0x04)

```
void sbi_send_ipi(const unsigned long *hart_mask)
```

Send an inter-processor interrupt to all the harts defined in hart_mask. Interprocessor interrupts manifest at the receiving harts as Supervisor Software Interrupts.

hart_mask is a virtual address that points to a bit-vector of harts. The bit vector is represented as a sequence of unsigned longs whose length equals the number of harts in the system divided by the number of bits in an unsigned long, rounded up to the next integer.

## 4.6. Extension: Remote FENCE.I (EID #0x05)

```
void sbi_remote_fence_i(const unsigned long *hart_mask)
```

Instructs remote harts to execute `FENCE.I` instruction. The `hart_mask` is same as described in `sbi_send_ipi()`.

## 4.7. Extension: Remote SFENCE.VMA (EID #0x06)

```
void sbi_remote_sfence_vma(const unsigned long *hart_mask,
                           unsigned long start,
                           unsigned long size)
```

Instructs the remote harts to execute one or more `SFENCE.VMA` instructions, covering the range of virtual addresses between start and size.

## 4.8. Extension: Remote SFENCE.VMA with ASID (EID #0x07)

```
void sbi_remote_sfence_vma_asid(const unsigned long *hart_mask,
                                unsigned long start,
                                unsigned long size,
                                unsigned long asid)
```

Instruct the remote harts to execute one or more `SFENCE.VMA` instructions, covering the range of virtual addresses between start and size. This covers only the given `ASID`.

## 4.9. Extension: System Shutdown (EID #0x08)

```
void sbi_shutdown(void)
```

Puts all the harts to shutdown state from supervisor point of view. This SBI call doesn’t return.

## 4.10. Function Listing

| Function Name | FID | EID | Replacement EID |
|---|---|---|---|
| sbi_set_timer | 0 | 0x00 | 0x54494D45 |
| sbi_console_putchar | 0 | 0x01 | N/A |
| sbi_console_getchar | 0 | 0x02 | N/A |
| sbi_clear_ipi | 0 | 0x03 | N/A |
| sbi_send_ipi | 0 | 0x04 | 0x735049 |
| sbi_remote_fence_i | 0 | 0x05 | 0x52464E43 |
| sbi_remote_sfence_vma | 0 | 0x06 | 0x52464E43 |
| sbi_remote_sfence_vma_asid | 0 | 0x07 | 0x52464E43 |
| sbi_shutdown | 0 | 0x08 | 0x53525354 |
| **RESERVED** |  | 0x09-0x0F |  |

# Chapter 5. Timer Extension (EID #0x54494D45 "TIME")

This replaces legacy timer extension (EID #0x00). It follows the new calling convention defined in v0.2.

## 5.1. Function: Set Timer (FID #0)

```
struct sbiret sbi_set_timer(uint64_t stime_value)
```

Programs the clock for next event after **stime_value** time. **stime_value** is in absolute time. This function must clear the pending timer interrupt bit as well.

If the supervisor wishes to clear the timer interrupt without scheduling the next timer event, it can either request a timer interrupt infinitely far into the future (i.e., (uint64_t)-1), or it can instead mask the timer interrupt by clearing `sie.STIE` CSR bit.

## 5.2. Function Listing

| Function Name | FID | EID |
|---|---|---|
| sbi_set_timer | 0 | 0x54494D45 |

# Chapter 6. IPI Extension (EID #0x735049 "sPI: s-mode IPI")

This extension replaces the legacy extension (EID #0x04). The other IPI related legacy extension(0x3) is deprecated now. All the functions in this extension follow the `hart_mask` as defined in the binary encoding section.

## 6.1. Function: Send IPI (FID #0)

```
struct sbiret sbi_send_ipi(unsigned long hart_mask,
                           unsigned long hart_mask_base)
```

Send an inter-processor interrupt to all the harts defined in hart_mask. Interprocessor interrupts manifest at the receiving harts as the supervisor software interrupts.

**Returns** following possible values via sbiret.

| Error code | Description |
|---|---|
| SBI_SUCCESS | IPI was sent to all the targeted harts successfully. |

## 6.2. Function Listing

| Function Name | FID | EID |
|---|---|---|
| sbi_send_ipi | 0 | 0x735049 |

# Chapter 7. RFENCE Extension (EID #0x52464E43 "RFNC")

This extension defines all remote fence related functions and replaces the legacy extensions (EIDs #0x05 - #0x07). All the functions follow the `hart_mask` as defined in binary encoding section. Any function wishes to use range of addresses (i.e. start_addr and size), have to abide by the below constraints on range parameters.

The remote fence function acts as a full TLB flush if

- `start_addr` and `size` are both 0
- `size` is equal to 2^XLEN-1

## 7.1. Function: Remote FENCE.I (FID #0)

```
struct sbiret sbi_remote_fence_i(unsigned long hart_mask,
                                 unsigned long hart_mask_base)
```

Instructs remote harts to execute `FENCE.I` instruction.

**Returns** following possible values via sbiret.

| Error code | Description |
|---|---|
| SBI_SUCCESS | IPI was sent to all the targeted harts successfully. |

## 7.2. Function: Remote SFENCE.VMA (FID #1)

```
struct sbiret sbi_remote_sfence_vma(unsigned long hart_mask,
                                    unsigned long hart_mask_base,
                                    unsigned long start_addr,
                                    unsigned long size)
```

Instructs the remote harts to execute one or more `SFENCE.VMA` instructions, covering the range of virtual addresses between start and size.

**Returns** following possible values via sbiret.

| Error code | Description |
|---|---|
| SBI_SUCCESS | IPI was sent to all the targeted harts successfully. |
| SBI_ERR_INVALID_ADDRESS | `start_addr` or `size` is not valid. |

## 7.3. Function: Remote SFENCE.VMA with ASID (FID #2)

```
struct sbiret sbi_remote_sfence_vma_asid(unsigned long hart_mask,
                                         unsigned long hart_mask_base,
                                         unsigned long start_addr,
                                         unsigned long size,
                                         unsigned long asid)
```

Instruct the remote harts to execute one or more `SFENCE.VMA` instructions, covering the range of virtual addresses between start and size. This covers only the given `ASID`.

**Returns** following possible values via sbiret.

| Error code | Description |
|---|---|
| SBI_SUCCESS | IPI was sent to all the targeted harts successfully. |
| SBI_ERR_INVALID_ADDRESS | `start_addr` or `size` is not valid. |

## 7.4. Function: Remote HFENCE.GVMA with VMID (FID #3)

```
struct sbiret sbi_remote_hfence_gvma_vmid(unsigned long hart_mask,
                                          unsigned long hart_mask_base,
                                          unsigned long start_addr,
                                          unsigned long size,
                                          unsigned long vmid)
```

Instruct the remote harts to execute one or more `HFENCE.GVMA` instructions, covering the range of guest physical addresses between start and size only for the given `VMID`. This function call is only valid for harts implementing hypervisor extension.

**Returns** following possible values via sbiret.

| Error code | Description |
|---|---|
| SBI_SUCCESS | IPI was sent to all the targeted harts successfully. |
| SBI_ERR_NOT_SUPPORTED | This function is not supported as it is not implemented or one of the target hart doesn’t support hypervisor extension. |
| SBI_ERR_INVALID_ADDRESS | `start_addr` or `size` is not valid. |

## 7.5. Function: Remote HFENCE.GVMA (FID #4)

```
struct sbiret sbi_remote_hfence_gvma(unsigned long hart_mask,
                                     unsigned long hart_mask_base,
                                     unsigned long start_addr,
                                     unsigned long size)
```

Instruct the remote harts to execute one or more `HFENCE.GVMA` instructions, covering the range of guest physical addresses between start and size for all the guests. This function call is only valid for harts implementing hypervisor extension.

**Returns** following possible values via sbiret.

| Error code | Description |
|---|---|
| SBI_SUCCESS | IPI was sent to all the targeted harts successfully. |
| SBI_ERR_NOT_SUPPORTED | This function is not supported as it is not implemented or one of the target hart doesn't support hypervisor extension. |
| SBI_ERR_INVALID_ADDRESS | `start_addr` or `size` is not valid. |

## 7.6. Function: Remote HFENCE.VVMA with ASID (FID #5)

```
struct sbiret sbi_remote_hfence_vvma_asid(unsigned long hart_mask,
                                          unsigned long hart_mask_base,
                                          unsigned long start_addr,
                                          unsigned long size,
                                          unsigned long asid)
```

Instruct the remote harts to execute one or more `HFENCE.VVMA` instructions, covering the range of guest virtual addresses between start and size for the given `ASID` and current `VMID` (in `hgatp` CSR) of calling hart. This function call is only valid for harts implementing hypervisor extension.

**Returns** following possible values via sbiret.

| Error code | Description |
|---|---|
| SBI_SUCCESS | IPI was sent to all the targeted harts successfully. |
| SBI_ERR_NOT_SUPPORTED | This function is not supported as it is not implemented or one of the target hart doesn't support hypervisor extension. |
| SBI_ERR_INVALID_ADDRESS | `start_addr` or `size` is not valid. |

## 7.7. Function: Remote HFENCE.VVMA (FID #6)

```
struct sbiret sbi_remote_hfence_vvma(unsigned long hart_mask,
                                     unsigned long hart_mask_base,
                                     unsigned long start_addr,
                                     unsigned long size)
```

Instruct the remote harts to execute one or more `HFENCE.VVMA` instructions, covering the range of guest virtual addresses between start and size for current `VMID` (in `hgatp` CSR) of calling hart. This function call is only valid for harts implementing hypervisor extension.

**Returns** following possible values via sbiret.

| Error code | Description |
|---|---|
| SBI_SUCCESS | IPI was sent to all the targeted harts successfully. |
| SBI_ERR_NOT_SUPPORTED | This function is not supported as it is not implemented or one of the target hart doesn't support hypervisor extension. |
| SBI_ERR_INVALID_ADDRESS | `start_addr` or `size` is not valid. |

## 7.8. Function Listing

| Function Name | FID | EID |
|---|---|---|
| sbi_remote_fence_i | 0 | 0x52464E43 |
| sbi_remote_sfence_vma | 1 | 0x52464E43 |
| sbi_remote_sfence_vma_asid | 2 | 0x52464E43 |
| sbi_remote_hfence_gvma_vmid | 3 | 0x52464E43 |
| sbi_remote_hfence_gvma | 4 | 0x52464E43 |
| sbi_remote_hfence_vvma_asid | 5 | 0x52464E43 |
| sbi_remote_hfence_vvma | 6 | 0x52464E43 |

# Chapter 8. Hart State Management Extension (EID #0x48534D "HSM")

The Hart State Management (HSM) Extension introduces a set hart states and a set of functions which allow the supervisor-mode software to request a hart state change.

The Table 1 shown below describes all possible **HSM states** along with a unique **HSM state id** for each state:

| State ID | State Name | Description |
|---|---|---|
| 0 | STARTED | The hart is physically powered-up and executing normally. |
| 1 | STOPPED | The hart is not executing in supervisor-mode or any lower privilege mode. It is probably powered-down by the SBI implementation if the underlying platform has a mechanism to physically power-down harts. |
| 2 | START_PENDING | Some other hart has requested to start (or power-up) the hart from the **STOPPED** state and the SBI implementation is still working to get the hart in the **STARTED** state. |
| 3 | STOP_PENDING | The hart has requested to stop (or power-down) itself from the **STARTED** state and the SBI implementation is still working to get the hart in the **STOPPED** state. |
| 4 | SUSPENDED | This hart is in a platform specific suspend (or low power) state. |
| 5 | SUSPEND_PENDING | The hart has requestd to put itself in a platform specific low power state from the **STARTED** state and the SBI implementation is still working to get the hart in the platform specific **SUSPENDED** state. |
| 6 | RESUME_PENDING | An interrupt or platform specific hardware event has caused the hart to resume normal execution from the **SUSPENDED** state and the SBI implementation is still working to get the hart in the **STARTED** state. |

*Table 1. HSM Hart States*

At any point in time, a hart should be in one of the above mentioned hart states. The hart state transitions by the SBI implementation should follow the state machine shown below in the Figure 3.

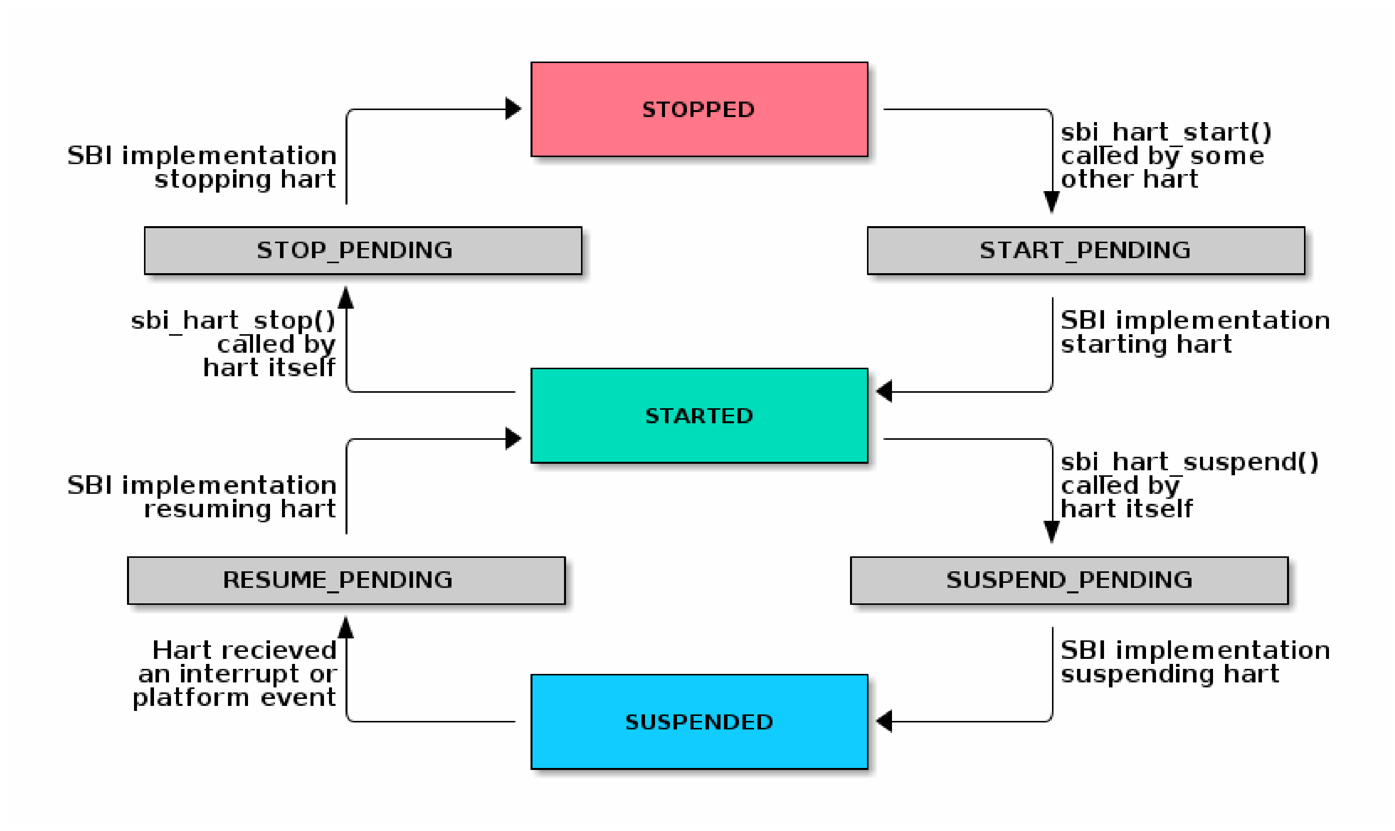
STOPPED
SBI implementation stopping hart
sbi_hart_start() called by some other hart
STOP_PENDING
START_PENDING
sbi_hart_stop() called by hart itself
SBI implementation starting hart
STARTED
SBI implementation resuming hart
sbi_hart_suspend() called by hart itself
RESUME_PENDING
SUSPEND_PENDING
Hart recieved an interrupt or platform event
SBI implementation suspending hart
SUSPENDED

*Figure 3. SBI HSM State Machine*

A platform can have multiple harts grouped into a hierarchical topology groups (namely cores, clusters, nodes, etc) with separate platform specific low-power states for each hierarchical group. These platform specific low-power states of hierarchial topology groups can be represented as platform specific suspend states of a hart. A SBI implementation can utilize the suspend states of higher topology groups using one of the following approaches:

1. **Platform-coordinated:** In this approach, when a hart becomes idle the supervisor-mode power-managment software will request deepest suspend state for the hart and higher topology groups. A SBI implementation should choose a suspend state at higher topology group which is:
    a. Not deeper than the specified suspend state
    b. Wake-up latency is not higher than the wake-up latency of the specified suspend state
2. **OS-inititated:** In this approach, the supervisor-mode power-managment software will directly request a suspend state for higher topology group after the last hart in that group becomes idle. When a hart becomes idle, the supervisor-mode power-managment software will always select suspend state for the hart itself but it will select a suspend state for a higher topology group only if the hart is the last running hart in the group. A SBI implementation should:
    a. Never choose a suspend state for higher topology group different from the specified suspend state
    b. Always prefer most recent suspend state requested for higher topology group

## 8.1. Function: HART start (FID #0)

```
struct sbiret sbi_hart_start(unsigned long hartid,
                             unsigned long start_addr,
                             unsigned long opaque)
```

Request the SBI implementation to start executing the target hart in supervisor-mode at address specified by `start_addr` parameter with specific registers values described in the Table 2 below.

| Register Name | Register Value |
|---|---|
| satp | 0 |
| sstatus.SIE | 0 |
| a0 | hartid |
| a1 | `opaque` parameter |
| All other registers remain in an undefined state. | |

*Table 2. HSM Hart Start Register State*

This call is asynchronous — more specifically, the `sbi_hart_start()` may return before target hart starts executing as long as the SBI implemenation is capable of ensuring the return code is accurate. It is recommended that if the SBI implementation is a platform runtime firmware executing in machine-mode (M-mode) then it MUST configure PMP and other M-mode state before executing in supervisor-mode.

The `hartid` parameter specifies the target hart which is to be started.

The `start_addr` parameter points to a runtime-specified physical address, where the hart can start executing in supervisor-mode.

The `opaque` parameter is a XLEN-bit value which will be set in the `a1` register when the hart starts executing at `start_addr`.

The possible error codes returned in `sbiret.error` are shown in the Table 3 below.

| Error code | Description |
|---|---|
| SBI_SUCCESS | Hart was previously in stopped state. It will start executing from `start_addr`. |
| SBI_ERR_INVALID_ADDRESS | `start_addr` is not valid possibly due to following reasons:<br>* It is not a valid physical address.<br>* The address is prohibited by PMP to run in supervisor mode. |
| SBI_ERR_INVALID_PARAM | `hartid` is not a valid hartid as corresponding hart cannot started in supervisor mode. |

| Error code | Description |
|---|---|
| SBI_ERR_ALREADY_AVAILABLE | The given hartid is already started. |
| SBI_ERR_FAILED | The start request failed for unknown reasons. |

*Table 3. HSM Hart Start Errors*

## 8.2. Function: HART stop (FID #1)

```
struct sbiret sbi_hart_stop(void)
```

Request the SBI implementation to stop executing the calling hart in supervisor-mode and return it's ownership to the SBI implementation. This call is not expected to return under normal conditions. The `sbi_hart_stop()` must be called with the supervisor-mode interrupts disabled.

The possible error codes returned in `sbiret.error` are shown in the Table 4 below.

| Error code | Description |
|---|---|
| SBI_ERR_FAILED | Failed to stop execution of the current hart |

*Table 4. HSM Hart Stop Errors*

## 8.3. Function: HART get status (FID #2)

```
struct sbiret sbi_hart_get_status(unsigned long hartid)
```

Get the current status (or HSM state id) of the given hart in `sbiret.value`, or an error through `sbiret.error`.

The `hartid` parameter specifies the target hart for which status is required.

The possible status (or HSM state id) values returned in `sbiret.value` are described in Table 1.

The possible error codes returned in `sbiret.error` are shown in the Table 5 below.

| Error code | Description |
|---|---|
| SBI_ERR_INVALID_PARAM | The given `hartid` or `start_addr` is not valid |

*Table 5. HSM Hart Get Status Errors*

The harts may transition HSM states at any time due to any concurrent `sbi_hart_start()` or `sbi_hart_stop()` or `sbi_hart_suspend()` calls, the return value from this function may not represent the actual state of the hart at the time of return value verification.

## 8.4. Function: HART suspend (FID #3)

```
struct sbiret sbi_hart_suspend(uint32_t suspend_type,
                               unsigned long resume_addr,
                               unsigned long opaque)
```

Request the SBI implementation to put the calling hart in a platform specfic suspend (or low power) state specified by the `suspend_type` parameter. The hart will automatically come out of suspended state and resume normal execution when it recieves an interrupt or platform specific hardware event.

The platform specific suspend states for a hart can be either retentive or non-rententive in nature. A retentive suspend state will preserve hart register and CSR values for all privilege modes whereas a non-retentive suspend state will not preserve hart register and CSR values.

Resuming from a retentive suspend state is straight forward and the supervisor-mode software will see SBI suspend call return without any failures.

Resuming from a non-retentive suspend state is relatively more involved and requires software to restore various hart registers and CSRs for all privilege modes. Upon resuming from non-retentive suspend state, the hart will jump to supervisor-mode at address specified by `resume_addr` with specific registers values described in the Table 6 below.

| Register Name | Register Value |
|---|---|
| satp | 0 |
| sstatus.SIE | 0 |
| a0 | hartid |
| a1 | `opaque` parameter |
| All other registers remain in an undefined state. | |

*Table 6. HSM Hart Resume Register State*

The `suspend_type` parameter is 32 bits wide and the possible values are shown in Table 7 below.

| Value | Description |
|---|---|
| 0x00000000 | Default retentive suspend |
| 0x00000001 - 0x0FFFFFFF | Reserved for future use |
| 0x10000000 - 0x7FFFFFFF | Platform specific retentive suspend |
| 0x80000000 | Default non-retentive suspend |
| 0x80000001 - 0x8FFFFFFF | Reserved for future use |
| 0x90000000 - 0xFFFFFFFF | Platform specific non-retentive suspend |
| > 0xFFFFFFFF | Reserved (and non-existent on RV32) |

*Table 7. HSM Hart Suspend Types*

The `resume_addr` parameter points to a runtime-specified physical address, where the hart can resume execution in supervisor-mode after a non-retentive suspend.

The `opaque` parameter is a XLEN-bit value which will be set in the `a1` register when the hart resumes exectution at `resume_addr` after a non-retentive suspend.

The possible error codes returned in `sbiret.error` are shown in the Table 8 below.

| Error code | Description |
|---|---|
| SBI_SUCCESS | Hart has suspended and resumed back successfully from a retentive suspend state. |
| SBI_ERR_INVALID_PARAM | `suspend_type` is not valid. |
| SBI_ERR_NOT_SUPPORTED | `suspend_type` is valid but not implemented. |
| SBI_ERR_INVALID_ADDRESS | `resume_addr` is not valid possibly due to following reasons:<br>* It is not a valid physical address.<br>* The address is prohibited by PMP to run in supervisor mode. |
| SBI_ERR_FAILED | The suspend request failed for unknown reasons. |

*Table 8. HSM Hart Suspend Errors*

## 8.5. Function Listing

| Function Name | SBI Version | FID | EID |
|---|---|---|---|
| sbi_hart_start | 0.2 | 0 | 0x48534D |
| sbi_hart_stop | 0.2 | 1 | 0x48534D |
| sbi_hart_get_status | 0.2 | 2 | 0x48534D |
| sbi_hart_suspend | 0.3 | 3 | 0x48534D |

*Table 9. HSM Function List*

# Chapter 9. System Reset Extension (EID #0x53525354 "SRST")

The System Reset Extension provides a function that allow the supervisor software to request system-level reboot or shutdown. The term "system" refers to the world-view of supervisor software and the underlying SBI implementation could be machine mode firmware or hypervisor.

## 9.1. Function: System reset (FID #0)

```
struct sbiret sbi_system_reset(unsigned long reset_type,
                               unsigned long reset_reason)
```

Reset the system based on provided `reset_type` and `reset_reason`. This is a synchronous call and does not return if it succeeds.

The `reset_type` parameter is 32 bits wide and has the following possible values:

| Value | Description |
|---|---|
| 0x00000000 | Shutdown |
| 0x00000001 | Cold reboot |
| 0x00000002 | Warm reboot |
| 0x00000003 - 0xEFFFFFFF | Reserved for future use |
| 0xF0000000 - 0xFFFFFFFF | Vendor or platform specific reset type |
| > 0xFFFFFFFF | Reserved (and non-existent on RV32) |

The `reset_reason` is an optional parameter representing the reason for system reset. This parameter is 32 bits wide and has the following possible values:

| Value | Description |
|---|---|
| 0x00000000 | No reason |
| 0x00000001 | System failure |
| 0x00000002 - 0xDFFFFFFF | Reserved for future use |
| 0xE0000000 - 0xEFFFFFFF | SBI implementation specific reset reason |
| 0xF0000000 - 0xFFFFFFFF | Vendor or platform specific reset reason |
| > 0xFFFFFFFF | Reserved (and non-existent on RV32) |

When supervisor software is running natively, the SBI implementation is machine mode firmware. In this case, shutdown is equivalent to physical power down of the entire system and cold reboot is equivalent to physical power cycle of the entire system. Further, warm reboot is equivalent to a power cycle of main processor and parts of the system but not the entire system. For example, on a server class system with a BMC (board management controller), a warm reboot will not power cycle the BMC whereas a cold reboot will definitely power cycle the BMC.

When supervisor software is running inside a virtual machine, the SBI implementation is a hypervisor. The shutdown, cold reboot and warm reboot will behave functionally the same as the native case but might not result in any physical power changes.

**Returns** one of the following possible SBI error codes through sbiret.error upon failure.

| Error code | Description |
|---|---|
| SBI_ERR_INVALID_PARAM | `reset_type` or `reset_reason` is not valid. |
| SBI_ERR_NOT_SUPPORTED | `reset_type` is valid but not implemented. |
| SBI_ERR_FAILED | Reset request failed for unknown reasons. |

## 9.2. Function Listing

| Function Name | FID | EID |
|---|---|---|
| sbi_system_reset | 0 | 0x53525354 |

# Chapter 10. Experimental SBI Extension Space (EIDs #0x08000000 - #0x08FFFFFF)

No management.

# Chapter 11. Vendor-Specific SBI Extension Space (EIDs #0x09000000 - #0x09FFFFFF)

Low bits from `mvendorid`.

# Chapter 12. Firmware Specific SBI Extension Space (EIDs #0x0A000000 - #0x0AFFFFFF)

Low bits is SBI implementation ID. The firmware specific SBI extensions are for SBI implementations. It provides firmware specific SBI functions which are defined in the external firmware specification.