RISC-V Supervisor Binary Interface Specification

RISC-V Platform Runtime Services Task Group

Version v3.0-rc3, 2025-01-06: This document is under development. Expect potential changes.

Table of Contents

Preamble

This document is in the Development state

Expect potential changes.

Copyright and license information

Contributors

This RISC-V specification has been contributed to directly or indirectly by:

Abner Chang <abner.chang@hpe.com>
Al Stone <ahs3@ahs3.net>
Andrew Jones <ajones@ventanamicro.com>
Anup Patel <apatel@ventanamicro.com>
Atish Patra <atishp04@gmail.com>
Atish Patra <atishp@rivosinc.com>
Bin Meng <bmeng.cn@gmail.com>
Chris Williams <diodesign@tuta.io>
Clément Léger <cleger@rivosinc.com>
Conor Dooley <conor.dooley@microchip.com>
Daniel Schaefer <git@danielschaefer.me>
Esteban Blanc <estblcsk@gmail.com>
hasheddan <georgedanielmangum@gmail.com>
Heinrich Schuchardt <xypron.glpk@gmx.de>
Jeff Scheel <jeff@riscv.org>
Jessica Clarke <jrtc27@jrtc27.com>
john <799433746@qq.com>
Konrad Schwarz <konrad.schwarz@siemens.com>
Luo Jia / Zhouqi Jiang <luojia@hust.edu.cn>
Nick Kossifidis <mickflemm@gmail.com>
Palmer Dabbelt <palmer@dabbelt.com>
Paolo Bonzini <pbonzini@redhat.com>
Rahul Pathak <rpathak@ventanamicro.com>
Samuel Holland <samuel.holland@sifive.com>
Sean Anderson <seanga2@gmail.com>
Stefano Stabellini <stefano.stabellini@amd.com>
Sunil V L <sunilvl@ventanamicro.com>
Tsukasa OI <research_trasio@irq.a4lg.com>
Yiting Wang <yiting.wang@windriver.com>

Change Log

Version 3.0-rc3

- Added low priority RAS events in SSE.
- Miscallenous clarification around reserved bits, fwft, notification events.
- Added a dedicated error code for fwft set denial due to lock status.

Version 3.0-rc1/rc2

- Added SBI PMU event info function and new raw event type
- Added SBI MPXY extension
- Added error code SBI_ERR_TIMEOUT
- Added error code SBI_ERR_IO
- Added sse mask/unmask function and pointer masking bit in fwft
- Clarify SBI IPI and RFENCE error codes
- Clarify the description of the `set_timer` function
- Added SBI DBTR extension
- Added SBI FWFT extension
- Added SBI SSE extension
- Added error code SBI_ERR_BAD_RANGE
- Added error code SBI_ERR_INVALID_STATE

Version 2.0

- Clarification around SBI PMU set memory function
- Base extension function name typo fix
- Upate the document state to Ratified

Version 2.0-rc8

- Clarfications STA extension and counter index in the pmu snapshot.

Version 2.0-rc7

- Few clarfications around system suspend and pmu snapshot.

Version 2.0-rc6

- Few clarifications around rfence extensions
- Marks public review period complete.

Version 2.0-rc5

- Update the document state to Frozen

Version 2.0-rc4

- Added flags parameter to sbi_pmu_snapshot_set_shmem()
- Return error code SBI_ERR_NO_SHMEM in SBI PMU extension wherever applicable
- Made flags parameter of sbi_steal_time_set_shmem() as unsigned long
- Split the specification into multiple adoc files
- Add more clarification for firmware/vendor/experimental extension space.
- Fix ambiguous usage of normative statements.

Version 2.0-rc3

- CI support added
- Fix revmark in the makefile.
- Few minor cleanups.

Version 2.0-rc2

- Added clarification for SUSP, NACL & STA extensions.
- Standardization of hart usage.
- Added an error code in SBI DBCN extension.

Version 2.0-rc1

- Added common description for shared memory physical address range parameter
- Added SBI debug console extension
- Relaxed the counter width requirement on SBI PMU firmware counters
- Added sbi_pmu_counter_fw_read_hi() in SBI PMU extension
- Reserved space for SBI implementation specific firmware events
- Added SBI system suspend extension
- Added SBI CPPC extension
- Clarified that an SBI extension can be partially implemented only if it defines a mechanism to discover implemented SBI functions
- Added error code SBI_ERR_NO_SHMEM
- Added SBI nested acceleration extension
- Added common description for a virtual hart
- Added SBI steal-time accounting extension

- Added SBI PMU snapshot extension

Version 1.0.0

- Updated the version for ratification

Version 1.0-rc3

- Updated the calling convention
- Fixed a typo in PMU extension
- Added a abbreviation table

Version 1.0-rc2

- Update to RISC-V formatting
- Improved the introduction
- Removed all references to RV32

Version 1.0-rc1

- A typo fix

Version 0.3.0

- Few typo fixes
- Updated the LICENSE with detailed text instead of a hyperlink

Version 0.3-rc1

- Improved document styling and naming conventions
- Added SBI system reset extension
- Improved SBI introduction section
- Improved documentation of SBI hart state management extension
- Added suspend function to SBI hart state management extension
- Added performance monitoring unit extension
- Clarified that an SBI extension shall not be partially implemented

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.

An SBI extension defines a set of SBI functions which provides a particular functionality to supervisor-mode software. SBI extensions as a whole are optional and cannot be partially implemented unless an SBI extension defines a mechanism to discover implemented SBI functions. If sbi_probe_extension() signals that an extension is available, all functions present in the SBI version reported by sbi_get_spec_version() must conform to that version of the SBI specification.

The higher privilege software providing SBI interface to the supervisor-mode software is referred as an SBI implementation or Supervisor Execution Environment (SEE). An 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).

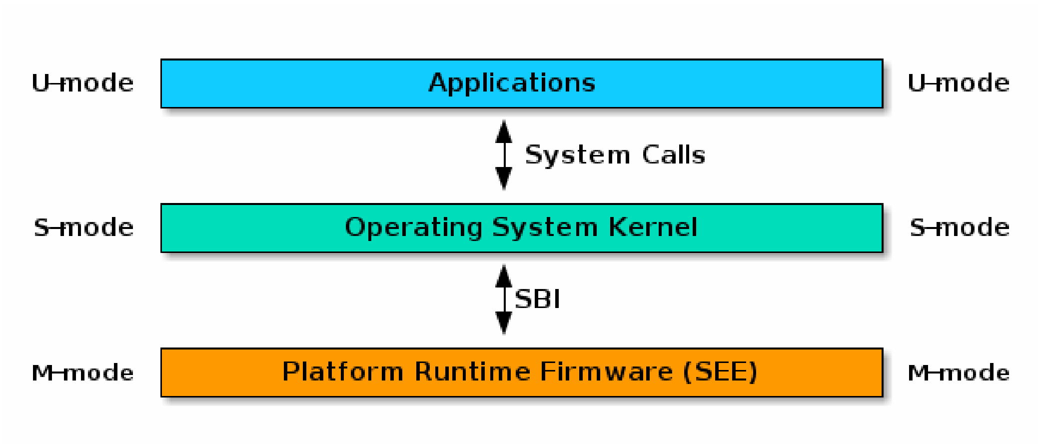

Figure 1. RISC-V System without H-extension

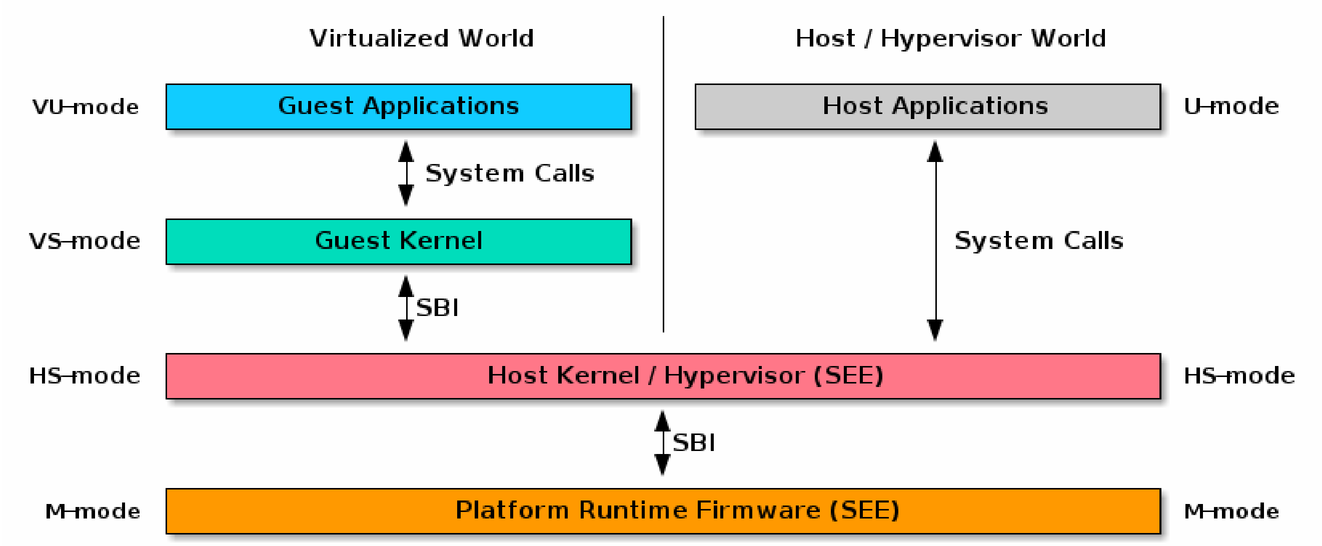

Figure 2. RISC-V System with H-extension

Harts are provisioned by the SBI implementation for supervisor-mode software. Hence, from the perspective of the SBI implementation, the S-mode hart contexts are referred to as virtual harts. In the case that the implementation is a hypervisor, virtual harts represent the VS-mode guest contexts.

The SBI specification doesn't specify any method for hardware discovery. The supervisor software must rely on the other industry standard hardware discovery methods (i.e. Device Tree or ACPI) for that.

Chapter 2. Terms and Abbreviations

This specification uses the following terms and abbreviations:

Term	Meaning
ACPI	Advanced Configuration and Power Interface
ASID	Address Space Identifier
BMC	Baseboard Management Controller
CPPC	Collaborative Processor Performance Control
EID	Extension ID
FID	Function ID
HSM	Hart State Management
IPI	Inter Processor Interrupt
PMU	Performance Monitoring Unit
SBI	Supervisor Binary Interface
SEE	Supervisor Execution Environment
VMID	Virtual Machine Identifier

Chapter 3. Binary Encoding

All SBI functions share a single binary encoding, which facilitates the mixing of SBI extensions. The SBI specification follows the below calling convention.

- An `ECALL` is used as the control transfer instruction between the supervisor and the SEE.
- `a7` encodes the SBI extension ID (**EID**).
- `a6` encodes the SBI function ID (**FID**) for a given extension ID encoded in `a7` for any SBI extension defined in or after SBI v0.2.
- `a0` through `a5` contain the arguments for the SBI function call. Registers that are not defined in the SBI function call are not reserved.
- All registers except `a0` & `a1` must be preserved across an SBI call by the callee.
- 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

```c
    struct sbiret {
        long error;
        union {
            long value;
            unsigned long uvalue;
        };
    };
```

Data type `long` in C pseudocode is XLEN bits wide.

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 above calling convention rules.

The Table 1 below provides a list of Standard SBI error codes.

Table 1. Standard SBI Errors

Error Type	Value	Description
SBI_SUCCESS	0	Completed successfully
SBI_ERR_FAILED	-1	Failed
SBI_ERR_NOT_SUPPORTED	-2	Not supported
SBI_ERR_INVALID_PARAM	-3	Invalid parameter(s)
SBI_ERR_DENIED	-4	Denied or not allowed
SBI_ERR_INVALID_ADDRESS	-5	Invalid address(s)
SBI_ERR_ALREADY_AVAILABLE	-6	Already available
SBI_ERR_ALREADY_STARTED	-7	Already started
SBI_ERR_ALREADY_STOPPED	-8	Already stopped
SBI_ERR_NO_SHMEM	-9	Shared memory not available

Error Type	Value	Description
SBI_ERR_INVALID_STATE	-10	Invalid state
SBI_ERR_BAD_RANGE	-11	Bad (or invalid) range
SBI_ERR_TIMEOUT	-12	Failed due to timeout
SBI_ERR_IO	-13	Input/Output error
SBI_ERR_DENIED_LOCKED	-14	Denied or not allowed due to lock status

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`.

If an SBI function call returns an error code other than `SBI_SUCCESS`, the value returned in `a1` is unspecified unless explicitly defined for that SBI function.

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. In case the data is defined as 32bit wide, higher privilege software must ensure that it only uses 32 bit data. Parameters that are 2×XLEN bits wide are passed in a pair of argument registers, with the low-order XLEN bits in the lower-numbered register and the high-order XLEN bits in the higher-numbered register.

3.1. Hart list parameter

If an SBI function caller 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 SBI function, requiring a hart mask, must take the 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 the bit-vector must be computed.

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

Any SBI function taking hart mask arguments may return the error values listed in the Table 2 below which are in addition to function specific error values.

Table 2. Hart Mask Errors

Error code	Description
SBI_ERR_INVALID_PARAM	Either `hart_mask_base`, or at least one 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.

3.2. Shared memory physical address range parameter

If an SBI function needs to pass a shared memory physical address range to the SBI implementation (or higher privilege mode), then this physical memory address range MUST satisfy the following requirements:

- The SBI implementation MUST check that the specified physical memory range is composed of accessible physical addresses and return SBI_ERR_INVALID_ADDRESS when any address in the range is not accessible.

> *An accessible address is one that S-mode could reasonably expect to access per its description of the platform's physical memory layout. As an SBI implementation may further restrict the allowed range, it may return a generic SBI_ERR_FAILED (instead of SBI_ERR_INVALID_ADDRESS) when input is inaccessible with respect to its specific limits. Returning SBI_ERR_FAILED instead of SBI_ERR_INVALID_ADDRESS, in this case, is not a violation of the above specification because the SBI implementation should detect the distinct case of violating the more strict range first, making it appropriate to return the error associated with the stricter range case immediately.*

- The SBI implementation MUST check that the supervisor-mode software is allowed to access the specified physical memory range with the access type requested (read and/or write).
- The SBI implementation MUST access the specified physical memory range using the PMA attributes.

> *If the supervisor-mode software accesses the same physical memory range using a memory type different than the PMA, then a loss of coherence or unexpected memory ordering may occur. The invoking software should follow the rules and sequences defined in the RISC-V Svpbmt specification to prevent the loss of coherence and memory ordering.*

- The data in the shared memory MUST follow little-endian byte ordering.

It is recommended that a memory physical address passed to an SBI function should use at least two `unsigned long` parameters to support platforms which have memory physical addresses wider than XLEN bits.

Chapter 4. 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.

4.1. 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. When XLEN is greater than 32, bits 32 and above are also reserved and must be `0`.

4.2. 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.

4.3. 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.

4.4. 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 1 if it is available unless defined as any other non-zero value by the implementation.

4.5. 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.

4.6. 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.

4.7. 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.

4.8. Function Listing

Table 3. Base Function List

Function Name	SBI Version	FID	EID
sbi_get_spec_version	0.2	0	0x10
sbi_get_impl_id	0.2	1	0x10
sbi_get_impl_version	0.2	2	0x10
sbi_probe_extension	0.2	3	0x10
sbi_get_mvendorid	0.2	4	0x10
sbi_get_marchid	0.2	5	0x10
sbi_get_mimpid	0.2	6	0x10

4.9. SBI Implementation IDs

Table 4. SBI Implementation IDs

Implementation ID	Name
0	Berkeley Boot Loader (BBL)
1	OpenSBI
2	Xvisor
3	KVM
4	RustSBI
5	Diosix
6	Coffer
7	Xen Project

Implementation ID	Name
8	PolarFire Hart Software Services
9	coreboot
10	oreboot
11	bhyve

Chapter 5. Legacy Extensions (EIDs #0x00 - #0x0F)

The legacy SBI extensions follow a slightly different calling convention as compared to the SBI v0.2 (or higher) specification where:

- The SBI function ID field in `a6` register is ignored because these are encoded as multiple SBI extension IDs.
- Nothing is returned in `a1` register.
- All registers except `a0` must be preserved across an SBI call by the callee.
- The value returned in `a0` register is SBI legacy extension specific.

The page and access faults taken by the SBI implementation while accessing memory on behalf of the supervisor are redirected back to the supervisor with `sepc` CSR pointing to the faulting `ECALL` instruction.

The legacy SBI extensions is deprecated in favor of the other extensions listed below.

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

```
long 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.

This SBI call returns 0 upon success or an implementation specific negative error code.

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

```
long 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.

This SBI call returns 0 upon success or an implementation specific negative error code.

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

```
long sbi_console_getchar(void)
```

Read a byte from debug console.

The SBI call returns the byte on success, or -1 for failure.

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

```
long 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.

This SBI call returns `0` if no IPI had been pending, or an implementation specific positive value if an IPI had been pending.

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

```
long 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.

This SBI call returns `0` upon success or an implementation specific negative error code.

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

```
long 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()`.

This SBI call returns `0` upon success or an implementation specific negative error code.

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

```
long 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 `start + size`.

The remote fence operation applies to the entire address space if either:

- `start` and `size` are both `0`, or
- `size` is equal to 2^XLEN-1.

This SBI call returns `0` upon success or an implementation specific negative error code.

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

```
long 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 `start + size`. This covers only the given `ASID`.

The remote fence operation applies to the entire address space if either:

- `start` and `size` are both `0`, or
- `size` is equal to 2^XLEN-1.

This SBI call returns `0` upon success or an implementation specific negative error code.

5.9. 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 irrespective whether it succeeds or fails.

5.10. Function Listing

Table 5. Legacy Function List

Function Name	SBI Version	FID	EID	Replacement EID
sbi_set_timer	0.1	0	0x00	0x54494D45
sbi_console_putchar	0.1	0	0x01	0x4442434E
sbi_console_getchar	0.1	0	0x02	0x4442434E
sbi_clear_ipi	0.1	0	0x03	N/A
sbi_send_ipi	0.1	0	0x04	0x735049

Function Name	SBI Version	FID	EID	Replacement EID
sbi_remote_fence_i	0.1	0	0x05	0x52464E43
sbi_remote_sfence_vma	0.1	0	0x06	0x52464E43
sbi_remote_sfence_vma_asid	0.1	0	0x07	0x52464E43
sbi_shutdown	0.1	0	0x08	0x53525354
RESERVED			0x09-0x0F	

Chapter 6. Timer Extension (EID #0x54494D45 "TIME")

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

6.1. 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.

If the supervisor wishes to clear the timer interrupt without scheduling the next timer event, it may request a timer interrupt infinitely far into the future (i.e., (uint64_t)-1). Alternatively, to not receive timer interrupts, it may mask timer interrupts by clearing the `sie.STIE` CSR bit.

This function must clear the pending timer interrupt bit when **stime_value** is set to some time in the future, regardless of whether timer interrupts are masked or not.

This function always returns SBI_SUCCESS in `sbiret.error`.

6.2. Function Listing

Table 6. TIME Function List

Function Name	SBI Version	FID	EID
sbi_set_timer	0.2	0	0x54494D45

Chapter 7. 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.

7.1. 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.

The possible error codes returned in `sbiret.error` are shown in the Table 7 below.

Table 7. IPI Send Errors

Error code	Description
SBI_SUCCESS	IPI was sent to all the targeted harts successfully.
SBI_ERR_INVALID_PARAM	Either `hart_mask_base` or at least one 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.
SBI_ERR_FAILED	The request failed for unspecified or unknown other reasons.

7.2. Function Listing

Table 8. IPI Function List

Function Name	SBI Version	FID	EID
sbi_send_ipi	0.2	0	0x735049

Chapter 8. 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 which accepts a range of addresses (i.e. `start_addr` and `size`) must abide by the below constraints on range parameters.

The remote fence operation applies to the entire address space if either:

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

8.1. 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.

The possible error codes returned in `sbiret.error` are shown in the Table 9 below.

Table 9. RFENCE Remote FENCE.I Errors

Error code	Description
SBI_SUCCESS	IPI was sent to all the targeted harts successfully.
SBI_ERR_INVALID_PARAM	Either `hart_mask_base` or at least one 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.
SBI_ERR_FAILED	The request failed for unspecified or unknown other reasons.

8.2. 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_addr` and `start_addr + size`.

The possible error codes returned in `sbiret.error` are shown in the Table 10 below.

Table 10. RFENCE Remote SFENCE.VMA Errors

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.
SBI_ERR_INVALID_PARAM	Either `hart_mask_base` or at least one 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.
SBI_ERR_FAILED	The request failed for unspecified or unknown other reasons.

8.3. 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_addr` and `start_addr + size`. This covers only the given `ASID`.

The possible error codes returned in `sbiret.error` are shown in the Table 11 below.

Table 11. RFENCE Remote SFENCE.VMA with ASID Errors

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.
SBI_ERR_INVALID_PARAM	Either `asid`, `hart_mask_base`, or at least one 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.
SBI_ERR_FAILED	The request failed for unspecified or unknown other reasons.

8.4. 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_addr` and `start_addr + size` only for the given `VMID`. This function call is only valid for harts implementing hypervisor extension.

The possible error codes returned in `sbiret.error` are shown in the Table 12 below.

Table 12. RFENCE Remote HFENCE.GVMA with VMID Errors

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.
SBI_ERR_INVALID_PARAM	Either `vmid`, `hart_mask_base`, or at least one 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.
SBI_ERR_FAILED	The request failed for unspecified or unknown other reasons.

8.5. 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_addr` and `start_addr + size` for all the guests. This function call is only valid for harts implementing hypervisor extension.

The possible error codes returned in `sbiret.error` are shown in the Table 13 below.

Table 13. RFENCE Remote HFENCE.GVMA Errors

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.
SBI_ERR_INVALID_PARAM	Either `hart_mask_base` or at least one 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.
SBI_ERR_FAILED	The request failed for unspecified or unknown other reasons.

8.6. 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_addr` and `start_addr + 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.

The possible error codes returned in `sbiret.error` are shown in the Table 14 below.

Table 14. RFENCE Remote HFENCE.VVMA with ASID Errors

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.
SBI_ERR_INVALID_PARAM	Either `asid`, `hart_mask_base`, or at least one 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.
SBI_ERR_FAILED	The request failed for unspecified or unknown other reasons.

8.7. 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_addr` and `start_addr + size` for current `VMID` (in `hgatp` CSR) of calling hart. This function call is only valid for harts implementing hypervisor extension.

The possible error codes returned in `sbiret.error` are shown in the Table 15 below.

Table 15. RFENCE Remote HFENCE.VVMA Errors

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.
SBI_ERR_INVALID_PARAM	Either `hart_mask_base` or at least one 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.

Error code	Description
SBI_ERR_FAILED	The request failed for unspecified or unknown other reasons.

8.8. Function Listing

Table 16. RFENCE Function List

Function Name	SBI Version	FID	EID
sbi_remote_fence_i	0.2	0	0x52464E43
sbi_remote_sfence_vma	0.2	1	0x52464E43
sbi_remote_sfence_vma_asid	0.2	2	0x52464E43
sbi_remote_hfence_gvma_vmid	0.2	3	0x52464E43
sbi_remote_hfence_gvma	0.2	4	0x52464E43
sbi_remote_hfence_vvma_asid	0.2	5	0x52464E43
sbi_remote_hfence_vvma	0.2	6	0x52464E43

Chapter 9. Hart State Management Extension (EID #0x48534D "HSM")

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

The Table 17 shown below describes all possible **HSM states** along with a unique **HSM state id** for each state:

Table 17. HSM Hart States

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 requested 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.

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.

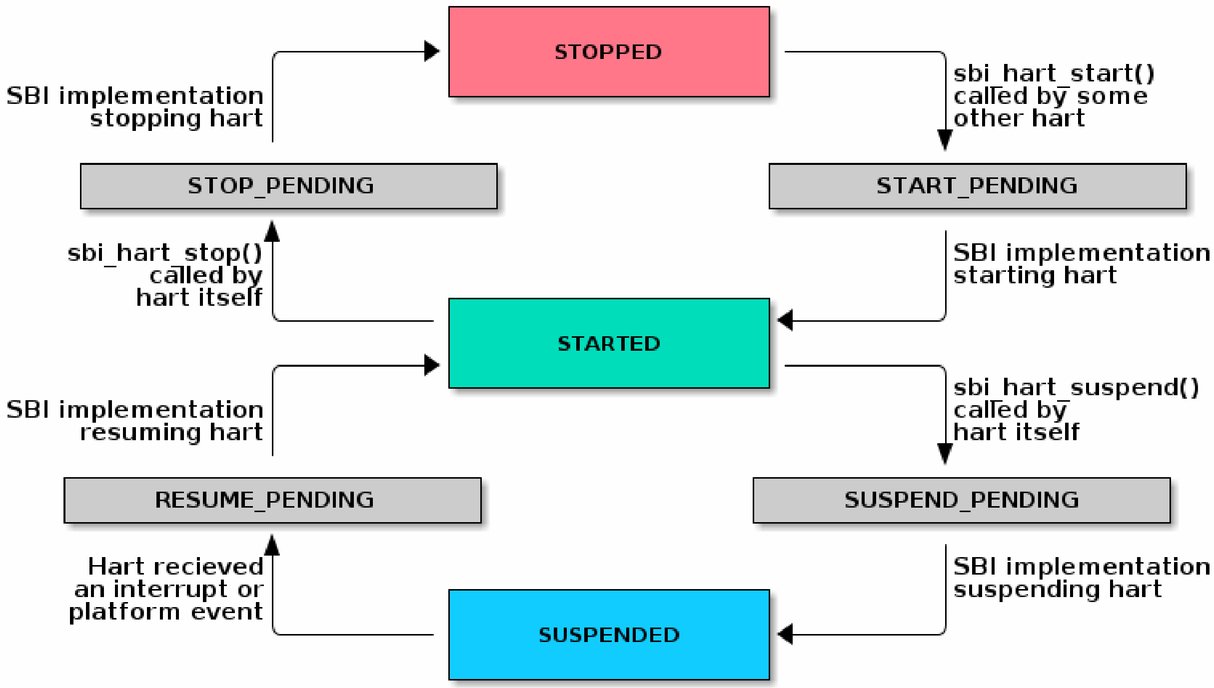

Figure 3. SBI HSM State Machine

A platform can have multiple harts grouped into 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 hierarchical topology groups can be represented as platform specific suspend states of a hart. An 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. An 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. An 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

9.1. 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 the address specified by `start_addr`, with the specific register values described in Table 18.

Table 18. HSM Hart Start Register State

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

> *A single `unsigned long` parameter is sufficient as `start_addr`, because the hart will start execution in supervisor-mode with the MMU off, hence `start_addr` must be less than XLEN bits wide.*

This call is asynchronous — more specifically, the `sbi_hart_start()` may return before the target hart starts executing as long as the SBI implementation is capable of ensuring the return code is accurate. If the SBI implementation is a platform runtime firmware executing in machine-mode (M-mode), then it MUST configure any physical memory protection it supports, such as that defined by PMP, and other M-mode state, before transferring control to supervisor-mode software.

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 an 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 19 below.

Table 19. HSM Hart Start Errors

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 the following reasons: * It is not a valid physical address. * Executable access to the address is prohibited by a physical memory protection mechanism or H-extension G-stage for supervisor-mode.
SBI_ERR_INVALID_PARAM	`hartid` is not a valid hartid as the corresponding hart cannot be started in supervisor mode.
SBI_ERR_ALREADY_AVAILABLE	The given hartid is already started.
SBI_ERR_FAILED	The start request failed for unspecified or unknown other reasons.

9.2. 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 its ownership to the SBI implementation. This call is not expected to return under normal conditions. The `sbi_hart_stop()` must be called with supervisor-mode interrupts disabled.

The possible error codes returned in `sbiret.error` are shown in the Table 20 below.

Table 20. HSM Hart Stop Errors

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

9.3. 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 17.

The possible error codes returned in `sbiret.error` are shown in the Table 21 below.

Table 21. HSM Hart Get Status Errors

Error code	Description
SBI_ERR_INVALID_PARAM	The given `hartid` is not valid.

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.

9.4. 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 specific 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 receives an interrupt or platform specific hardware event.

The platform specific suspend states for a hart can be either retentive or non-retentive 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. The `resume_addr` parameter is unused during retentive suspend.

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 22 below.

Table 22. HSM Hart Resume Register State

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

> *A single `unsigned long` parameter is sufficient for `resume_addr`, because the hart will resume execution in supervisor-mode with the MMU off, hence `resume_addr` must be less than XLEN bits wide.*

The `suspend_type` parameter is 32 bits wide and the possible values are shown in Table 23 below.

Table 23. HSM Hart Suspend Types

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

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 an XLEN-bit value which will be set in the `a1` register when the hart resumes execution at `resume_addr` after a non-retentive suspend.

The possible error codes returned in `sbiret.error` are shown in the Table 24 below.

Table 24. HSM Hart Suspend Errors

Error code	Description
SBI_SUCCESS	Hart has suspended and resumed successfully from a retentive suspend state.
SBI_ERR_INVALID_PARAM	`suspend_type` is reserved or is platform-specific and unimplemented.
SBI_ERR_NOT_SUPPORTED	`suspend_type` is not reserved and is implemented, but the platform does not support it due to one or more missing dependencies.
SBI_ERR_INVALID_ADDRESS	`resume_addr` is not valid, possibly due to the following reasons: * It is not a valid physical address. * Executable access to the address is prohibited by a physical memory protection mechanism or H-extension G-stage for supervisor-mode.
SBI_ERR_FAILED	The suspend request failed for unspecified or unknown other reasons.

9.5. Function Listing

Table 25. HSM Function List

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

Chapter 10. 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 provided by machine mode firmware or a hypervisor.

10.1. Function: System reset (FID #0)

```
struct sbiret sbi_system_reset(uint32_t reset_type, uint32_t 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 it's possible values are shown in the Table 26 below.

Table 26. SRST System Reset Types

Value	Description
0x00000000	Shutdown
0x00000001	Cold reboot
0x00000002	Warm reboot
0x00000003 - 0xEFFFFFFF	Reserved for future use
0xF0000000 - 0xFFFFFFFF	Vendor or platform specific reset type

The `reset_reason` is an optional parameter representing the reason for system reset. This parameter is 32 bits wide with possible values shown in the Table 27 below

Table 27. SRST System Reset Reasons

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

When supervisor software is running natively, the SBI implementation is provided by machine mode firmware. In this case, shutdown is equivalent to a physical power down of the entire system and cold reboot is equivalent to a physical power cycle of the entire system. Further, warm reboot is equivalent to a power cycle of the 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 provided by a hypervisor. Shutdown, cold reboot and warm reboot will behave functionally the same as the native case, but might not result in any physical power changes.

The possible error codes returned in `sbiret.error` are shown in the Table 28 below.

Table 28. SRST System Reset Errors

Error code	Description
SBI_ERR_INVALID_PARAM	At least one of `reset_type` or `reset_reason` is reserved or is platform-specific and unimplemented.
SBI_ERR_NOT_SUPPORTED	`reset_type` is not reserved and is implemented, but the platform does not support it due to one or more missing dependencies.
SBI_ERR_FAILED	The reset request failed for unspecified or unknown other reasons.

10.2. Function Listing

Table 29. SRST Function List

Function Name	SBI Version	FID	EID
sbi_system_reset	0.3	0	0x53525354

Chapter 11. Performance Monitoring Unit Extension (EID #0x504D55 "PMU")

The RISC-V hardware performance counters such as `mcycle`, `minstret`, and `mhpmcounterX` CSRs are accessible as read-only from supervisor-mode using `cycle`, `instret`, and `hpmcounterX` CSRs. The SBI performance monitoring unit (PMU) extension is an interface for supervisor-mode to configure and use the RISC-V hardware performance counters with assistance from the machine-mode (or hypervisor-mode). These hardware performance counters can only be started, stopped, or configured from machine-mode using `mcountinhibit` and `mhpmeventX` CSRs. Due to this, a machine-mode SBI implementation may choose to disallow SBI PMU extension if `mcountinhibit` CSR is not implemented by the RISC-V platform.

A RISC-V platform generally supports monitoring of various hardware events using a limited number of hardware performance counters which are up to 64 bits wide. In addition, a SBI implementation can also provide firmware performance counters which can monitor firmware events such as number of misaligned load/store instructions, number of RFENCEs, number of IPIs, etc. All firmware counters must have same number of bits and can be up to 64 bits wide.

The SBI PMU extension provides:

1. An interface for supervisor-mode software to discover and configure per-hart hardware/firmware counters
2. A typical perf compatible interface for hardware/firmware performance counters and events
3. Full access to microarchitecture's raw event encodings

To define SBI PMU extension calls, we first define important entities `counter_idx`, `event_idx`, and `event_data`. The `counter_idx` is a logical number assigned to each hardware/firmware counter. The `event_idx` represents a hardware (or firmware) event whereas the `event_data` is 64 bits wide and represents additional configuration (or parameters) for a hardware (or firmware) event.

The event_idx is a 20 bits wide number encoded as follows:

```
event_idx[19:16] = type
event_idx[15:0] = code
```

The below table describes the different types of events supported in this specification.

Table 30. PMU Event Type

Event ID Type	Value	Description
Type #0	0	Hardware general events
Type #1	1	Hardware Cache events
Type #2	2	Hardware raw events (deprecated) Bits allowed for mhpmeventX [0:47]
Type #3	3	Hardware raw events v2 Bits allowed for mhpmeventX [0:55]
Type #15	15	Firmware events

11.1. Event: Hardware general events (Type #0)

The `event_idx.type` (i.e. **event type**) should be `0x0` for all hardware general events and each hardware general event is identified by an unique `event_idx.code` (i.e. **event code**) described in the Table 31 below.

Table 31. PMU Hardware Events

General Event Name	Code	Description
SBI_PMU_HW_NO_EVENT	0	Unused event because `event_idx` cannot be zero
SBI_PMU_HW_CPU_CYCLES	1	Event for each CPU cycle
SBI_PMU_HW_INSTRUCTIONS	2	Event for each completed instruction
SBI_PMU_HW_CACHE_REFERENCES	3	Event for cache hit
SBI_PMU_HW_CACHE_MISSES	4	Event for cache miss
SBI_PMU_HW_BRANCH_INSTRUCTIONS	5	Event for a branch instruction
SBI_PMU_HW_BRANCH_MISSES	6	Event for a branch misprediction
SBI_PMU_HW_BUS_CYCLES	7	Event for each BUS cycle
SBI_PMU_HW_STALLED_CYCLES_FRONTEND	8	Event for a stalled cycle in microarchitecture frontend
SBI_PMU_HW_STALLED_CYCLES_BACKEND	9	Event for a stalled cycle in microarchitecture backend
SBI_PMU_HW_REF_CPU_CYCLES	10	Event for each reference CPU cycle

The `event_data` (i.e. **event data**) is unused for hardware general events and all non-zero values of `event_data` are reserved for future use.

A RISC-V platform might halt the CPU clock when it enters WAIT state using the WFI instruction or enters platform specific SUSPEND state using the SBI HSM hart suspend call.

The **SBI_PMU_HW_CPU_CYCLES** *event counts CPU clock cycles as counted by the* `cycle` *CSR. These may be variable frequency cycles, and are not counted when the CPU clock is halted.*

The **SBI_PMU_HW_REF_CPU_CYCLES** *counts fixed-frequency clock cycles while the CPU clock is not halted. The fixed-frequency of counting might, for example, be the same frequency at which the* `time` *CSR counts.*

The **SBI_PMU_HW_BUS_CYCLES** *counts fixed-frequency clock cycles. The fixed-frequency of counting might be the same frequency at which the* `time` *CSR counts, or may be the frequency of the clock at the boundary between the hart (and it's private caches) and the rest of the system.*

11.2. Event: Hardware cache events (Type #1)

The `event_idx.type` (i.e. **event type**) should be `0x1` for all hardware cache events and each hardware cache event is identified by an unique `event_idx.code` (i.e. **event code**) which is encoded as follows:

```
event_idx.code[15:3] = cache_id
event_idx.code[2:1] = op_id
event_idx.code[0:0] = result_id
```

Below tables show possible values of: `event_idx.code.cache_id` (i.e. **cache event id**), `event_idx.code.op_id` (i.e. **cache operation id**) and `event_idx.code.result_id` (i.e. **cache result id**).

Table 32. PMU Cache Event ID

Cache Event Name	Event ID	Description
SBI_PMU_HW_CACHE_L1D	0	Level1 data cache event
SBI_PMU_HW_CACHE_L1I	1	Level1 instruction cache event
SBI_PMU_HW_CACHE_LL	2	Last level cache event
SBI_PMU_HW_CACHE_DTLB	3	Data TLB event
SBI_PMU_HW_CACHE_ITLB	4	Instruction TLB event
SBI_PMU_HW_CACHE_BPU	5	Branch predictor unit event
SBI_PMU_HW_CACHE_NODE	6	NUMA node cache event

Table 33. PMU Cache Operation ID

Cache Operation Name	Operation ID	Description
SBI_PMU_HW_CACHE_OP_READ	0	Read cache line
SBI_PMU_HW_CACHE_OP_WRITE	1	Write cache line
SBI_PMU_HW_CACHE_OP_PREFETCH	2	Prefetch cache line

Table 34. PMU Cache Operation Result ID

Cache Result Name	Result ID	Description
SBI_PMU_HW_CACHE_RESULT_ACCESS	0	Cache access
SBI_PMU_HW_CACHE_RESULT_MISS	1	Cache miss

The `event_data` (i.e. **event data**) is unused for hardware cache events and all non-zero values of `event_data` are reserved for future use.

11.3. Event: Hardware raw events (Type #2)

The `event_idx.type` (i.e. **event type**) should be `0x2` for all hardware raw events and `event_idx.code` (i.e. **event code**) should be zero.

On RISC-V platforms with 32 bits wide `mhpmeventX` CSRs, the `event_data` configuration (or parameter) should have the 32-bit value to to be programmed in the `mhpmeventX` CSR.

On RISC-V platforms with 64 bits wide `mhpmeventX` CSRs, the `event_data` configuration (or parameter) should have the 48-bit value to be programmed in the lower 48-bits of `mhpmeventX` CSR and the SBI implementation shall determine the value to be programmed in the upper 16 bits of `mhpmeventX` CSR.

> *This event type is deprecated in favor of `raw events v2`.*

11.4. Event: Hardware raw events v2 (Type #3)

The `event_idx.type` (i.e. **event type**) should be `0x3` for all hardware raw events and `event_idx.code` (i.e. **event code**) should be zero.

On RISC-V platforms with 32 bits wide `mhpmeventX` CSRs, the `event_data` configuration (or parameter) should have the 32-bit value to to be programmed in the `mhpmeventX` CSR.

On RISC-V platforms with 64 bits wide `mhpmeventX` CSRs, the `event_data` configuration (or parameter) should have the 56-bit value be programmed in the lower 56-bits of `mhpmeventX` CSR and the SBI implementation shall determine the value to be programmed in the upper 8 bits of `mhpmeventX` CSR based on privilege specification definition.

> *The RISC-V platform hardware implementation may choose to define the expected value to be written to `mhpmeventX` CSR for a hardware event. In case of hardware general/cache events, the RISC-V platform hardware implementation may use the zero-extended `event_idx` as the expected value for simplicity.*

11.5. Event: Firmware events (Type #15)

The `event_idx.type` (i.e. **event type**) should be `0xf` for all firmware events and each firmware event is identified by an unique `event_idx.code` (i.e. **event code**) described in the Table 35 below.

Table 35. PMU Firmware Events

Firmware Event Name	Code	Description
SBI_PMU_FW_MISALIGNED_LOAD	0	Misaligned load trap event
SBI_PMU_FW_MISALIGNED_STORE	1	Misaligned store trap event
SBI_PMU_FW_ACCESS_LOAD	2	Load access trap event
SBI_PMU_FW_ACCESS_STORE	3	Store access trap event
SBI_PMU_FW_ILLEGAL_INSN	4	Illegal instruction trap event
SBI_PMU_FW_SET_TIMER	5	Set timer event
SBI_PMU_FW_IPI_SENT	6	Sent IPI to other hart event
SBI_PMU_FW_IPI_RECEIVED	7	Received IPI from other hart event
SBI_PMU_FW_FENCE_I_SENT	8	Sent FENCE.I request to other hart event
SBI_PMU_FW_FENCE_I_RECEIVED	9	Received FENCE.I request from other hart event
SBI_PMU_FW_SFENCE_VMA_SENT	10	Sent SFENCE.VMA request to other hart event
SBI_PMU_FW_SFENCE_VMA_RECEIVED	11	Received SFENCE.VMA request from other hart event
SBI_PMU_FW_SFENCE_VMA_ASID_SENT	12	Sent SFENCE.VMA with ASID request to other hart event

Firmware Event Name	Code	Description
SBI_PMU_FW_SFENCE_VMA_ASID_RECEIVED	13	Received SFENCE.VMA with ASID request from other hart event
SBI_PMU_FW_HFENCE_GVMA_SENT	14	Sent HFENCE.GVMA request to other hart event
SBI_PMU_FW_HFENCE_GVMA_RECEIVED	15	Received HFENCE.GVMA request from other hart event
SBI_PMU_FW_HFENCE_GVMA_VMID_SENT	16	Sent HFENCE.GVMA with VMID request to other hart event
SBI_PMU_FW_HFENCE_GVMA_VMID_RECEIVED	17	Received HFENCE.GVMA with VMID request from other hart event
SBI_PMU_FW_HFENCE_VVMA_SENT	18	Sent HFENCE.VVMA request to other hart event
SBI_PMU_FW_HFENCE_VVMA_RECEIVED	19	Received HFENCE.VVMA request from other hart event
SBI_PMU_FW_HFENCE_VVMA_ASID_SENT	20	Sent HFENCE.VVMA with ASID request to other hart event
SBI_PMU_FW_HFENCE_VVMA_ASID_RECEIVED	21	Received HFENCE.VVMA with ASID request from other hart event
Reserved	22 - 255	Reserved for future use
Implementation specific events	256 - 65534	SBI implementation specific firmware events
SBI_PMU_FW_PLATFORM	65535	RISC-V platform specific firmware events, where the `event_data` configuration (or parameter) contains the event encoding.

For all firmware events except SBI_PMU_FW_PLATFORM, the `event_data` configuration (or parameter) is unused and all non-zero values of `event_data` are reserved for future use.

11.6. Function: Get number of counters (FID #0)

```
struct sbiret sbi_pmu_num_counters()
```

Returns the number of counters (both hardware and firmware) in `sbiret.value` and always returns `SBI_SUCCESS` in sbiret.error.

11.7. Function: Get details of a counter (FID #1)

```
struct sbiret sbi_pmu_counter_get_info(unsigned long counter_idx)
```

Get details about the specified counter such as underlying CSR number, width of the counter, type of counter hardware/firmware, etc.

The `counter_info` returned by this SBI call is encoded as follows:

```
    counter_info[11:0] = CSR (12bit CSR number)
    counter_info[17:12] = Width (One less than number of bits in CSR)
    counter_info[XLEN-2:18] = Reserved for future use
    counter_info[XLEN-1] = Type (0 = hardware and 1 = firmware)
```

If `counter_info.type == 1` then `counter_info.csr` and `counter_info.width` should be ignored.

Returns the `counter_info` described above in `sbiret.value`.

The possible error codes returned in `sbiret.error` are shown in the Table 36 below.

Table 36. PMU Counter Get Info Errors

Error code	Description
SBI_SUCCESS	`counter_info` read successfully.
SBI_ERR_INVALID_PARAM	`counter_idx` points to an invalid counter.

11.8. Function: Find and configure a matching counter (FID #2)

```
struct sbiret sbi_pmu_counter_config_matching(unsigned long counter_idx_base,
                                              unsigned long counter_idx_mask,
                                              unsigned long config_flags,
                                              unsigned long event_idx,
                                              uint64_t event_data)
```

Find and configure a counter from a set of counters which is not started (or enabled) and can monitor the specified event. The `counter_idx_base` and `counter_idx_mask` parameters represent the set of counters whereas `event_idx` represents the event to be monitored and `event_data` represents any additional event configuration.

The `config_flags` parameter represents additional counter configuration and filter flags. The bit definitions of the `config_flags` parameter are shown in the Table 37 below.

Table 37. PMU Counter Config Match Flags

Flag Name	Bits	Description
SBI_PMU_CFG_FLAG_SKIP_MATCH	0:0	Skip the counter matching
SBI_PMU_CFG_FLAG_CLEAR_VALUE	1:1	Clear (or zero) the counter value in counter configuration
SBI_PMU_CFG_FLAG_AUTO_START	2:2	Start the counter after configuring a matching counter

Flag Name	Bits	Description
SBI_PMU_CFG_FLAG_SET_VUINH	3:3	Event counting inhibited in VU-mode
SBI_PMU_CFG_FLAG_SET_VSINH	4:4	Event counting inhibited in VS-mode
SBI_PMU_CFG_FLAG_SET_UINH	5:5	Event counting inhibited in U-mode
SBI_PMU_CFG_FLAG_SET_SINH	6:6	Event counting inhibited in S-mode
SBI_PMU_CFG_FLAG_SET_MINH	7:7	Event counting inhibited in M-mode
RESERVED	8:(XLEN-1)	Reserved for future use and must be zero.

When **SBI_PMU_CFG_FLAG_SKIP_MATCH** *is set in* `config_flags`*, the SBI implementation will unconditionally select the first counter from the set of counters specified by the* `counter_idx_base` *and* `counter_idx_mask`*.*

The **SBI_PMU_CFG_FLAG_AUTO_START** *flag in* `config_flags` *has no impact on the counter value.*

The `config_flags[3:7]` *bits are event filtering hints so these can be ignored or overridden by the SBI implementation for security concerns or due to lack of event filtering support in the underlying RISC-V platform.*

Returns the `counter_idx` in `sbiret.value` upon success.

In case of failure, the possible error codes returned in `sbiret.error` are shown in the Table 38 below.

Table 38. PMU Counter Config Match Errors

Error code	Description
SBI_SUCCESS	counter found and configured successfully.
SBI_ERR_INVALID_PARAM	set of counters has at least one invalid counter or the given flag parameter has a reserved bit set.
SBI_ERR_NOT_SUPPORTED	none of the counters can monitor the specified event.

11.9. Function: Start a set of counters (FID #3)

```
struct sbiret sbi_pmu_counter_start(unsigned long counter_idx_base,
                                    unsigned long counter_idx_mask,
                                    unsigned long start_flags,
                                    uint64_t initial_value)
```

Start or enable a set of counters on the calling hart with the specified initial value. The `counter_idx_base` and `counter_idx_mask` parameters represent the set of counters whereas the `initial_value` parameter specifies the initial value of the counter.

The bit definitions of the `start_flags` parameter are shown in the Table 39 below.

Table 39. PMU Counter Start Flags

Flag Name	Bits	Description
SBI_PMU_START_SET_INIT_VALUE	0:0	Set the value of counters based on the `initial_value` parameter
SBI_PMU_START_FLAG_INIT_SNAPSHOT	1:1	Initialize the given counters from shared memory if available.
RESERVED	2:(XLEN-1)	Reserved for future use and must be zero.

When `SBI_PMU_START_SET_INIT_VALUE` or `SBI_PMU_START_FLAG_INIT_SNAPSHOT` is not set in `start_flags`, the counter value will not be modified and the event counting will start from the current counter value.

The shared memory address must be set during boot via `sbi_pmu_snapshot_set_shmem` before the `SBI_PMU_START_FLAG_INIT_SNAPSHOT` flag may be used. The SBI implementation must initialize all the given valid counters (to be started) from the value set in the shared snapshot memory.

`SBI_PMU_START_SET_INIT_VALUE` and `SBI_PMU_START_FLAG_INIT_SNAPSHOT` are mutually exclusive as the former is only valid for a single counter.

The possible error codes returned in `sbiret.error` are shown in the Table 40 below.

Table 40. PMU Counter Start Errors

Error code	Description
SBI_SUCCESS	counter started successfully.
SBI_ERR_INVALID_PARAM	set of counters has at least one invalid counter or the given flag parameter has a reserved bit set.
SBI_ERR_ALREADY_STARTED	set of counters includes at least one counter which is already started.
SBI_ERR_NO_SHMEM	the snapshot shared memory is not available and `SBI_PMU_START_FLAG_INIT_SNAPSHOT` is set in the flags.

11.10. Function: Stop a set of counters (FID #4)

```
struct sbiret sbi_pmu_counter_stop(unsigned long counter_idx_base,
                                   unsigned long counter_idx_mask,
                                   unsigned long stop_flags)
```

Stop or disable a set of counters on the calling hart. The `counter_idx_base` and `counter_idx_mask` parameters represent the set of counters. The bit definitions of the `stop_flags` parameter are shown in the Table 41 below.

Table 41. PMU Counter Stop Flags

Flag Name	Bits	Description
SBI_PMU_STOP_FLAG_RESET	0:0	Reset the counter to event mapping.
SBI_PMU_STOP_FLAG_TAKE_SNAPSHOT	1:1	Save a snapshot of the given counter's values in the shared memory if available.
RESERVED	2:(XLEN-1)	Reserved for future use and must be zero.

The shared memory address must be set during boot via `sbi_pmu_snapshot_set_shmem` before the `SBI_PMU_STOP_FLAG_TAKE_SNAPSHOT` flag may be used. The SBI implementation must save the current value of all the stopped counters in the shared memory if `SBI_PMU_STOP_FLAG_TAKE_SNAPSHOT` is set. The values corresponding to all other counters must not be modified. The SBI implementation must additionally update the overflown counter bitmap in the shared memory.

The possible error codes returned in `sbiret.error` are shown in the Table 42 below.

Table 42. PMU Counter Stop Errors

Error code	Description
SBI_SUCCESS	counter stopped successfully.
SBI_ERR_INVALID_PARAM	set of counters has at least one invalid counter or the given flag parameter has a reserved bit set.
SBI_ERR_ALREADY_STOPPED	set of counters includes at least one counter which is already stopped.
SBI_ERR_NO_SHMEM	the snapshot shared memory is not available and `SBI_PMU_STOP_FLAG_TAKE_SNAPSHOT` is set in the flags.

11.11. Function: Read a firmware counter (FID #5)

```
struct sbiret sbi_pmu_counter_fw_read(unsigned long counter_idx)
```

Provide the current firmware counter value in `sbiret.value`. On RV32 systems, the `sbiret.value` will only contain the lower 32 bits of the current firmware counter value.

The possible error codes returned in `sbiret.error` are shown in the Table 43 below.

Table 43. PMU Counter Firmware Read Errors

Error code	Description
SBI_SUCCESS	firmware counter read successfully.
SBI_ERR_INVALID_PARAM	`counter_idx` points to a hardware counter or an invalid counter.

11.12. Function: Read a firmware counter high bits (FID #6)

```
struct sbiret sbi_pmu_counter_fw_read_hi(unsigned long counter_idx)
```

Provide the upper 32 bits of the current firmware counter value in `sbiret.value`. This function always returns zero in `sbiret.value` for RV64 (or higher) systems.

The possible error codes returned in `sbiret.error` are shown in Table 44 below.

Table 44. PMU Counter Firmware Read High Errors

Error code	Description
SBI_SUCCESS	Firmware counter read successfully.
SBI_ERR_INVALID_PARAM	`counter_idx` points to a hardware counter or an invalid counter.

11.13. Function: Set PMU snapshot shared memory (FID #7)

```
struct sbiret sbi_pmu_snapshot_set_shmem(unsigned long shmem_phys_lo,
                                         unsigned long shmem_phys_hi,
                                         unsigned long flags)
```

Set and enable the PMU snapshot shared memory on the calling hart.

If both `shmem_phys_lo` and `shmem_phys_hi` parameters are not all-ones bitwise then `shmem_phys_lo` specifies the lower XLEN bits and `shmem_phys_hi` specifies the upper XLEN bits of the snapshot shared memory physical base address. The `shmem_phys_lo` MUST be 4096 bytes (i.e. page) aligned and the size of the snapshot shared memory must be 4096 bytes. The layout of the snapshot shared memory is described in Table 45.

If both `shmem_phys_lo` and `shmem_phys_hi` parameters are all-ones bitwise then the PMU snapshot shared memory is cleared and disabled.

The `flags` parameter is reserved for future use and must be zero.

This is an optional function and the SBI implementation may choose not to implement it.

Table 45. SBI PMU Snapshot shared memory layout

Name	Offset	Size	Description
counter_overflow_bitmap	0x0000	8	A bitmap of all logical overflown counters relative to the `counter_idx_base`. This is valid only if the `Sscofpmf` ISA extension is available. Otherwise, it must be zero.

Name	Offset	Size	Description
counter_values	0x0008	512	An array of 64-bit logical counters where each index represents the value of each logical counter associated with hardware/firmware relative to the `counter_idx_base`.
Reserved	0x0208	3576	Reserved for future use

Any future revisions to this structure should be made in a backward compatible manner and will be associated with an SBI version.

The logical counter indices in the `counter_overflow_bitmap` and `counter_values` array are relative w.r.t to `counter_idx_base` argument present in the `sbi_pmu_counter_stop` and `sbi_pmu_counter_start` functions. This allows the users to use snapshot feature for more than XLEN counters if required.

This function should be invoked only once per hart at boot time. Once configured, the SBI implementation has read/write access to the shared memory when `sbi_pmu_counter_stop` is invoked with the `SBI_PMU_STOP_FLAG_TAKE_SNAPSHOT` flag set. The SBI implementation has read only access when `sbi_pmu_counter_start` is invoked with the `SBI_PMU_START_FLAG_INIT_SNAPSHOT` flag set. The SBI implementation must not access this memory any other time.

The possible error codes returned in `sbiret.error` are shown in Table 46 below.

Table 46. PMU Setup Snapshot Area Errors

Error code	Description
SBI_SUCCESS	Shared memory was set or cleared successfully.
SBI_ERR_NOT_SUPPORTED	The SBI PMU snapshot functionality is not available in the SBI implementation.
SBI_ERR_INVALID_PARAM	The `flags` parameter is not zero or the `shmem_phys_lo` parameter is not 4096 bytes aligned.
SBI_ERR_INVALID_ADDRESS	The shared memory pointed to by the `shmem_phys_lo` and `shmem_phys_hi` parameters is not writable or does not satisfy other requirements of Section 3.2.
SBI_ERR_FAILED	The request failed for unspecified or unknown other reasons.

11.14. Function: Get PMU Event info (FID #8)

```
struct sbiret sbi_pmu_event_get_info(unsigned long shmem_phys_lo,
                                     unsigned long shmem_phys_hi,
                                     unsigned long num_entries,
                                     unsigned long flags)
```

Get details about any PMU event via shared memory. The supervisor software can get event specific information for multiple events in one shot by writing an entry for each event in the shared memory. Each entry in the shared memory must be encoded as follows:

Table 47. Event info entry format

Word	Name	ACCESS(SBI Implementation)	Encoding
0	event_idx	RO	BIT[0:19] - Describes the `event_idx` BIT[20:31] - Reserved for the future purpose. Must be zero.
1	output	RW	BIT[0] - Boolean value to indicate `event_idx` is supported or not. The SBI implementation MUST update this entire 32-bit word if valid `event_idx` and `event_data` (if applicable) are specified in the entry. BIT[1:31] - Reserved for the future purpose. Must be zero
2-3	event_data	RO	BIT[0:63] - Valid when `event_idx.type` is either `0x2`, `0x3` or `0xf`. It describes the `event_data` for the specific event specified in `event_idx` if applicable.

The caller must initialize the shared memory and add `num_entries` of each event for which it wishes to discover information about. The `shmem_phys_lo` MUST be 16-byte aligned and the size of the share memory must be (16 * `num_entries`) bytes.

The `flags` parameter is reserved for future use and MUST be zero.

The SBI implementation MUST NOT touch the shared memory once this call returns as supervisor software may free the memory at any time.

The possible error codes returned in `sbiret.error` are shown in Table 48 below.

Table 48. PMU Get Event Info Errors

Error code	Description
SBI_SUCCESS	The output field is updated for each event.
SBI_ERR_NOT_SUPPORTED	The SBI PMU event info retrieval function is not available in the SBI implementation.
SBI_ERR_INVALID_PARAM	The `flags` parameter is not zero or the `shmem_phys_lo` parameter is not 16-bytes aligned or any reserved bit in an `event_idx` word is set.
SBI_ERR_INVALID_ADDRESS	The shared memory pointed to by the `shmem_phys_lo` and `shmem_phys_hi` parameters is not writable or does not satisfy other requirements of Section 3.2.
SBI_ERR_FAILED	The write failed for unspecified or unknown other reasons.

11.15. Function Listing

Table 49. PMU Function List

Function Name	SBI Version	FID	EID
sbi_pmu_num_counters	0.3	0	0x504D55
sbi_pmu_counter_get_info	0.3	1	0x504D55
sbi_pmu_counter_config_matching	0.3	2	0x504D55

Function Name	SBI Version	FID	EID
sbi_pmu_counter_start	0.3	3	0x504D55
sbi_pmu_counter_stop	0.3	4	0x504D55
sbi_pmu_counter_fw_read	0.3	5	0x504D55
sbi_pmu_counter_fw_read_hi	2.0	6	0x504D55
sbi_pmu_snapshot_set_shmem	2.0	7	0x504D55
sbi_pmu_event_get_info	3.0	8	0x504D55

Chapter 12. Debug Console Extension (EID #0x4442434E "DBCN")

The debug console extension defines a generic mechanism for debugging and boot-time early prints from supervisor-mode software.

This extension replaces the legacy console putchar (EID #0x01) and console getchar (EID #0x02) extensions. The debug console extension allows supervisor-mode software to write or read multiple bytes in a single SBI call.

If the underlying physical console has extra bits for error checking (or correction) then these extra bits should be handled by the SBI implementation.

It is recommended that bytes sent/received using the debug console extension follow UTF-8 character encoding.

12.1. Function: Console Write (FID #0)

```
struct sbiret sbi_debug_console_write(unsigned long num_bytes,
                                      unsigned long base_addr_lo,
                                      unsigned long base_addr_hi)
```

Write bytes to the debug console from input memory.

The `num_bytes` parameter specifies the number of bytes in the input memory. The physical base address of the input memory is represented by two XLEN bits wide parameters. The `base_addr_lo` parameter specifies the lower XLEN bits and the `base_addr_hi` parameter specifies the upper XLEN bits of the input memory physical base address.

This is a non-blocking SBI call and it may do partial/no writes if the debug console is not able to accept more bytes.

The number of bytes written is returned in `sbiret.uvalue` and the possible error codes returned in `sbiret.error` are shown in Table 50 below.

Table 50. Debug Console Write Errors

Error code	Description
SBI_SUCCESS	Bytes written successfully.
SBI_ERR_INVALID_PARAM	The memory pointed to by the `num_bytes`, `base_addr_lo`, and `base_addr_hi` parameters does not satisfy the requirements described in the Section 3.2
SBI_ERR_DENIED	Writes to the debug console is not allowed.
SBI_ERR_FAILED	Failed to write due to I/O errors.

12.2. Function: Console Read (FID #1)

```
struct sbiret sbi_debug_console_read(unsigned long num_bytes,
                                      unsigned long base_addr_lo,
                                      unsigned long base_addr_hi)
```

Read bytes from the debug console into an output memory.

The `num_bytes` parameter specifies the maximum number of bytes which can be written into the output memory. The physical base address of the output memory is represented by two XLEN bits wide parameters. The `base_addr_lo` parameter specifies the lower XLEN bits and the `base_addr_hi` parameter specifies the upper XLEN bits of the output memory physical base address.

This is a non-blocking SBI call and it will not write anything into the output memory if there are no bytes to be read in the debug console.

The number of bytes read is returned in `sbiret.uvalue` and the possible error codes returned in `sbiret.error` are shown in Table 51 below.

Table 51. Debug Console Read Errors

Error code	Description
SBI_SUCCESS	Bytes read successfully.
SBI_ERR_INVALID_PARAM	The memory pointed to by the `num_bytes`, `base_addr_lo`, and `base_addr_hi` parameters does not satisfy the requirements described in the Section 3.2
SBI_ERR_DENIED	Reads from the debug console is not allowed.
SBI_ERR_FAILED	Failed to read due to I/O errors.

12.3. Function: Console Write Byte (FID #2)

```
struct sbiret sbi_debug_console_write_byte(uint8_t byte)
```

Write a single byte to the debug console.

This is a blocking SBI call and it will only return after writing the specified byte to the debug console. It will also return, with SBI_ERR_FAILED, if there are I/O errors.

The `sbiret.uvalue` is set to zero and the possible error codes returned in `sbiret.error` are shown in Table 52 below.

Table 52. Debug Console Write Byte Errors

Error code	Description
SBI_SUCCESS	Byte written successfully.
SBI_ERR_DENIED	Write to the debug console is not allowed.

Error code	Description
SBI_ERR_FAILED	Failed to write the byte due to I/O errors.

12.4. Function Listing

Table 53. DBCN Function List

Function Name	SBI Version	FID	EID
sbi_debug_console_write	2.0	0	0x4442434E
sbi_debug_console_read	2.0	1	0x4442434E
sbi_debug_console_write_byte	2.0	2	0x4442434E

Chapter 13. System Suspend Extension (EID #0x53555350 "SUSP")

The system suspend extension defines a set of system-level sleep states and a function which allows the supervisor-mode software to request that the system transitions to a sleep state. Sleep states are identified with 32-bit wide identifiers (`sleep_type`). The possible values for the identifiers are shown in Table 54.

The term "system" refers to the world-view of the supervisor software domain invoking the call. System suspend may only suspend the part of the overall system which is visible to the invoking supervisor software domain.

The system suspend extension does not provide any way for supported sleep types to be probed. Platforms are expected to specify their supported system sleep types and per-type wake up devices in their hardware descriptions. The `SUSPEND_TO_RAM` sleep type is the one exception, and its presence is implied by that of the extension.

Table 54. SUSP System Sleep Types

Type	Name	Description
0	SUSPEND_TO_RAM	This is a “suspend to RAM” sleep type, similar to ACPI’s S2 or S3. Entry requires all but the calling hart be in the HSM `STOPPED` state and all hart registers and CSRs saved to RAM.
0x00000001 - 0x7fffffff		Reserved for future use
0x80000000 - 0xffffffff		Platform-specific system sleep types

13.1. Function: System Suspend (FID #0)

```
struct sbiret sbi_system_suspend(uint32_t sleep_type,
                                 unsigned long resume_addr,
                                 unsigned long opaque)
```

A return from a `sbi_system_suspend()` call implies an error and an error code from Table 56 will be in `sbiret.error`. A successful suspend and wake up, results in the hart which initiated the suspend, resuming from the `STOPPED` state. To resume, the hart will jump to supervisor-mode, at the address specified by `resume_addr`, with the specific register values described in Table 55.

Table 55. SUSP System Resume Register State

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

> *A single `unsigned long` parameter is sufficient for `resume_addr`, because the hart will resume execution in supervisor-mode with the MMU off, hence `resume_addr` must be less than XLEN bits wide.*

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

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

Besides ensuring all entry criteria for the selected sleep type are met, such as ensuring other harts are in the `STOPPED` state, the caller must ensure all power units and domains are in a state compatible with the selected sleep type. The preparation of the power units, power domains, and wake-up devices used for resumption from the system sleep state is platform specific and beyond the scope of this specification.

When supervisor software is running inside a virtual machine, the SBI implementation is provided by a hypervisor. System suspend will behave similarly to the native case from the point of view of the supervisor software.

The possible error codes returned in `sbiret.error` are shown in Table 56.

Table 56. SUSP System Suspend Errors

Error code	Description
SBI_ERR_INVALID_PARAM	`sleep_type` is reserved or is platform-specific and unimplemented.
SBI_ERR_NOT_SUPPORTED	`sleep_type` is not reserved and is implemented, but the platform does not support it due to one or more missing dependencies.
SBI_ERR_INVALID_ADDRESS	`resume_addr` is not valid, possibly due to the following reasons: * It is not a valid physical address. * Executable access to the address is prohibited by a physical memory protection mechanism or H-extension G-stage for supervisor mode.
SBI_ERR_DENIED	The suspend request failed due to unsatisfied entry criteria.
SBI_ERR_FAILED	The suspend request failed for unspecified or unknown other reasons.

13.2. Function Listing

Table 57. SUSP Function List

Function Name	SBI Version	FID	EID
sbi_system_suspend	2.0	0	0x53555350

Chapter 14. CPPC Extension (EID #0x43505043 "CPPC")

ACPI defines the Collaborative Processor Performance Control (CPPC) mechanism, which is an abstract and flexible mechanism for the supervisor-mode power-management software to collaborate with an entity in the platform to manage the performance of the processors.

The SBI CPPC extension provides an abstraction to access the CPPC registers through SBI calls. The CPPC registers can be memory locations shared with a separate platform entity such as a BMC. Even though CPPC is defined in the ACPI specification, it may be possible to implement a CPPC driver based on Device Tree.

Table 58 defines 32-bit identifiers for all CPPC registers to be used by the SBI CPPC functions. The first half of the 32-bit register space corresponds to the registers as defined by the ACPI specification. The second half provides the information not defined in the ACPI specification, but is additionally required by the supervisor-mode power-management software.

Table 58. CPPC Registers

Register ID	Register	Bit Width	Attribute	Description
0x00000000	HighestPerformance	32	Read-only	ACPI Spec 6.5: 8.4.6.1.1.1
0x00000001	NominalPerformance	32	Read-only	ACPI Spec 6.5: 8.4.6.1.1.2
0x00000002	LowestNonlinearPerformance	32	Read-only	ACPI Spec 6.5: 8.4.6.1.1.4
0x00000003	LowestPerformance	32	Read-only	ACPI Spec 6.5: 8.4.6.1.1.5
0x00000004	GuaranteedPerformanceRegister	32	Read-only	ACPI Spec 6.5: 8.4.6.1.1.6
0x00000005	DesiredPerformanceRegister	32	Read / Write	ACPI Spec 6.5: 8.4.6.1.2.3
0x00000006	MinimumPerformanceRegister	32	Read / Write	ACPI Spec 6.5: 8.4.6.1.2.2
0x00000007	MaximumPerformanceRegister	32	Read / Write	ACPI Spec 6.5: 8.4.6.1.2.1
0x00000008	PerformanceReductionTolerance Register	32	Read / Write	ACPI Spec 6.5: 8.4.6.1.2.4
0x00000009	TimeWindowRegister	32	Read / Write	ACPI Spec 6.5: 8.4.6.1.2.5
0x0000000A	CounterWraparoundTime	32 / 64	Read-only	ACPI Spec 6.5: 8.4.6.1.3.1
0x0000000B	ReferencePerformanceCounterRegister	32 / 64	Read-only	ACPI Spec 6.5: 8.4.6.1.3.1
0x0000000C	DeliveredPerformanceCounterRegister	32 / 64	Read-only	ACPI Spec 6.5: 8.4.6.1.3.1
0x0000000D	PerformanceLimitedRegister	32	Read / Write	ACPI Spec 6.5: 8.4.6.1.3.2

Register ID	Register	Bit Width	Attribute	Description
0x0000000E	CPPCEnableRegister	32	Read / Write	ACPI Spec 6.5: 8.4.6.1.4
0x0000000F	AutonomousSelectionEnable	32	Read / Write	ACPI Spec 6.5: 8.4.6.1.5
0x00000010	AutonomousActivityWindowRegister	32	Read / Write	ACPI Spec 6.5: 8.4.6.1.6
0x00000011	EnergyPerformancePreferenceRegister	32	Read / Write	ACPI Spec 6.5: 8.4.6.1.7
0x00000012	ReferencePerformance	32	Read-only	ACPI Spec 6.5: 8.4.6.1.1.3
0x00000013	LowestFrequency	32	Read-only	ACPI Spec 6.5: 8.4.6.1.1.7
0x00000014	NominalFrequency	32	Read-only	ACPI Spec 6.5: 8.4.6.1.1.7
0x00000015 - 0x7FFFFFFF				Reserved for future use.
0x80000000	TransitionLatency	32	Read-only	Provides the maximum (worst-case) performance state transition latency in nanoseconds.
0x80000001 - 0xFFFFFFFF				Reserved for future use.

14.1. Function: Probe CPPC register (FID #0)

```
struct sbiret sbi_cppc_probe(uint32_t cppc_reg_id)
```

Probe whether the CPPC register as specified by the `cppc_reg_id` parameter is implemented or not by the platform.

If the register is implemented, `sbiret.value` will contain the register width. If the register is not implemented, `sbiret.value` will be set to `0`.

The possible error codes returned in `sbiret.error` are shown in Table 59.

Table 59. CPPC Probe Errors

Error code	Description
SBI_SUCCESS	Probe completed successfully.
SBI_ERR_INVALID_PARAM	`cppc_reg_id` is reserved.
SBI_ERR_FAILED	The probe request failed for unspecified or unknown other reasons.

14.2. Function: Read CPPC register (FID #1)

```
struct sbiret sbi_cppc_read(uint32_t cppc_reg_id)
```

Reads the register as specified in the `cppc_reg_id` parameter and returns the value in `sbiret.value`. When supervisor mode XLEN is 32, the `sbiret.value` will only contain the lower 32 bits of the CPPC register value.

The possible error codes returned in `sbiret.error` are shown in Table 60.

Table 60. CPPC Read Errors

Error code	Description
SBI_SUCCESS	Read completed successfully.
SBI_ERR_INVALID_PARAM	`cppc_reg_id` is reserved.
SBI_ERR_NOT_SUPPORTED	`cppc_reg_id` is not implemented by the platform.
SBI_ERR_DENIED	`cppc_reg_id` is a write-only register.
SBI_ERR_FAILED	The read request failed for unspecified or unknown other reasons.

14.3. Function: Read CPPC register high bits (FID #2)

```
struct sbiret sbi_cppc_read_hi(uint32_t cppc_reg_id)
```

Reads the upper 32-bit value of the register specified in the `cppc_reg_id` parameter and returns the value in `sbiret.value`. This function always returns zero in `sbiret.value` when supervisor mode XLEN is 64 or higher.

The possible error codes returned in `sbiret.error` are shown in Table 61.

Table 61. CPPC Read Hi Errors

Error code	Description
SBI_SUCCESS	Read completed successfully.
SBI_ERR_INVALID_PARAM	`cppc_reg_id` is reserved.
SBI_ERR_NOT_SUPPORTED	`cppc_reg_id` is not implemented by the platform.
SBI_ERR_DENIED	`cppc_reg_id` is a write-only register.
SBI_ERR_FAILED	The read request failed for unspecified or unknown other reasons.

14.4. Function: Write to CPPC register (FID #3)

```
struct sbiret sbi_cppc_write(uint32_t cppc_reg_id, uint64_t val)
```

Writes the value passed in the `val` parameter to the register as specified in the `cppc_reg_id` parameter.

The possible error codes returned in `sbiret.error` are shown in Table 62.

Table 62. CPPC Write Errors

Error code	Description
SBI_SUCCESS	Write completed successfully.
SBI_ERR_INVALID_PARAM	`cppc_reg_id` is reserved.
SBI_ERR_NOT_SUPPORTED	`cppc_reg_id` is not implemented by the platform.
SBI_ERR_DENIED	`cppc_reg_id` is a read-only register.
SBI_ERR_FAILED	The write request failed for unspecified or unknown other reasons.

14.5. Function Listing

Table 63. CPPC Function List

Function Name	SBI Version	FID	EID
sbi_cppc_probe	2.0	0	0x43505043
sbi_cppc_read	2.0	1	0x43505043
sbi_cppc_read_hi	2.0	2	0x43505043
sbi_cppc_write	2.0	3	0x43505043

Chapter 15. Nested Acceleration Extension (EID #0x4E41434C "NACL")

Nested virtualization is the ability of a hypervisor to run another hypervisor as a guest. RISC-V nested virtualization requires an L0 hypervisor (running in hypervisor-mode) to trap-and-emulate the RISC-V H-extension [1] functionality (such as CSR accesses, HFENCE instructions, HLV/HSV instructions, etc.) for the L1 hypervisor (running in virtualized supervisor-mode).

The SBI nested acceleration extension defines a shared memory based interface between the SBI implementation (or L0 hypervisor) and the supervisor software (or L1 hypervisor) which allows both to collaboratively reduce traps taken by the L0 hypervisor for emulating RISC-V H-extension functionality. The nested acceleration shared memory allows the L1 hypervisor to batch multiple RISC-V H-extension CSR accesses and HFENCE requests which are then emulated by the L0 hypervisor upon an explicit synchronization SBI call.

> *The M-mode firmware should not implement the SBI nested acceleration extension if the underlying platform has the RISC-V H-extension implemented in hardware.*

This SBI extension defines optional features which MUST be discovered by the supervisor software (or L1 hypervisor) before using the corresponding SBI functions. Each nested acceleration feature is assigned a unique ID which is an unsigned 32-bit integer. The Table 64 below provides a list of all nested acceleration features.

Table 64. Nested acceleration features

Feature ID	**Feature Name**	**Description**
`0x00000000`	SBI_NACL_FEAT_SYNC_CSR	Synchronize CSR
`0x00000001`	SBI_NACL_FEAT_SYNC_HFENCE	Synchronize HFENCE
`0x00000002`	SBI_NACL_FEAT_SYNC_SRET	Synchronize SRET
`0x00000003`	SBI_NACL_FEAT_AUTOSWAP_CSR	Autoswap CSR
> `0x00000003`	RESERVED	Reserved for future use

To use the SBI nested acceleration extension, the supervisor software (or L1 hypervisor) MUST set up a nested acceleration shared memory physical address for each virtual hart at boot-time. The physical base address of the nested acceleration shared memory MUST be 4096 bytes (i.e. page) aligned and the size of the nested acceleration shared memory must be `4096 + (1024 * (XLEN / 8))` bytes. The Table 65 below shows the layout of nested acceleration shared memory.

Table 65. Nested acceleration shared memory layout

Name	**Offset**	**Size (bytes)**	**Description**
Scratch space	`0x00000000`	4096	Nested acceleration feature specific data.
CSR space	`0x00001000`	`XLEN * 128`	An array of 1024 XLEN-bit words where each word corresponds to a possible RISC-V H-extension CSR defined in the Table 2.1 of the RISC-V privileged specification [1].

Any nested acceleration feature may define the contents of the scratch space shown in the Table 65 above if required.

The contents of the CSR space shown in the Table 65 above is an array of RISC-V H-extension CSR values where CSR <x> is at index <i> = ((<x> & 0xc00) >> 2) | (<x> & 0xff). The SBI implementation (or L0 hypervisor) MUST update the CSR space whenever the state of any RISC-V H-extension CSR changes unless some nested acceleration feature defines a different behaviour. The Table 66 below shows CSR space index ranges for all possible 1024 RISC-V H-extension CSRs.

Table 66. Nested acceleration H-extension CSR index ranges

H-extension CSR address				**SBI NACL CSR space index**
[11:10]	[9:8]	[7:4]	Hex Range	Hex Range
00	10	xxxx	0x200 - 0x2ff	0x000 - 0x0ff
01	10	0xxx	0x600 - 0x67f	0x100 - 0x17f
01	10	10xx	0x680 - 0x6bf	0x180 - 0x1bf
01	10	11xx	0x6c0 - 0x6ff	0x1c0 - 0x1ff
10	10	0xxx	0xa00 - 0xa7f	0x200 - 0x27f
10	10	10xx	0xa80 - 0xabf	0x280 - 0x2bf
10	10	11xx	0xac0 - 0xaff	0x2c0 - 0x2ff
11	10	0xxx	0xe00 - 0xe7f	0x300 - 0x37f
11	10	10xx	0xe80 - 0xebf	0x380 - 0x3bf
11	10	11xx	0xec0 - 0xeff	0x3c0 - 0x3ff

15.1. Feature: Synchronize CSR (ID #0)

The synchronize CSR feature describes the ability of the SBI implementation (or L0 hypervisor) to allow supervisor software (or L1 hypervisor) to write RISC-V H-extension CSRs using the CSR space.

This nested acceleration feature defines the scratch space offset range 0x0F80 - 0x0FFF (128 bytes) as nested CSR dirty bitmap. The nested CSR dirty bitmap contains 1-bit for each possible RISC-V H-extension CSR.

To write a CSR <x> in nested acceleration shared memory, the supervisor software (or L1 hypervisor) MUST do the following:

1. Compute <i> = ((<x> & 0xc00) >> 2) | (<x> & 0xff)
2. Write a new CSR value at word with index <i> in the CSR space
3. Set the <i> bit in the nested CSR dirty bitmap

To synchronize a CSR <x>, the SBI implementation (or L0 hypervisor) MUST do the following:

1. Compute <i> = ((<x> & 0xc00) >> 2) | (<x> & 0xff)
2. If bit <i> is not set in the nested CSR dirty bitmap then goto step 5
3. Emulate write to CSR <x> with the new CSR value taken from the word with index <i> in the CSR space
4. Clear the <i> bit in the nested CSR dirty bitmap
5. Write back the latest CSR value of CSR <x> to the word with index <i> in the CSR space

When synchronizing multiple CSRs, if the value of a CSR <y> depends on the value of some other CSR <x> then the SBI implementation (or L0 hypervisor) MUST synchronize CSR <x> before CSR <y>. For example, the value of CSR `hip` depends on the value of the CSR `hvip`, which means `hvip` is emulated and written first, followed by `hip`.

15.2. Feature: Synchronize HFENCE (ID #1)

The synchronize HFENCE feature describes the ability of the SBI implementation (or L0 hypervisor) to allow supervisor software (or L1 hypervisor) to issue HFENCE using the scratch space.

This nested acceleration feature defines the scratch space offset range `0x0800 - 0x0F7F` (1920 bytes) as an array of nested HFENCE entries. The total number of nested HFENCE entries are `3840 / XLEN` where each nested HFENCE entry consists of four XLEN-bit words.

A nested HFENCE entry is equivalent to an HFENCE over a range of guest addresses. The Table 67 below shows the nested HFENCE entry format whereas Table 68 below provides a list of nested HFENCE entry types. Upon an explicit synchronize HFENCE request from supervisor software (or L1 hypervisor), the SBI implementation (or L0 hypervisor) will process nested HFENCE entries with the `Config.Pending` bit set. After processing pending nested HFENCE entries, the SBI implementation (or L0 hypervisor) will clear the `Config.Pending` bit of these entries.

Table 67. Nested HFENCE entry format

Word	Name	Encoding

0	Config	Config information about the nested HFENCE entry
		
		BIT[XLEN-1:XLEN-1] - Pending
		BIT[XLEN-2:XLEN-4] - Reserved and must be zero
		BIT[XLEN-5:XLEN-8] - Type
		BIT[XLEN-9:XLEN-9] - Reserved and must be zero
		BIT[XLEN-10:XLEN-16] - Order
		if XLEN == 32 then
		   BIT[15:9] - VMID
		   BIT[8:0] - ASID
		else
		   BIT[29:16] - VMID
		   BIT[15:0] - ASID
		
		The page size for invalidation must be
		`1 << (Config.Order + 12)` bytes.

1	Page_Number	Page address right shifted by `Config.Order + 12`
2	Reserved	Reserved for future use and must be zero
3	Page_Count	Number of pages to invalidate

Table 68. Nested HFENCE entry types

Type	Name	Description
0	GVMA	Invalidate a guest physical address range across all VMIDs. The `VMID` and `ASID` fields of the `Config` word are ignored and MUST be zero.

Type	Name	Description
1	GVMA_ALL	Invalidate all guest physical addresses across all VMIDs. The `Order`, `VMID` and `ASID` fields of the `Config` word are ignored and MUST be zero. The `Page_Number` and `Page_Count` words are ignored and MUST be zero.
2	GVMA_VMID	Invalidate a guest physical address range for a particular VMID. The `ASID` field of the `Config` word is ignored and MUST be zero.
3	GVMA_VMID_ALL	Invalidate all guest physical addresses for a particular VMID. The `Order` and `ASID` fields of the `Config` word are ignored and MUST be zero. The `Page_Number` and `Page_Count` words are ignored and MUST be zero.
4	VVMA	Invalidate a guest virtual address range for a particular VMID. The `ASID` field of the `Config` word is ignored and MUST be zero.
5	VVMA_ALL	Invalidate all guest virtual addresses for a particular VMID. The `Order` and `ASID` fields of the `Config` word are ignored and MUST be zero. The `Page_Number` and `Page_Count` words are ignored and MUST be zero.
6	VVMA_ASID	Invalidate a guest virtual address range for a particular VMID and ASID.
7	VVMA_ASID_ALL	Invalidate all guest virtual addresses for a particular VMID and ASID. The `Order` field of the `Config` word is ignored and MUST be zero. The `Page_Number` and `Page_Count` words are ignored and MUST be zero.
> 7	Reserved	Reserved for future use.

To add a nested HFENCE entry, the supervisor software (or L1 hypervisor) MUST do the following:

1. Find an unused nested HFENCE entry with `Config.Pending == 0`
2. Update the `Page_Number` and `Page_Count` words in the nested HFENCE entry
3. Update the `Config` word in the nested HFENCE entry such that `Config.Pending` bit is set

To synchronize a nested HFENCE entry, the SBI implementation (or L0 hypervisor) MUST do the following:

1. If `Config.Pending == 0` then do nothing and skip below steps
2. Process HFENCE based on details in the nested HFENCE entry
3. Clear the `Config.Pending` bit in the nested HFENCE entry

15.3. Feature: Synchronize SRET (ID #2)

The synchronize SRET feature describes the ability of the SBI implementation (or L0 hypervisor) to do synchronization of CSRs and HFENCEs in the nested acceleration shared memory for the supervisor software (or L1 hypervisor) along with SRET emulation.

This nested acceleration feature defines the scratch space offset range `0x0000 - 0x01FF` (512 bytes) as nested SRET context. The Table 69 below shows contents of the nested SRET context.

Table 69. Nested SRET context

Offset	Name	Encoding
`0 * (XLEN / 8)`	Reserved	Reserved for future use and must be zero
`1 * (XLEN / 8)`	X1	Value to be restored in GPR X1
`2 * (XLEN / 8)`	X2	Value to be restored in GPR X2
`3 * (XLEN / 8)`	X3	Value to be restored in GPR X3
`4 * (XLEN / 8)`	X4	Value to be restored in GPR X4
`5 * (XLEN / 8)`	X5	Value to be restored in GPR X5
`6 * (XLEN / 8)`	X6	Value to be restored in GPR X6
`7 * (XLEN / 8)`	X7	Value to be restored in GPR X7
`8 * (XLEN / 8)`	X8	Value to be restored in GPR X8
`9 * (XLEN / 8)`	X9	Value to be restored in GPR X9
`10 * (XLEN / 8)`	X10	Value to be restored in GPR X10
`11 * (XLEN / 8)`	X11	Value to be restored in GPR X11
`12 * (XLEN / 8)`	X12	Value to be restored in GPR X12
`13 * (XLEN / 8)`	X13	Value to be restored in GPR X13
`14 * (XLEN / 8)`	X14	Value to be restored in GPR X14
`15 * (XLEN / 8)`	X15	Value to be restored in GPR X15
`16 * (XLEN / 8)`	X16	Value to be restored in GPR X16
`17 * (XLEN / 8)`	X17	Value to be restored in GPR X17
`18 * (XLEN / 8)`	X18	Value to be restored in GPR X18
`19 * (XLEN / 8)`	X19	Value to be restored in GPR X19
`20 * (XLEN / 8)`	X20	Value to be restored in GPR X20
`21 * (XLEN / 8)`	X21	Value to be restored in GPR X21
`22 * (XLEN / 8)`	X22	Value to be restored in GPR X22
`23 * (XLEN / 8)`	X23	Value to be restored in GPR X23
`24 * (XLEN / 8)`	X24	Value to be restored in GPR X24
`25 * (XLEN / 8)`	X25	Value to be restored in GPR X25
`26 * (XLEN / 8)`	X26	Value to be restored in GPR X26
`27 * (XLEN / 8)`	X27	Value to be restored in GPR X27
`28 * (XLEN / 8)`	X28	Value to be restored in GPR X28
`29 * (XLEN / 8)`	X29	Value to be restored in GPR X29
`30 * (XLEN / 8)`	X30	Value to be restored in GPR X30
`31 * (XLEN / 8)`	X31	Value to be restored in GPR X31
`32 * (XLEN / 8)` - 0x1FF	Reserved	Reserved for future use

Before sending a synchronize SRET request to the SBI implementation (or L0 hypervisor), the supervisor software (or L1 hypervisor) MUST write the GPR X<i> values to be restored at offset <i> * (XLEN / 8) of the nested SRET context.

Upon a synchronize SRET request from the supervisor software (or L1 hypervisor), the SBI implementation (or L0 hypervisor) MUST do the following:

1. If SBI_NACL_FEAT_SYNC_CSR feature is available then
    a. All RISC-V H-extension CSRs implemented by the SBI implementation (or L0 hypervisor) are synchronized as described in the Section 15.1. This is equivalent to the SBI call `sbi_nacl_sync_csr(-1UL)`.
2. If SBI_NACL_FEAT_SYNC_HFENCE feature is available then
    a. All nested HFENCE entries are synchronized as described in the Section 15.2. This is equivalent to the SBI call `sbi_nacl_sync_hfence(-1UL)`.
3. Restore GPR X<i> registers from the nested SRET context.
4. Emulate the SRET instruction as defined by the RISC-V Privilege specification [1].

15.4. Feature: Autoswap CSR (ID #3)

The autoswap CSR feature describes the ability of the SBI implementation (or L0 hypervisor) to automatically swap certain RISC-V H-extension CSR values from the nested acceleration shared memory in the following situations:

- Before emulating the SRET instruction for a synchronized SRET request from the supervisor software (or L1 hypervisor).
- After supervisor (or L1) virtualization state changes from ON to OFF.

> *The supervisor software (or L1 hypervisor) should use the autoswap CSR feature in conjunction with the synchronize SRET feature.*

This nested acceleration feature defines the scratch space offset range `0x0200 - 0x027F` (128 bytes) as nested autoswap context. The Table 70 below shows contents of the nested autoswap context.

Table 70. Nested autoswap context

Offset	Name	Encoding
`0 * (XLEN / 8)`	Autoswap_Flags	Autoswap flags BIT[XLEN-1:1] - Reserved for future use and must be zero BIT[0:0] - HSTATUS
`1 * (XLEN / 8)`	HSTATUS	Value to be swapped with `HSTATUS` CSR
`2 * (XLEN / 8)` - 0x7F	Reserved	Reserved for future use.

To enable automatic swapping of CSRs from the nested autoswap context, the supervisor software (or L1 hypervisor) MUST do the following:

1. Write the `HSTATUS` swap value in the nested autoswap context.
2. Set `Autoswap_Flags.HSTATUS` bit in the nested autoswap context.

To swap CSRs from the nested autoswap context, the SBI implementation (or L0 hypervisor) MUST do the following:

1. If `Autoswap_Flags.HSTATUS` bit is set in the nested autoswap context then swap the supervisor `HSTATUS` CSR value with the `HSTATUS` value in the nested autoswap context.

15.5. Function: Probe nested acceleration feature (FID #0)

```
struct sbiret sbi_nacl_probe_feature(uint32_t feature_id)
```

Probe a nested acceleration feature. This is a mandatory function of the SBI nested acceleration extension. The `feature_id` parameter specifies the nested acceleration feature to probe. Table 64 provides a list of possible feature IDs.

This function always returns SBI_SUCCESS in `sbiret.error`. It returns `0` in `sbiret.value` if the given `feature_id` is not available, or `1` in `sbiret.value` if it is available.

15.6. Function: Set nested acceleration shared memory (FID #1)

```
struct sbiret sbi_nacl_set_shmem(unsigned long shmem_phys_lo,
                                 unsigned long shmem_phys_hi,
                                 unsigned long flags)
```

Set and enable the shared memory for nested acceleration on the calling hart. This is a mandatory function of the SBI nested acceleration extension.

If both `shmem_phys_lo` and `shmem_phys_hi` parameters are not all-ones bitwise then `shmem_phys_lo` specifies the lower XLEN bits and `shmem_phys_hi` specifies the upper XLEN bits of the shared memory physical base address. `shmem_phys_lo` MUST be 4096 bytes (i.e. page) aligned and the size of the shared memory must be `4096 + (XLEN * 128)` bytes.

If both `shmem_phys_lo` and `shmem_phys_hi` parameters are all-ones bitwise then the nested acceleration features are disabled.

The `flags` parameter is reserved for future use and must be zero.

The possible error codes returned in `sbiret.error` are shown in Table 71.

Table 71. NACL Set Shared Memory Errors

Error code	Description
SBI_SUCCESS	Shared memory was set or cleared successfully.
SBI_ERR_INVALID_PARAM	The `flags` parameter is not zero or or the `shmem_phys_lo` parameter is not 4096 bytes aligned.
SBI_ERR_INVALID_ADDRESS	The shared memory pointed to by the `shmem_phys_lo` and `shmem_phys_hi` parameters does not satisfy the requirements described in Section 3.2.

Error code	Description
SBI_ERR_FAILED	The request failed for unspecified or unknown other reasons.

15.7. Function: Synchronize shared memory CSRs (FID #2)

```
struct sbiret sbi_nacl_sync_csr(unsigned long csr_num)
```

Synchronize CSRs in the nested acceleration shared memory. This is an optional function which is only available if the SBI_NACL_FEAT_SYNC_CSR feature is available. The parameter `csr_num` specifies the set of RISC-V H-extension CSRs to be synchronized.

If `csr_num` is all-ones bitwise then all RISC-V H-extension CSRs implemented by the SBI implementation (or L0 hypervisor) are synchronized as described in the Section 15.1.

If `(csr_num & 0x300) == 0x200` and `csr_num < 0x1000` then only a single RISC-V H-extension CSR specified by the `csr_num` parameter is synchronized as described in the Section 15.1.

The possible error codes returned in `sbiret.error` are shown in Table 72.

Table 72. NACL Synchronize CSR Errors

Error code	Description
SBI_SUCCESS	CSRs synchronized successfully.
SBI_ERR_NOT_SUPPORTED	SBI_NACL_FEAT_SYNC_CSR feature is not available.
SBI_ERR_INVALID_PARAM	`csr_num` is not all-ones bitwise and either: * `(csr_num & 0x300) != 0x200` or * `csr_num >= 0x1000` or * `csr_num` is not implemented by the SBI implementation
SBI_ERR_NO_SHMEM	Nested acceleration shared memory not available.

15.8. Function: Synchronize shared memory HFENCEs (FID #3)

```
struct sbiret sbi_nacl_sync_hfence(unsigned long entry_index)
```

Synchronize HFENCEs in the nested acceleration shared memory. This is an optional function which is only available if the SBI_NACL_FEAT_SYNC_HFENCE feature is available. The parameter `entry_index` specifies the set of nested HFENCE entries to be synchronized.

If `entry_index` is all-ones bitwise then all nested HFENCE entries are synchronized as described in the Section 15.2.

If `entry_index < (3840 / XLEN)` then only a single nested HFENCE entry specified by the `entry_index` parameter is synchronized as described in the Section 15.2.

The possible error codes returned in `sbiret.error` are shown in Table 73.

Table 73. NACL Synchronize HFENCE Errors

Error code	Description
SBI_SUCCESS	HFENCEs synchronized successfully.
SBI_ERR_NOT_SUPPORTED	SBI_NACL_FEAT_SYNC_HFENCE feature is not available.
SBI_ERR_INVALID_PARAM	`entry_index` is not all-ones bitwise and `entry_index >= (3840 / XLEN)`.
SBI_ERR_NO_SHMEM	Nested acceleration shared memory not available.

15.9. Function: Synchronize shared memory and emulate SRET (FID #4)

```
struct sbiret sbi_nacl_sync_sret(void)
```

Synchronize CSRs and HFENCEs in the nested acceleration shared memory and emulate the SRET instruction. This is an optional function which is only available if the SBI_NACL_FEAT_SYNC_SRET feature is available.

This function is used by supervisor software (or L1 hypervisor) to do a synchronize SRET request and the SBI implementation (or L0 hypervisor) MUST handle it as described in the Section 15.3.

This function does not return upon success and the possible error codes returned in `sbiret.error` upon failure are shown in Table 74.

Table 74. NACL Synchronize SRET Errors

Error code	Description
SBI_ERR_NOT_SUPPORTED	SBI_NACL_FEAT_SYNC_SRET feature is not available.
SBI_ERR_NO_SHMEM	Nested acceleration shared memory not available.

15.10. Function Listing

Table 75. NACL Function List

Function Name	SBI Version	FID	EID
sbi_nacl_probe_feature	2.0	0	0x4E41434C
sbi_nacl_set_shmem	2.0	1	0x4E41434C
sbi_nacl_sync_csr	2.0	2	0x4E41434C
sbi_nacl_sync_hfence	2.0	3	0x4E41434C
sbi_nacl_sync_sret	2.0	4	0x4E41434C

Chapter 16. Steal-time Accounting Extension (EID #0x535441 "STA")

SBI implementations may encounter situations where virtual harts are ready to run, but must be withheld from running. These situations may be, for example, when multiple SBI domains share processors or when an SBI implementation is a hypervisor and guest contexts share processors with other guest contexts or host tasks. When virtual harts are at times withheld from running, observers within the contexts of the virtual harts may need a way to account for less progress than would otherwise be expected. The time a virtual hart was ready, but had to wait, is called "stolen time" and the tracking of it is referred to as steal-time accounting. The Steal-time Accounting (STA) extension defines the mechanism in which an SBI implementation provides steal-time and preemption information, for each virtual hart, to supervisor-mode software.

16.1. Function: Set Steal-time Shared Memory Address (FID #0)

```
struct sbiret sbi_steal_time_set_shmem(unsigned long shmem_phys_lo,
                                       unsigned long shmem_phys_hi,
                                       unsigned long flags)
```

Set the shared memory physical base address for steal-time accounting of the calling virtual hart and enable the SBI implementation's steal-time information reporting.

If `shmem_phys_lo` and `shmem_phys_hi` are not all-ones bitwise, then `shmem_phys_lo` specifies the lower XLEN bits and `shmem_phys_hi` specifies the upper XLEN bits of the shared memory physical base address. `shmem_phys_lo` MUST be 64-byte aligned. The size of the shared memory must be at least 64 bytes. The SBI implementation MUST zero the first 64 bytes of the shared memory before returning from the SBI call.

If `shmem_phys_lo` and `shmem_phys_hi` are all-ones bitwise, the SBI implementation will stop reporting steal-time information for the virtual hart.

The `flags` parameter is reserved for future use and MUST be zero.

It is not expected for the shared memory to be written by the supervisor-mode software while it is in use for steal-time accounting. However, the SBI implementation MUST not misbehave if a write from supervisor-mode software occurs, however, in that case, it MAY leave the shared memory filled with inconsistent data.

The SBI implementation MUST stop writing to the shared memory when the supervisor-mode software is not runnable, such as upon system reset or system suspend.

> *Not writing to the shared memory when the supervisor-mode software is not runnable avoids unnecessary work and supports repeatable capture of a system image while the supervisor-mode software is suspended.*

The shared memory layout is defined in Table 76

Table 76. STA Shared Memory Structure

Name	Offset	Size	Description
sequence	0	4	The SBI implementation MUST increment this field to an odd value before writing the `steal` field, and increment it again to an even value after writing `steal` (i.e. an odd sequence number indicates an in-progress update). The SBI implementation SHOULD ensure that the sequence field remains odd for only very short periods of time. The supervisor-mode software MUST check this field before and after reading the `steal` field, and repeat the read if it is different or odd. *This sequence field enables the value of the steal field to be read by supervisor-mode software executing in a 32-bit environment.*
flags	4	4	Always zero. *Future extensions of the SBI call might allow the supervisor-mode software to write to some of the fields of the shared memory. Such extensions will not be enabled as long as a zero value is used for the flags argument to the SBI call.*
steal	8	8	The amount of time in which this virtual hart was not idle and scheduled out, in nanoseconds. The time during which the virtual hart is idle will not be reported as steal-time.
preempted	16	1	An advisory flag indicating whether the virtual hart which registered this structure is running or not. A non-zero value MAY be written by the SBI implementation if the virtual hart has been preempted (i.e. while the `steal` field is increasing), while a zero value MUST be written before the virtual hart starts to run again. *This preempted field can, for example, be used by the supervisor-mode software to check if a lock holder has been preempted, and, in that case, disable optimistic spinning.*
pad	17	47	Pad with zeros to a 64 byte boundary.

`sbiret.value` is set to zero and the possible error codes returned in `sbiret.error` are shown in Table 77 below.

Table 77. STA Set Steal-time Shared Memory Address Errors

Error code	Description
SBI_SUCCESS	The steal-time shared memory physical base address was set or cleared successfully.
SBI_ERR_INVALID_PARAM	The `flags` parameter is not zero or the `shmem_phys_lo` is not 64-byte aligned.

Error code	Description
SBI_ERR_INVALID_ADDRESS	The shared memory pointed to by the `shmem_phys_lo` and `shmem_phys_hi` parameters is not writable or does not satisfy other requirements of Section 3.2.
SBI_ERR_FAILED	The request failed for unspecified or unknown other reasons.

16.2. Function Listing

Table 78. STA Function List

Function Name	SBI Version	FID	EID
sbi_steal_time_set_shmem	2.0	0	0x535441

Chapter 17. Supervisor Software Events Extension (EID #0x535345 "SSE")

The SBI Supervisor Software Events (SSE) extension provides a mechanism to inject software events from an SBI implementation to supervisor software such that it preempts all other traps and interrupts. The supervisor software will receive software events only on harts which are ready to receive them. A software event is delivered only after supervisor software has registered an event handler and enabled the software event.

The software events can be of two types: local or global. A local software event is local to a hart and can be handled only on that hart whereas a global software event is a system event and can be handled by any participating hart.

17.1. Software Event Identification

Each software event is identified by a unique 32-bit unsigned integer called `event_id`. The `event_id` space is divided into multiple 16-bit ranges where each 16-bit range is encoded as follows:

```
event_id[14:14] = Platform (0: Standard event, 1: Platform specific event)
event_id[15:15] = Global (0: Local event, 1: Global event)
```

The Table 79 below show the complete `event_id` space along with standard events based on the above encoding.

Table 79. SSE Event ID Space

Software Event ID	Description
Range 0x00000000 - 0x0000ffff	
0x00000000	Local High Priority RAS event
0x00000001	Local double trap event
0x00000002 - 0x00003fff	Local events reserved for future use
0x00004000 - 0x00007fff	Platform specific local events
0x00008000	Global High Priority RAS event
0x00008001 - 0x0000bfff	Global events reserved for future use
0x0000c000 - 0x0000ffff	Platform specific global events
Range 0x00010000 - 0x0001ffff	
0x00010000	Local PMU overflow event (depends on overflow IRQ)
0x00010001 - 0x00013fff	Local events reserved for future use
0x00014000 - 0x00017fff	Platform specific local events
0x00018000 - 0x0001bfff	Global events reserved for future use
0x0001c000 - 0x0001ffff	Platform specific global events
...	
Range 0x00100000 - 0x0010ffff	

Software Event ID	Description
0x00100000	Local Low Priority RAS event
0x00100001 - 0x00103fff	Local events reserved for future use
0x00104000 - 0x00107fff	Platform specific local events
0x00108000	Global Low Priority RAS event
0x00108001 - 0x0010bfff	Global events reserved for future use
0x0010c000 - 0x0010ffff	Platform specific global events
...	
Range 0xffff0000 - 0xffffffff	
0xffff0000	Software injected local event
0xffff0001 - 0xffff3fff	Local events reserved for future use
0xffff4000 - 0xffff7fff	Platform specific local events
0xffff8000	Software injected global event
0xffff8001 - 0xffffbfff	Global events reserved for future use
0xffffc000 - 0xffffffff	Platform specific global events

17.2. Software Event States

At any point in time, a software event can be in one of the following states:

1. **UNUSED** - Software event is not used by supervisor software
2. **REGISTERED** - Supervisor software has provided an event handler for the software event
3. **ENABLED** - Supervisor software is ready to handle the software event
4. **RUNNING** - Supervisor software is handling the software event

A **global** software event **MUST** be registered and enabled only once by any hart. By default, a global software event will be routed to any hart which is ready to receive software events but supervisor software can provide a preferred hart to handle this software event. The state of a global software event **MUST** be common to all harts.

> *The preferred hart assigned to a global software event by the supervisor software is only a hint about supervisor software's preference. The SBI implementation may choose a different hart for handling the global software event to avoid an inter-processor interrupt.*

A **local** software event **MUST** be registered and enabled by all harts which want to handle this event. A local event is delivered to a hart only when the hart is ready to receive software events and the local event is registered and enabled on that hart. The state of a local software event **MUST** be tracked separately for each hart.

> *If a software event in `RUNNING` state is signalled by the event source again, the software event will be taken only after the running event handler completes, provided that supervisor software doesn't disable the software event upon completion.*

The Figure 4 below shows the state transitions of a software event.

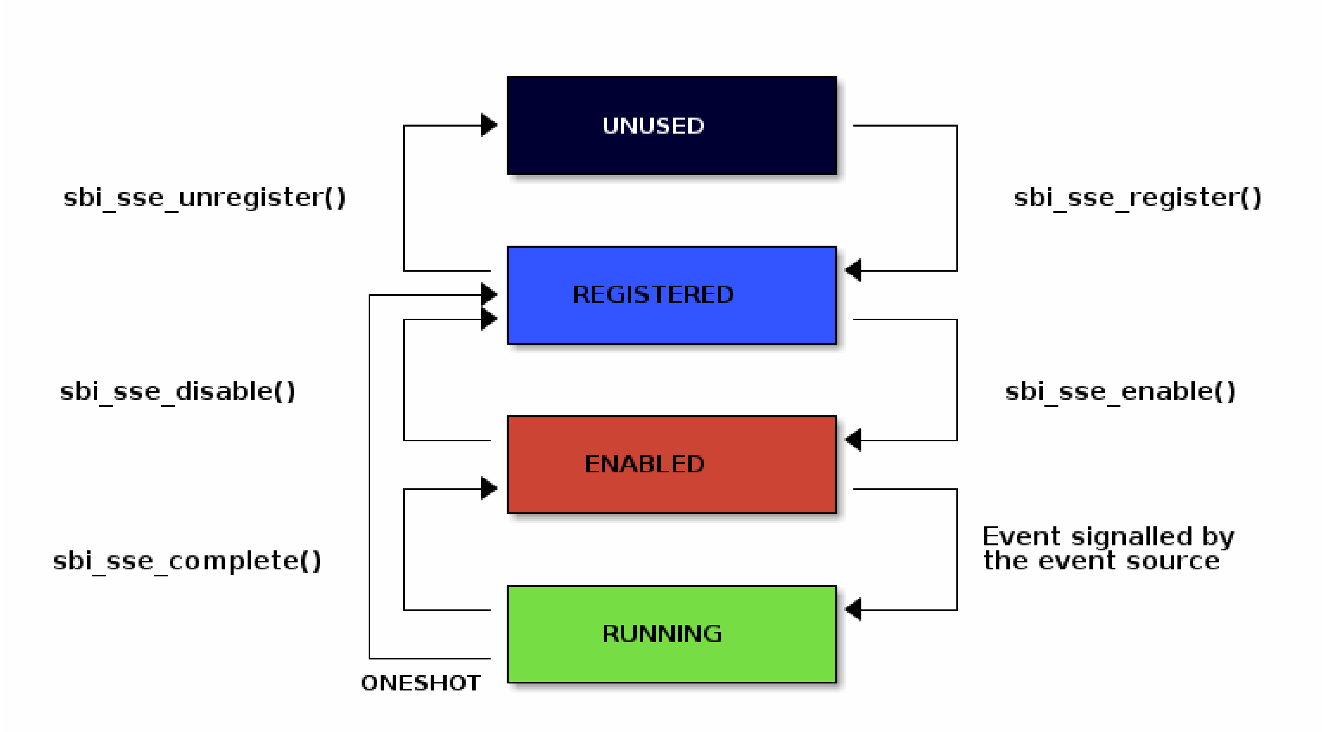

Figure 4. SBI SSE State Machine

17.3. Software Event Priority

Each software event has an associated priority (referred as `event_priority`) which can be used by an SBI implementation to select a software event for injection when multiple software events are pending on the same hart.

The priority of a software event is a 32-bit unsigned integer where lower value means higher priority. By default, all software events have event priority as zero.

If two or more software events have same priority on a given hart then the SBI implementation must use `event_id` for tie-breaking where lower `event_id` has higher priority.

A higher priority software event, unless disabled by supervisor software, **always** preempts a lower priority software event in `RUNNING` state on the same hart. Once a higher priority software event is completed, the previous lower priority software event will be resumed.

17.4. Software Event Attributes

A software event can have various XLEN bits wide attributes associated to it where each event attribute is identified by a unique 32-bit unsigned integer called `attr_id`. A software event attribute can have a Read-Only or Read-Write access permissions. The Table 80 below provides a list event attributes.

Table 80. SSE Event Attributes

Attribute Name	Attribute ID (attr_id)	Access (RO / RW)	Description
STATUS	0x00000000	RO	Status of the software event which is encoded as follows: bit[1:0]: Event state with following possible values: 0 = UNUSED, 1 = REGISTERED, 2 = ENABLED, and 3 = RUNNING bit[2:2]: Event pending status (1 = Pending and 0 = Not Pending). This flag is set by the event source and it is cleared when the software event is moved to RUNNING state. bit[3:3]: Event injection using the sbi_sse_inject call (1 = Allowed and 0 = Not allowed) bit[XLEN-1:4]: Reserved for future use and should be zero The reset value of this attribute is zero.
PRIORITY	0x00000001	RW	Software event priority where only lower 32-bits of the value are used and other bits are always set to zero. This attribute can be updated only when the software event is in UNUSED or REGISTERED state. The reset value of this attribute is zero.
CONFIG	0x00000002	RW	Additional configuration of the software event. This attribute can be updated only when the software event is in UNUSED or REGISTERED state. The encoding of this event attribute is as follows: bit[0:0]: Disable software event upon sbi_sse_complete call (one-shot) bit[XLEN-1:1]: Reserved for future use and should be zero The reset value of this attribute is zero.

Attribute Name	**Attribute ID (attr_id)**	**Access (RO / RW)**	**Description**
PREFERRED_HART	0x00000003	RW (global) RO (local)	Hart id of the preferred hart that should handle the global software event. The value of this attribute must always be valid hart id for both local and global software events. This attribute is read-only for local software events and for global software events it can be updated only when the software event is in `UNUSED` or `REGISTERED` state. The reset value of this attribute is SBI implementation specific.
ENTRY_PC	0x00000004	RO	Entry program counter value for handling the software event in supervisor software. The value of this event attribute MUST be 2-bytes aligned. The reset value of this attribute is zero.
ENTRY_ARG	0x00000005	RO	Entry argument (or parameter) value for handling the software event in supervisor software. This attribute value is passed to the supervisor software via `A7` GPR. The reset value of this attribute is zero.
INTERRUPTED_SEPC	0x00000006	RW	Interrupted `sepc` CSR value which is saved before handling the software event in supervisor software. This attribute can be updated only when the software event is in `RUNNING` state. For global events, only the hart executing the event handler can modify it. The reset value of this attribute is zero.

Attribute Name	Attribute ID (attr_id)	Access (RO / RW)	Description
INTERRUPTED_FLAGS	0x00000007	RW	Interrupted flags which are saved before handling the software event in supervisor software. This attribute can be updated only when the software event is in `RUNNING` state. For global events, only the hart executing the event handler can modify it. The encoding of this event attribute is as follows: `bit[0:0]`: interrupted `sstatus.SPP` CSR bit value `bit[1:1]`: interrupted `sstatus.SPIE` CSR bit value `bit[2:2]`: interrupted `hstatus.SPV` CSR bit value `bit[3:3]`: interrupted `hstatus.SPVP` CSR bit value `bit[XLEN-1:4]`: Reserved for future use and should be zero
INTERRUPTED_A6	0x00000008	RW	Interrupted `A6` GPR value which is saved before handling the software event in supervisor software. This attribute can be updated only when the software event is in `RUNNING` state. For global events, only the hart executing the event handler can modify it. The reset value of this attribute is zero.
INTERRUPTED_A7	0x00000009	RW	Interrupted `A7` GPR value which is saved before handling the software event in supervisor software. This attribute can be updated only when the software event is in `RUNNING` state. For global events, only the hart executing the event handler can modify it. The reset value of this attribute is zero.
RESERVED	> 0x00000009	---	Reserved for future use

17.5. Software Event Injection

To inject a software event on a hart, the SBI implementation must do the following:

1. Save interrupted state of supervisor mode
    a. Set `INTERRUPTED_FLAGS` event attribute as follows:

      i. INTERRUPTED_FLAGS[0:0] = interrupted sstatus.SPP CSR bit value
      ii. INTERRUPTED_FLAGS[1:1] = interrupted sstatus.SPIE CSR bit value
      iii. if H-extension is available to supervisor mode:
         A. Set INTERRUPTED_FLAGS[2:2] = interrupted hstatus.SPV CSR bit value
         B. Set INTERRUPTED_FLAGS[3:3] = interrupted hstatus.SPVP CSR bit value
      iv. else
         A. Set INTERRUPTED_FLAGS[3:2] = zero
      v. Set INTERRUPTED_FLAGS[XLEN-1:4] = zero
   b. Set INTERRUPTED_SEPC event attribute = interrupted sepc CSR
   c. Set INTERRUPTED_A6 event attribute = interrupted A6 GPR value
   d. Set INTERRUPTED_A7 event attribute = interrupted A7 GPR value
2. Redirect execution to supervisor event handler
   a. Set A6 GPR = Current Hart id
   b. Set A7 GPR = ENTRY_ARG event attribute value
   c. Set sepc = Interrupted program counter value
   d. Set sstatus.SPP CSR bit = interrupted privilege mode
   e. Set sstatus.SPIE CSR bit = sstatus.SIE CSR bit value
   f. Set sstatus.SIE CSR bit = zero
   g. if H-extension is available to supervisor mode:
      i. Set hstatus.SPV CSR bit = interrupted virtualization state
      ii. if hstatus.SPV CSR bit == 1:
         A. Set hstatus.SPVP CSR bit = sstatus.SPP CSR bit value
   h. Set virtualization state = OFF
   i. Set privilege mode = S-mode
   j. Set program counter = ENTRY_PC event attribute value

17.6. Software Event Completion

After handling the software event on a hart, the supervisor software must notify the SBI implementation about completion of event handling using sbi_sse_complete call. The SBI implementation must do the following to resume the interrupted state for a completed event:

1. Set program counter = sepc CSR value
2. Set privilege mode = sstatus.SPP CSR bit value
3. if H-extension is available to supervisor mode:
   a. Set virtualization state = hstatus.SPV CSR bit value
   b. Set hstatus.SPV CSR bit = INTERRUPTED_FLAGS[2:2] event attribute value
   c. Set hstatus.SPVP CSR bit = INTERRUPTED_FLAGS[3:3] event attribute value

4. Set sstatus.SIE CSR bit = sstatus.SPIE CSR bit
5. Set sstatus.SPIE CSR bit = INTERRUPTED_FLAGS[1:1] event attribute value
6. Set sstatus.SPP CSR bit = INTERRUPTED_FLAGS[0:0] event attribute value
7. Set A7 GPR = INTERRUPTED_A7 event attribute value
8. Set A6 GPR = INTERRUPTED_A6 event attribute value
9. Set sepc = INTERRUPTED_SEPC event attribute value

If the supervisor software wishes to resume from a different location, it can update the event attributes of the software event before calling sbi_sse_complete.

17.7. Function: Read software event attributes (FID #0)

```
struct sbiret sbi_sse_read_attrs(uint32_t event_id,
                                 uint32_t base_attr_id, uint32_t attr_count,
                                 unsigned long output_phys_lo,
                                 unsigned long output_phys_hi)
```

Read a range of event attribute values from a software event.

The event_id parameter specifies the software event whereas base_attr_id and attr_count parameters specifies the range of event attribute ids.

The event attribute values are written to a output shared memory which is specified by the output_phys_lo and output_phys_hi parameters where:

- The output_phys_lo parameter MUST be XLEN / 8 bytes aligned
- The size of output shared memory is assumed to be (XLEN / 8) * attr_count
- The value of event attribute with id base_attr_id + i should be written at offset (XLEN / 8) * (base_attr_id + i)

In case of an error, the possible error codes are shown in the Table 81 below:

Table 81. SSE Event Attributes Read Errors

Error code	Description
SBI_SUCCESS	Event attribute values read successfully.
SBI_ERR_NOT_SUPPORTED	event is not reserved and valid, but the platform does not support it due to one or more missing dependencies (Hardware or SBI implementation).
SBI_ERR_INVALID_PARAM	event_id is invalid or attr_count is zero.
SBI_ERR_BAD_RANGE	One of the event attribute in the range specified by base_attr_id and attr_count does not exist.
SBI_ERR_INVALID_ADDRESS	The shared memory pointed to by the output_phys_lo and output_phys_hi parameters does not satisfy the requirements described in Section 3.2.

Error code	Description
SBI_ERR_FAILED	The read failed for unspecified or unknown other reasons.

17.8. Function: Write software event attributes (FID #1)

```
struct sbiret sbi_sse_write_attrs(uint32_t event_id,
                                  uint32_t base_attr_id, uint32_t attr_count,
                                  unsigned long input_phys_lo,
                                  unsigned long input_phys_hi)
```

Write a range of event attribute values to a software event.

The `event_id` parameter specifies the software event whereas `base_attr_id` and `attr_count` parameters specifies the range of event attribute ids.

The event attribute values are read from a input shared memory which is specified by the `input_phys_lo` and `input_phys_hi` parameters where:

- The `input_phys_lo` parameter MUST be `XLEN / 8` bytes aligned
- The size of input shared memory is assumed to be `(XLEN / 8) * attr_count`
- The value of event attribute with id `base_attr_id + i` should be read from offset `(XLEN / 8) * (base_attr_id + i)`

For local events, the event attributes are updated only for the calling hart. For global events, the event attributes are updated for all the harts.

The possible error codes returned in `sbiret.error` are shown in Table 82 below.

Table 82. SSE Event Attributes Write Errors

Error code	Description
SBI_SUCCESS	Event attribute values written successfully.
SBI_ERR_NOT_SUPPORTED	`event` is not reserved and valid, but the platform does not support it due to one or more missing dependencies (Hardware or SBI implementation).
SBI_ERR_INVALID_PARAM	`event_id` is invalid or `attr_count` is zero.
SBI_ERR_BAD_RANGE	One of the event attribute in the range specified by `base_attr_id` and `attr_count` does not exist or is read-only.
SBI_ERR_INVALID_ADDRESS	The shared memory pointed to by the `input_phys_lo` and `input_phys_hi` parameters does not satisfy the requirements described in Section 3.2.
SBI_ERR_FAILED	The write failed for unspecified or unknown other reasons.

17.9. Function: Register a software event (FID #2)

```
struct sbiret sbi_sse_register(uint32_t event_id,
                               unsigned long handler_entry_pc,
                               unsigned long handler_entry_arg)
```

Register an event handler for the software event.

The `event_id` parameter specifies the event ID for which an event handler is being registered. The `handler_entry_pc` parameter MUST be 2-bytes aligned and specifies the `ENTRY_PC` event attribute of the software event whereas the `handler_entry_arg` parameter specifies the `ENTRY_ARG` event attribute of the software event.

For local events, the event is registered only for the calling hart. For global events, the event is registered for all the harts.

The event MUST be in `UNUSED` state otherwise this function will fail.

> *It is advisable to use different values for* `handler_entry_arg` *for different events because a higher priority event can preempt a lower priority event.*

Upon success, the event state moves from `UNUSED` to `REGISTERED`. In case of an error, possible error codes are listed in Table 83 below.

Table 83. SSE Event Register Errors

Error code	Description
SBI_SUCCESS	Event handler is registered successfully.
SBI_ERR_NOT_SUPPORTED	`event` is not reserved and valid, but the platform does not support it due to one or more missing dependencies (Hardware or SBI implementation).
SBI_ERR_INVALID_STATE	The event is not in `UNUSED` state.
SBI_ERR_INVALID_PARAM	`event_id` is invalid or `handler_entry_pc` is not 2-bytes aligned.

17.10. Function: Unregister a software event (FID #3)

```
struct sbiret sbi_sse_unregister(uint32_t event_id)
```

Unregister the event handler for given `event_id`.

For local events, the event is unregistered only for the calling hart. For global events, the event is unregistered for all the harts.

The event MUST be in `REGISTERED` state otherwise this function will fail.

Upon success, the event state moves from `REGISTERED` to `UNUSED`. In case of an error, possible error codes are listed in Table 84 below.

Table 84. SSE Event Unregister Errors

Error code	Description
SBI_SUCCESS	Event handler is unregistered successfully.
SBI_ERR_NOT_SUPPORTED	`event` is not reserved and valid, but the platform does not support it due to one or more missing dependencies (Hardware or SBI implementation).
SBI_ERR_INVALID_STATE	Event is not in `REGISTERED` state.
SBI_ERR_INVALID_PARAM	`event_id` is invalid.

17.11. Function: Enable a software event (FID #4)

```
struct sbiret sbi_sse_enable(uint32_t event_id)
```

Enable the software event specified by the `event_id` parameter.

For local events, the event is enabled only for the calling hart. For global events, the event is enabled for all the harts.

The event MUST be in `REGISTERED` state otherwise this function will fail.

Upon success, the event state moves from `REGISTERED` to `ENABLED`. In case of an error, possible error codes are listed in Table 85 below.

Table 85. SSE Event Enable Errors

Error code	Description
SBI_SUCCESS	Event is successfully enabled.
SBI_ERR_NOT_SUPPORTED	`event` is not reserved and valid, but the platform does not support it due to one or more missing dependencies (Hardware or SBI implementation).
SBI_ERR_INVALID_PARAM	`event_id` is not valid.
SBI_ERR_INVALID_STATE	The event is not in `REGISTERED` state.

17.12. Function: Disable a software event (FID #5)

```
struct sbiret sbi_sse_disable(uint32_t event_id)
```

Disable the software event specified by the `event_id` parameter.

For local events, the event is disabled only for the calling hart. For global events, the event is disabled for all the harts.

The event MUST be in `ENABLED` state otherwise this function will fail.

Upon success, the event state moves from `ENABLED` to `REGISTERED`. In case of an error, possible error codes are listed in Table 86 below.

Table 86. SSE Event Disable Errors

Error code	Description
SBI_SUCCESS	Event is successfully disabled.
SBI_ERR_NOT_SUPPORTED	`event` is not reserved and valid, but the platform does not support it due to one or more missing dependencies (Hardware or SBI implementation).
SBI_ERR_INVALID_PARAM	`event_id` is not valid.
SBI_ERR_INVALID_STATE	Event is not in `ENABLED` state.

17.13. Function: Complete software event handling (FID #6)

```
struct sbiret sbi_sse_complete(void)
```

Complete the supervisor event handling for the highest priority event in `RUNNING` state on the calling hart.

If there were no events in `RUNNING` state on the calling hart then this function does nothing and returns `SBI_SUCCESS` otherwise it moves the highest priority event in `RUNNING` state to `ENABLED` state and resumes interrupted supervisor state as decribed in Section 17.6.

17.14. Function: Inject a software event (FID #7)

```
struct sbiret sbi_sse_inject(uint32_t event_id, unsigned long hart_id)
```

The supervisor software can inject a software event with the help of this function. The `event_id` paramater refers to the event to be injected.

For local events, the `hart_id` parameter refers to the hart on which the event is to be injected. For global events, the `hart_id` parameter is ignored.

An event can only be injected if it is allowed by the event attribute as described in Table 80.

In case of an error, possible error codes are listed in Table 87 below.

Table 87. SSE Event Inject Errors

Error code	Description
SBI_SUCCESS	Event is successfully injected.
SBI_ERR_NOT_SUPPORTED	`event` is not reserved and valid, but the platform does not support it due to one or more missing dependencies (Hardware or SBI implementation).
SBI_ERR_INVALID_PARAM	`event_id` or `hart_id` is invalid.

Error code	Description
SBI_ERR_FAILED	The injection failed for unspecified or unknown other reasons.

17.15. Function: Unmask software events on a hart (FID #8)

```
struct sbiret sbi_sse_hart_unmask(void)
```

Start receiving (or unmask) software events on the calling hart. In other words, the calling hart is ready to receive software events from the SBI implementation.

The software events are masked initially on all harts so the supervisor software must explicitly unmask software events on relevant harts at boot-time.

In case of an error, possible error codes are listed in Table 88 below.

Table 88. SSE Hart Unmask Errors

Error code	Description
SBI_SUCCESS	Software events unmasked successfully on the calling hart.
SBI_ERR_ALREADY_STARTED	Software events were already unmasked on the calling hart.
SBI_ERR_FAILED	The request failed for unspecified or unknown other reasons.

17.16. Function: Mask software events on a hart (FID #9)

```
struct sbiret sbi_sse_hart_mask(void)
```

Stop receiving (or mask) software events on the calling hart. In other words, the calling hart is not ready to receive software events from the SBI implementation.

In case of an error, possible error codes are listed in Table 89 below.

Table 89. SSE Hart Mask Errors

Error code	Description
SBI_SUCCESS	Software events masked successfully on the calling hart.
SBI_ERR_ALREADY_STOPPED	Software events were already masked on the calling hart.
SBI_ERR_FAILED	The request failed for unspecified or unknown other reasons.

17.17. Function Listing

Table 90. SSE Function List

Function Name	SBI Version	FID	EID
sbi_sse_read_attrs	3.0	0	0x535345
sbi_sse_write_attrs	3.0	1	0x535345
sbi_sse_register	3.0	2	0x535345
sbi_sse_unregister	3.0	3	0x535345
sbi_sse_enable	3.0	4	0x535345
sbi_sse_disable	3.0	5	0x535345
sbi_sse_complete	3.0	6	0x535345
sbi_sse_inject	3.0	7	0x535345
sbi_sse_hart_unmask	3.0	8	0x535345
sbi_sse_hart_mask	3.0	9	0x535345

Chapter 18. SBI Firmware Features Extension (EID #0x46574654 "FWFT")

The Firmware Features extension enables supervisor-mode software to manage and control specific hardware capabilities or SBI implementation features. Table 91 defines 32-bit identifiers for the features which supervisor-mode software may request to set or get.

Table 91. FWFT Feature Types

Value	Name	Description
0x00000000	MISALIGNED_EXC_DELEG	Control misaligned access exception delegation to supervisor-mode
0x00000001	LANDING_PAD	Control landing pad support for supervisor-mode.
0x00000002	SHADOW_STACK	Control shadow stack support for supervisor-mode.
0x00000003	DOUBLE_TRAP	Control double trap support.
0x00000004	PTE_AD_HW_UPDATING	Control hardware updating of PTE A/D bits.
0x00000005	POINTER_MASKING_PMLEN	Control the pointer masking length for supervisor-mode.
0x00000006 - 0x3fffffff		Local feature types reserved for future use.
0x40000000 - 0x7fffffff		Platform specific local feature types.
0x80000000 - 0xbfffffff		Global feature types reserved for future use.
0xc0000000 - 0xffffffff		Platform specific global feature types.

These features have some attributes that define their behavior and are described in Table 92. The attribute values are defined for each feature in Table 93.

Table 92. FWFT Feature Attributes

Attribute	Description
Scope	Defines if a feature is local (per-hart) or global. Global features only need to be enabled/disabled by a single hart, whereas local features need to be enabled/disabled by each hart. The status and flags of local features can be different from one hart to another.
Reset value	Reset value of the feature. Might be implementation defined.
Values	Per feature values that can be set.

During non-retentive suspend, feature values are retained and restored by the SBI when resuming operations. Upon hart reset, local feature values are not retained and reset to their default reset values according to the feature description. Upon system reset, global and local feature values are reset.

Table 93. FWFT Feature Attribute Values

Feature Name	Reset	Scope	Values	
MISALIGNED_EXC_DELEG	Implementation-defined	Local	0	Disable misaligned exception delegation.
			1	Enable misaligned exception delegation.
LANDING_PAD	0	Local	0	Disable landing pad for supervisor-mode.
			1	Enable landing pad for supervisor-mode.
SHADOW_STACK	0	Local	0	Disable shadow-stack for supervisor-mode.
			1	Enable shadow-stack for supervisor-mode.
DOUBLE_TRAP	0	Local	0	Disable double trap
			1	Enable double trap
PTE_AD_HW_UPDATING	0	Local	0	Disable hardware updating of PTE A/D bits for supervisor-mode.
			1	Enable hardware updating of PTE A/D bits for supervisor-mode.
POINTER_MASKING_PMLEN	0	Local	0	Disable pointer masking for supervisor-mode.
			N	Enable pointer masking for supervisor-mode with PMLEN >= N. A call to `sbi_fwft_get()` returns the actual value of PMLEN.

18.1. Function: Firmware Features Set (FID #0)

```
struct sbiret sbi_fwft_set(uint32_t feature,
                           unsigned long value,
                           unsigned long flags)
```

A successful return from `sbi_fwft_set()` results in the requested firmware feature to be set according to the `value` and `flags` parameters for which per feature supported values are described in Table 93 and flags in Table 94.

The set operation will succeed if requested `value` matches the existing value.

Table 94. FWFT Firmware Features Set Flags

Name	Encoding	Description
LOCK	BIT[0]	If provided, once set, the feature value can no longer be modified until: - hart reset for feature with local scope - system reset for feature with global scope
	BIT[XLEN-1:1]	Reserved for future use and must be zero.

In case of failure, `feature` value is not modified and the possible error codes returned in `sbiret.error` are shown in Table 95 below.

Table 95. FWFT Firmware Features Set Errors

Error code	Description
SBI_SUCCESS	`feature` was set successfully.
SBI_ERR_NOT_SUPPORTED	`feature` is not reserved and valid, but the platform does not support it due to one or more missing dependencies (Hardware or SBI implementation).
SBI_ERR_INVALID_PARAM	Provided `value` or `flags` parameter is invalid.
SBI_ERR_DENIED	`feature` set operation failed because either: - it was denied by the SBI implementation - `feature` is reserved or is platform-specific and unimplemented
SBI_ERR_DENIED_LOCKED	`feature` set operation failed because the `feature` is locked
SBI_ERR_FAILED	The set operation failed for unspecified or unknown other reasons.

The rationale for an SBI implementation to return `SBI_ERR_DENIED` is for instance to allow some hypervisors to simply passthrough the misaligned delegation state to the Guest/VM and deny any changes to that delegation state from the Guest/VM. If authorized, an SBI call would be required at each Guest/VM switch if delegation choices are different between Host and Guest/VM.

18.2. Function: Firmware Features Get (FID #1)

```
struct sbiret sbi_fwft_get(uint32_t feature)
```

A successful return from `sbi_fwft_get()` results in the firmware feature configuration value to be returned in `sbiret.value`. Possible `sbiret.value` values are described in Table 93 for each feature ID.

In case of failure, the content of `sbiret.value` is zero and the possible error codes returned in `sbiret.error` are shown in Table 96.

Table 96. FWFT Firmware Features Get Errors

Error code	Description
SBI_SUCCESS	Feature status was retrieved successfully.
SBI_ERR_NOT_SUPPORTED	`feature` is not reserved and valid, but the platform does not support it due to one or more missing dependencies (Hardware or SBI implementation).
SBI_ERR_DENIED	`feature` is reserved or is platform-specific and unimplemented.

Error code	Description
SBI_ERR_FAILED	The get operation failed for unspecified or unknown other reasons.

18.3. Function Listing

Table 97. FWFT Function List

Function Name	SBI Version	FID	EID
sbi_fwft_set	3.0	0	0x46574654
sbi_fwft_get	3.0	1	0x46574654

Chapter 19. Debug Triggers Extension (EID #0x44425452 "DBTR")

The RISC-V Sdtrig extension [2] allows machine-mode software to directly configure debug triggers which in-turn allows native (or hosted) debugging in machine-mode without any external debugger. Unfortunately, the debug triggers are only accessible to machine-mode.

The SBI debug trigger extension defines a SBI based abstraction to provide native debugging for supervisor-mode software such that it is:

1. Suitable for the rich operating systems and hypervisors running in supervisor-mode.
2. Allows Guest (VS-mode) and Hypervisor (HS-mode) to share debug triggers on a hart.

Each hart on a RISC-V platform has a fixed number of debug triggers which is referred to as `trig_max` in this SBI extension. Each debug trigger is assigned a logical index called `trig_idx` by the SBI implementation where `-1 < trig_idx < trig_max`.

> *The `trig_max` may vary across harts on a platform with asymmetric harts.*

The configuration of each debug trigger is expressed by three parameters `trig_tdata1`, `trig_tdata2`, and `trig_tdata3` which are encoded in the same way as the `tdata1`, `tdata2`, and `tdata3` CSRs defined by the RISC-V Sdtrig extension [2] but with the following additional constraints:

1. The `trig_tdata1.dmode` bit must always be zero.
2. The `trig_tdata1.m` bit must always be zero.

The SBI implementation MUST also maintain an additional software state for each debug trigger called `trig_state` which is encoded as shown in Table 98 below.

Table 98. Debug Trigger State Fields

Field Name	Bits	Description
hw_trig_idx	`trig_state[XLEN-1:8]`	hardware (or physical) index of the debug trigger. This field must be ignored when `trig_state.have_hw_trig == 0`.
reserved	`trig_state[7:6]`	Reserved for future use and must be zero.
have_hw_trig	`trig_state[5:5]`	When set, the hardware (or physical) debug trigger details are available.
vs	`trig_state[4:4]`	Saved copy of the `trig_tdata1.vs` bit.
vu	`trig_state[3:3]`	Saved copy of the `trig_tdata1.vu` bit.
s	`trig_state[2:2]`	Saved copy of the `trig_tdata1.s` bit.
u	`trig_state[1:1]`	Saved copy of the `trig_tdata1.u` bit.
mapped	`trig_state[0:0]`	When set, the trigger has been mapped to some HW debug trigger.

19.1. Function: Get number of triggers (FID #0)

```
struct sbiret sbi_debug_num_triggers(unsigned long trig_tdata1)
```

Get the number of debug triggers on the calling hart which can support the trigger configuration specified by `trig_tdata1` parameter.

This function always returns SBI_SUCCESS in `sbiret.error`. It will return `trig_max` in `sbiret.value` when `trig_tdata1 == 0` otherwise it will return the number of matching debug triggers in `sbiret.value`.

19.2. Function: Set trigger shared memory (FID #1)

```
struct sbiret sbi_debug_set_shmem(unsigned long shmem_phys_lo,
                                  unsigned long shmem_phys_hi,
                                  unsigned long flags)
```

Set and enable the shared memory for debug trigger configuration on the calling hart.

If both `shmem_phys_lo` and `shmem_phys_hi` parameters are not all-ones bitwise then `shmem_phys_lo` specifies the lower XLEN bits and `shmem_phys_hi` specifies the upper XLEN bits of the shared memory physical base address. The `shmem_phys_lo` MUST be `(XLEN / 8)` bytes aligned and the size of shared memory is assumed to be `trig_max * (XLEN / 2)` bytes.

If both `shmem_phys_lo` and `shmem_phys_hi` parameters are all-ones bitwise then shared memory for debug trigger configuration is disabled.

The `flags` parameter is reserved for future use and MUST be zero.

The possible error codes returned in `sbiret.error` are shown in Table 99.

Table 99. Debug Triggers Set Shared Memory Errors

Error code	Description
SBI_SUCCESS	Shared memory was set or cleared successfully.
SBI_ERR_INVALID_PARAM	The `flags` parameter is not zero or the `shmem_phys_lo` parameter is not `(XLEN / 8)` bytes aligned.
SBI_ERR_INVALID_ADDRESS	The shared memory pointed to by the `shmem_phys_lo` and `shmem_phys_hi` parameters does not satisfy the requirements described in Section 3.2.
SBI_ERR_FAILED	The request failed for unspecified or unknown other reasons.

19.3. Function: Read triggers (FID #2)

```
struct sbiret sbi_debug_read_triggers(unsigned long trig_idx_base,
```

```
                                                unsigned long trig_count)
```

Read the debug trigger state and configuration into shared memory for a range of debug triggers specified by the `trig_idx_base` and `trig_count` parameters on the calling hart.

For each debug trigger with index `trig_idx_base + i` where `-1 < i < trig_count`, the debug trigger state and configuration consisting of four XLEN-bit words are written in little-endian format at `offset = i * (XLEN / 2)` of the shared memory as follows:

```
    word[0] = `trig_state` written by the SBI implementation
    word[1] = `trig_tdata1` written by the SBI implementation
    word[2] = `trig_tdata2` written by the SBI implementation
    word[3] = `trig_tdata3` written by the SBI implementation
```

The possible error codes returned in `sbiret.error` are shown in Table 100.

Table 100. Debug Triggers Read Errors

Error code	Description
SBI_SUCCESS	State and configuration of triggers read successfully.
SBI_ERR_NO_SHMEM	Shared memory for debug triggers is disabled.
SBI_ERR_BAD_RANGE	Either `trig_idx_base >= trig_max` or `trig_idx_base + trig_count >= trig_max`

19.4. Function: Install triggers (FID #3)

```
struct sbiret sbi_debug_install_triggers(unsigned long trig_count)
```

Install debug triggers based on an array of trigger configurations in the shared memory of the calling hart. The `trig_idx` assigned to each installed trigger configuration is written back in the shared memory.

The `trig_count` parameter represents the number of trigger configuration entries in the shared memory at offset `0x0`.

The i'th trigger configuration at `offset = i * (XLEN / 2)` in the shared memory consists of four consecutive XLEN-bit words in little-endian format which are organized as follows:

```
    word[0] = `trig_idx` written back by the SBI implementation
    word[1] = `trig_tdata1` read by the SBI implementation
    word[2] = `trig_tdata2` read by the SBI implementation
    word[3] = `trig_tdata3` read by the SBI implementation
```

The SBI implementation MUST consider trigger configurations in the increasing order of the array index and starting with array index `0`. To install a debug trigger for the trigger configuration at array index `i` in the shared memory, the SBI implementation MUST do the following:

1. Map an unused HW debug trigger which matches the trigger configuration to an an unused `trig_idx`.
2. Save a copy of the `trig_tdata1.vs`, `trig_tdata1.vu`, `trig_tdata1.s`, and `trig_tdata.u` bits in `trig_state`.
3. Update the `tdata1`, `tdata2`, and `tdata3` CSRs of the HW debug trigger.
4. Write `trig_idx` at `offset = i * (XLEN / 2)` in the shared memory.

Additionally for each trigger configuration chain in the shared memory, the SBI implementation MUST assign contiguous `trig_idx` values and contiguous HW debug triggers when installing the trigger configuration chain.

The last trigger configuration in the shared memory MUST not have `trig_tdata1.chain == 1` for `trig_tdata1.type = 2 or 6` to prevent incomplete trigger configuration chain in the shared memory.

The `sbiret.value` is set to zero upon success or if shared memory is disabled whereas `sbiret.value` is set to the array index `i` of the failing trigger configuration upon other failures.

The possible error codes returned in `sbiret.error` are shown in Table 101.

Table 101. Debug Triggers Install Errors

Error code	Description
SBI_SUCCESS	Triggers installed successfully.
SBI_ERR_NO_SHMEM	Shared memory for debug triggers is disabled.
SBI_ERR_BAD_RANGE	`trig_count >= trig_max`
SBI_ERR_INVALID_PARAM	One of the trigger configuration words `trig_tdata1`, `trig_tdata2`, or `trig_tdata3` has an invalid value.
SBI_ERR_FAILED	Failed to assign `trig_idx` or HW debug trigger for one of the trigger configurations.
SBI_ERR_NOT_SUPPORTED	One of the trigger configuration can't be programmed due to unimplemented optional bits in `tdata1`, `tdata2`, or `tdata3` CSRs.

19.5. Function: Update triggers (FID #4)

```
struct sbiret sbi_debug_update_triggers(unsigned long trig_count)
```

Update already installed debug triggers based on a trigger configuration array in the shared memory of the calling hart.

The `trig_count` parameter represents the number of trigger configuration entries in the shared memory at offset `0x0`.

The i'th trigger configuration at `offset = i * (XLEN / 2)` in the shared memory consists of four consecutive XLEN-bit words in little-endian format as follows:

```
    word[0] = `trig_idx` read by the SBI implementation
    word[1] = `trig_tdata1` read by the SBI implementation
    word[2] = `trig_tdata2` read by the SBI implementation
    word[3] = `trig_tdata3` read by the SBI implementation
```

The SBI implementation MUST consider trigger configurations in the increasing order of array index and starting with array index `0`. To update a debug trigger based on trigger configuration at array index `i` in the shared memory, the SBI implementation MUST do the following:

1. Check and fail if any of the following constraints are not satisfied:
   a. `trig_idx` represents logical index of a installed debug trigger
   b. `trig_tdata1.type` matches with original installed debug trigger
   c. `trig_tdata1.chain` matches with original installed debug trigger
2. Save a copy of the `trig_tdata1.vs`, `trig_tdata1.vu`, `trig_tdata1.s`, and `trig_tdata.u` bits in `trig_state`.
3. Update the `tdata1`, `tdata2`, and `tdata3` CSRs of the HW debug trigger.

The `sbiret.value` is set to zero upon success or if shared memory is disabled whereas `sbiret.value` is set to the array index `i` of the failing trigger configuration upon other failures.

The possible error codes returned in `sbiret.error` are shown in Table 102.

Table 102. Debug Triggers Update Errors

Error code	Description
SBI_SUCCESS	Triggers updated successfully.
SBI_ERR_NO_SHMEM	Shared memory for debug triggers is disabled.
SBI_ERR_BAD_RANGE	`trig_count >= trig_max`
SBI_ERR_INVALID_PARAM	One of the trigger configuration in the shared memory has an invalid of `trig_idx` (i.e. `trig_idx >= trig_max`), `trig_tdata1`, `trig_tdata2`, or `trig_tdata3`.
SBI_ERR_FAILED	One of the trigger configurations has valid `trig_idx` but the corresponding debug trigger is not mapped to any HW debug trigger.
SBI_ERR_NOT_SUPPORTED	One of the trigger configuration can't be programmed due to unimplemented optional bits in `tdata1`, `tdata2`, or `tdata3` CSRs.

19.6. Function: Uninstall a set of triggers (FID #5)

```
struct sbiret sbi_debug_uninstall_triggers(unsigned long trig_idx_base,
                                           unsigned long trig_idx_mask)
```

Uninstall a set of debug triggers specified by the `trig_idx_base` and `trig_idx_mask` parameters on the calling hart. The `trig_idx_base` specifies the starting trigger index, while the `trig_idx_mask` is a bitmask indicating which triggers, relative to the base, are to be uninstalled. Each bit in the mask corresponds to a specific trigger, allowing for batch operations on multiple triggers simultaneously.

For each debug trigger in the specified set of debug triggers, the SBI implementation MUST:

1. Clear the `tdata1`, `tdata2`, and `tdata3` CSRs of the mapped HW debug trigger.
2. Clear the `trig_state` of the debug trigger.
3. Unmap and free the HW debug trigger and corresponding `trig_idx` for re-use in the future trigger installations.

The possible error codes returned in `sbiret.error` are shown in Table 103.

Table 103. Debug Triggers Uninstall Errors

Error code	Description
SBI_SUCCESS	Triggers uninstalled successfully.
SBI_ERR_INVALID_PARAM	One of the debug triggers with index `trig_idx` in the specified set of debug triggers either not mapped to any HW debug trigger OR has `trig_idx` >= `trig_max`.

19.7. Function: Enable a set of triggers (FID #6)

```
struct sbiret sbi_debug_enable_triggers(unsigned long trig_idx_base,
                                        unsigned long trig_idx_mask)
```

Enable a set of debug triggers specified by the `trig_idx_base` and `trig_idx_mask` parameters on the calling hart. The `trig_idx_base` specifies the starting trigger index, while the `trig_idx_mask` is a bitmask indicating which triggers, relative to the base, are to be enabled. Each bit in the mask corresponds to a specific trigger, allowing for batch operations on multiple triggers simultaneously.

To enable a debug trigger in the specified set of debug triggers, the SBI implementation MUST restore the `vs`, `vu`, `s`, and `u` bits of the mapped HW debug trigger from their saved copy in `trig_state`.

The possible error codes returned in `sbiret.error` are shown in Table 104.

Table 104. Debug Triggers Enable Errors

Error code	Description
SBI_SUCCESS	Triggers enabled successfully.
SBI_ERR_INVALID_PARAM	One of the debug triggers with index `trig_idx` in the specified set of debug triggers either not mapped to any HW debug trigger OR has `trig_idx` >= `trig_max`.

19.8. Function: Disable a set of triggers (FID #7)

```
struct sbiret sbi_debug_disable_triggers(unsigned long trig_idx_base,
                                         unsigned long trig_idx_mask)
```

Disable a set of debug triggers specified by the `trig_idx_base` and `trig_idx_mask` parameters on the calling hart. The `trig_idx_base` specifies the starting trigger index, while the `trig_idx_mask` is a bitmask indicating which triggers, relative to the base, are to be disabled. Each bit in the mask corresponds to a specific trigger, allowing for batch operations on multiple triggers simultaneously.

To disable a debug trigger in the specified set of debug triggers, the SBI implementation MUST clear the `vs`, `vu`, `s`, and `u` bits of the mapped HW debug trigger.

The possible error codes returned in `sbiret.error` are shown in Table 105.

Table 105. Debug Triggers Disable Errors

Error code	Description
SBI_SUCCESS	Triggers disabled successfully.
SBI_ERR_INVALID_PARAM	One of the debug triggers with index `trig_idx` in the specified set of debug triggers either not mapped to any HW debug trigger OR has `trig_idx` >= `trig_max`.

19.9. Function Listing

Table 106. Debug Triggers Function List

Function Name	SBI Version	FID	EID
sbi_debug_num_triggers	3.0	0	0x44425452
sbi_debug_set_shmem	3.0	1	0x44425452
sbi_debug_read_triggers	3.0	2	0x44425452
sbi_debug_install_triggers	3.0	3	0x44425452
sbi_debug_update_triggers	3.0	4	0x44425452
sbi_debug_uninstall_triggers	3.0	5	0x44425452
sbi_debug_enable_triggers	3.0	6	0x44425452
sbi_debug_disable_triggers	3.0	7	0x44425452

Chapter 20. Message Proxy Extension (EID #0x4D505859 "MPXY")

The Message Proxy (MPXY) extension allows the supervisor software to send and receive messages through the SBI implementation. This extension defines a generic interface that allows the supervisor software to implement clients for various messaging protocols implemented by the SBI implementation (such as RPMI [3], etc). The SBI MPXY is an abstract interface and agnostic of message protocol implementations in the SBI implementation. The message format used by a client in the supervisor software to send/receive messages through the SBI MPXY extension is defined by the corresponding message protocol specification.

This extension requires a per-hart shared memory between the supervisor software and the SBI implementation for message data transfer. This per-hart shared memory is different from the message protocol specific shared memory that is used between the SBI implementation and the remote entity that implements the message protocol. The remote entity can be implemented as a system-level partition on the same hart or as firmware running on a platform microcontroller or emulated by an SBI implementation. The supervisor software MUST call the `sbi_mpxy_set_shmem` function to set up the shared memory before calling any other function defined in the extension.

20.1. SBI MPXY and Dedicated SBI extension rule

The implementation may only provide either an SBI MPXY or a dedicated SBI extension interface for a specific functionality within the specified message protocol, but never both.

20.2. Message Channels

The MPXY extension defines an abstract message channel which is identified by a unique 32 bits unsigned integer referred to as `channel_id`. The supervisor software can discover the `channel_id` of a message channel using standard hardware discovery mechanisms. The message protocol specification associated with a message channel is discovered through the standard message channel attributes defined in the following sections.

The type of message data, or the group of messages, that may be transmitted over an MPXY message channel is defined by the message protocol specification. The message protocol specification may define multiple message groups, but may allow only a selected set of messages accessible to the supervisor software via the MPXY extension.

Any `channel_id` exported to the supervisor software via the hardware discovery mechanism is implicitly associated with a particular message protocol transport. This binding is internal to the SBI implementation. To the supervisor software, a message channel is an abstract entity with associated attributes that can be accessed through the MPXY extension. The message channel attributes describe the characteristics of a message channel depending on the associated message protocol.

20.3. Message Channel Attributes

Each message channel (`channel_id`) has a set of associated attributes which are identified by a unique 32 bits unsigned integer called `attribute_id` where each attribute value is 32 bits wide.

The message channel attributes are divided into two categories: standard attributes and message protocol specific attributes. The encoding of message channel `attribute_id` is as follows:

```
attribute_id[31] = 0 (Standard)
attribute_id[31] = 1 (Message protocol)
```

Standard attributes are defined by the MPXY extension and all message channels MUST support these attributes. Apart from standard attributes, a message channel may also have message protocol attributes which are defined by the message protocol specification.

Once a client in supervisor software has verified the channel and its associated attributes, it can use the MPXY interface to send messages over the message channel where each message is identified by a 32 bits unsigned integer called `message_id`. The set of `message_id` that can be sent over an MPXY channel are defined by the message protocol specification.

Table 107. MPXY Channel Attributes

Attribute Name	Attribute ID	Access	Description
MSG_PROT_ID	0x00000000	RO	**Message Protocol Identifier** Unique ID for identifying the message protocol specification. The table Table 108 provides a list of supported message protocol specifications and their IDs.

MSG_PROT_VERSION	0x00000001	RO	**Message Protocol Version**
			Version of the message protocol specification.
			`[31:16]: Major version.`
			`[15:0]: Minor version.`
			If the message protocol specification has additional version fields or if the above encoding is not suitable, the message protocol specification may define message protocol specific attribute for discovering the version of the message protocol specification.

MSG_DATA_MAX_LEN	0x00000002	RO	**Maximum Message Data Length** Maximum message data size in bytes supported by the message channel.
MSG_SEND_TIMEOUT	0x00000003	RO	**Message Send Timeout** Timeout for sending a message in microseconds as supported by the message protocol specification. Functions which do not wait for response can use this timeout value.
MSG_COMPLETION_TIMEOUT	0x00000004	RO	**Message Completion Timeout** This is the aggregate of `MSG_SEND_TIMEOUT` and the response receive timeout in microseconds as supported by the message protocol specification. Functions which wait for response can use this timeout value.

<table>
<tr><th>Attribute Name</th><th>Attribute ID</th><th>Access</th><th>Description</th></tr>
<tr><td>CHANNEL_CAPABILITY</td><td>0x00000005</td><td>RO</td><td>Channel Capabilities Bits
<pre>[31:6]: Reserved and `0`

   [5]: Get Notifications (FID #6)
        Support

   [4]: Send Message without
        Response (FID #5) Support

   [3]: Send Message with
        Response (FID #4) Support

   [2]: Events State Support

   [1]: SSE Event

   [0]: MSI</pre>
Any defined bit as 1 means the corresponding capability is supported.
The SBI implementation only needs to support one notification indication method, either MSI or SSE. If both are enabled, the MSI is preferred over the SSE event.
If the Get Notifications (FID #6) is not supported then the Events State Support, SSE Event and MSI bits will be 0. </td></tr>
<tr><td>SSE_EVENT_ID</td><td>0x00000006</td><td>RO</td><td>SSE Event ID
Channel SSE event ID if the SSE is supported as discovered via CHANNEL_CAPABILITY attribute. If the SSE is not supported then this value is unspecified.</td></tr>
</table>

<table>
<tr><th>Attribute Name</th><th>Attribute ID</th><th>Access</th><th>Description</th></tr>
<tr><td>MSI_CONTROL</td><td>0x00000007</td><td>RW</td><td>MSI Control
MSI control for notification indication.
<pre>0 = Disable
1 = Enable</pre>This attribute can be set to 1 if MSI_ADDR_LOW and MSI_ADDR_HIGH attributes point to a valid MSI target.

If the message channel does not support MSI based notification indication discovered via the CHANNEL_CAPABILITY attribute, then the MSI_CONTROL will ignore writes and always reads zero.

The reset value of this attribute is 0.</td></tr>
<tr><td>MSI_ADDR_LOW</td><td>0x00000008</td><td>RW</td><td>MSI Address Low
Low 32 bits of the MSI target physical address.

If the message channel does not support MSI based notification indication then this attribute ignores writes and always reads 0.

The reset value of this attribute is 0.</td></tr>
<tr><td>MSI_ADDR_HIGH</td><td>0x00000009</td><td>RW</td><td>MSI Address High
High 32 bits of the MSI target physical address.

If the message channel does not support MSI based notification indication then this attribute ignores writes and always reads 0.

The reset value of this attribute is 0.</td></tr>
<tr><td>MSI_DATA</td><td>0x0000000A</td><td>RW</td><td>MSI Data
MSI data word written to the MSI target.

If the message channel does not support MSI based notification indication then this attribute ignores writes and always reads 0.

The reset value of this attribute is 0.</td></tr>
</table>

Attribute Name	Attribute ID	Access	Description
EVENTS_STATE_CONTROL	0x0000000B	RW	**Events State Control.** If the message channel supports notification events state data then this attribute can be used to enable state reporting like number of events `RETURNED`, `REMAINING` or `LOST` after a call to Get Notifications (FID #6) function. The reset value of this attribute is `0`, which means disabled. If a client wants to enable events state reporting, it MUST write `1`. If the events state reporting is not supported by the channel or the Get Notifications (FID #6) function is not implemented as indicated by the `CHANNEL_CAPABILITY` attribute, then the writes to this attribute will be ignored. More details on events state data are mentioned in the function Get Notifications (FID #6) description.
RESERVED	0x0000000C - 0x7fffffff		Reserved for future use.
Message Protocol Attributes	0x80000000 - 0xffffffff		Attributes defined by the message protocol specification. Refer to message protocol specification for details.

20.4. Message Protocol IDs

Each message protocol specification supporting MPXY extension will be assigned a 32 bits identifier which is listed in the table below. New message protocol enabling support for MPXY will need to be added in the below table with its assigned ID.

Table 108. MPXY Message Protocol IDs

Message Protocol Name	MSG_PROT_ID value	Description
RPMI	0x00000000	RPMI [3]
RESERVED	0x00000001 - 0x7fffffff	
Vendor Specific	0x80000000 - 0xffffffff	Custom vendor specific message protocol

20.5. Function: Set shared memory (FID #0)

```
struct sbiret sbi_mpxy_set_shmem(unsigned long shmem_size,
                                 unsigned long shmem_phys_lo,
                                 unsigned long shmem_phys_hi,
                                 unsigned long flags)
```

Set the shared memory for sending and receiving messages on the calling hart.

If both `shmem_phys_lo` and `shmem_phys_hi` parameters are not all-ones bit-wise then the `shmem_phys_lo` specifies the lower XLEN bits and `shmem_phys_hi` specifies the upper XLEN bits of the shared memory physical base address. The `shmem_phys_lo` MUST be 4096 bytes aligned and the `shmem_size` MUST be multiples of 4096 bytes.

If both `shmem_phys_lo` and `shmem_phys_hi` parameters are all-ones bit-wise then shared memory is disabled and `shmem_size` parameter is ignored.

The `flags` parameter specifies configuration for shared memory setup and it is encoded as follows:

```
flags[XLEN-1:2]: Reserved for future use and must be zero.
flags[1:0]: Shared memory setup mode (Refer table below).
```

Table 109. MPXY Shared Memory Setup Mode

Mode	flags[1:0]	Description
OVERWRITE	0b00	Ignore the current shared memory state and force setup the new shared memory based on the passed parameters.
OVERWRITE-RETURN	0b01	Same as `OVERWRITE` mode and additionally after the new shared memory state is enabled, the old shared memory `shmem_size`, `shmem_phys_lo` and `shmem_phys_hi` are written in the same order to the new shared memory at offset `0x0`. This flag provide provision to software layers in the supervisor software that want to send messages using the shared memory but do not know the shared memory details that has already been setup. Those software layers can temporarily setup their own shared memory on the calling hart, send messages and then restore back the previous shared memory with the SBI implementation.
RESERVED	0b10 - 0b11	Reserved for future use. Must be initialized to `0`.

The supervisor software may consist of several software layers, including an operating system and runtime firmware, which are mutually exclusive and without any provision for data exchange. Typically, a call is required to invoke the runtime firmware when required by the operating system, and once the runtime firmware has finished the task it returns control to the operating system.
The operating system may setup the shared memory per-hart using the `OVERWRITE` *flag during boot. The runtime firmware may also need to use the MPXY channel to send the message data when its invoked. In such a scenario the runtime firmware can setup its own MPXY channel shared memory on the called hart using the* `OVERWRITE-RETURN` *flag and when finished, can restore the previous shared memory before returning control to the operating system.*

The possible error codes returned in `sbiret.error` are below.

Table 110. MPXY Set Shared Memory Errors

Error code	Description
SBI_SUCCESS	Shared memory was set or cleared successfully.
SBI_ERR_INVALID_PARAM	The `flags` parameter has invalid value or the bits set are within the reserved range. Or the `shmem_phys_lo` parameter is not 4096 bytes aligned or `shmem_size` is not multiple of 4096 bytes.
SBI_ERR_INVALID_ADDRESS	The shared memory pointed to by the `shmem_phys_lo` and `shmem_phys_hi` parameters does not satisfy the requirements described in Section 3.2.
SBI_ERR_FAILED	Failed due to other unspecified errors.

The supervisor software MUST call this function to setup the shared memory first before calling any other function in this extension.

20.6. Function: Get Channel IDs (FID #1)

```
struct sbiret sbi_mpxy_get_channel_ids(uint32_t start_index)
```

Get channel IDs of the message channels accessible to the supervisor software in the shared memory of the calling hart. The channel IDs are returned as an array of 32 bits unsigned integers where the `start_index` parameter specifies the array index of the first channel ID to be returned in the shared memory.

The SBI implementation will return channel IDs in the shared memory of the calling hart as specified by the table below:

Table 111. MPXY Channel IDs Shared Memory Layout

Offset	Field	Description
0x0	REMAINING	Remaining number of channel IDs.
0x4	RETURNED	Number of channel IDs (N) returned in the shared memory.
0x8	CHANNEL_ID [start_index + 0]	Channel ID
0xC	CHANNEL_ID [start_index + 1]	Channel ID
0x8 + ((N-1) * 4)	CHANNEL_ID [start_index + N - 1]	Channel ID

The number of channel IDs returned in the shared memory are specified by the `RETURNED` field whereas the `REMAINING` field specifies the number of remaining channel IDs. If the `REMAINING` is not `0` then supervisor software can call this function again to get remaining channel IDs with `start_index` passed accordingly. The supervisor software may require multiple SBI calls to get the complete list of channel IDs depending on the `RETURNED` and `REMAINING` fields.

The `sbiret.uvalue` is always set to zero whereas the possible error codes returned in `sbiret.error` are below.

Table 112. MPXY Get Channel IDs Errors

Error code	Description
SBI_SUCCESS	The channel ID array has been written successfully.
SBI_ERR_INVALID_PARAM	`start_index` is invalid.
SBI_ERR_NO_SHMEM	The shared memory setup is not done or disabled for the calling hart.
SBI_ERR_DENIED	Getting channel ID array is not allowed on the calling hart.
SBI_ERR_FAILED	Failed due to other unspecified errors.

20.7. Function: Read Channel Attributes (FID #2)

```
struct sbiret sbi_mpxy_read_attributes(uint32_t channel_id,
                                       uint32_t base_attribute_id,
                                       uint32_t attribute_count)
```

Read message channel attributes. The `channel_id` parameter specifies the message channel whereas `base_attribute_id` and `attribute_count` parameters specify the range of attribute ids to be read.

Supervisor software MUST call this function for the contiguous attribute range where the `base_attribute_id` is the starting index of that range and `attribute_count` is the number of attributes in the contiguous range. If there are multiple such attribute ranges then multiple calls of this function may be done from supervisor software. Supervisor software MUST read the message protocol specific attributes via separate call to this function with `base_attribute_id` and `attribute_count` without any overlap with the MPXY standard attributes.

Upon calling this function the message channel attribute values are returned starting from the offset `0x0` in the shared memory of the calling hart where the value of the attribute with `attribute_id = base_attribute_id + i` is available at the shared memory offset `4 * i`.

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

Table 113. MPXY Read Channel Attributes Errors

Error code	Description
SBI_SUCCESS	Message channel attributes has been read successfully.
SBI_ERR_INVALID_PARAM	`attribute_count` is `0`. Or the `attribute_count > (shared memory size)/4`. Or the `base_attribute_id` is not valid.
SBI_ERR_NOT_SUPPORTED	`channel_id` is not supported or invalid.
SBI_ERR_BAD_RANGE	One of the attributes in the range specified by the `base_attribute_id` and `attribute_count` do not exist.
SBI_ERR_NO_SHMEM	The shared memory setup is not done or disabled for calling hart.
SBI_ERR_FAILED	Failed due to other unspecified errors.

20.8. Function: Write Channel Attributes (FID #3)

```
struct sbiret sbi_mpxy_write_attributes(uint32_t channel_id,
                                        uint32_t base_attribute_id,
                                        uint32_t attribute_count)
```

Write message channel attributes. The `channel_id` parameter specifies the message channel whereas `base_attribute_id` and `attribute_count` parameters specify the range of attribute ids.

Supervisor software MUST call this function for the contiguous attribute range where the `base_attribute_id` is the starting index of that range and `attribute_count` is the number of attributes in the contiguous range. If there are multiple such attribute ranges then multiple calls of this function may be done from supervisor software. Apart from contiguous attribute indices, supervisor software MUST also consider the attribute access permissions and attributes with RO (Read Only) access MUST be excluded from the attribute range. Supervisor software MUST write the message protocol specific attributes via separate call to this function with `base_attribute_id` and `attribute_count` without any overlap with the MPXY standard attributes.

Before calling this function, the supervisor software must populate the shared memory of the calling hart starting from offset `0x0` with the message channel attribute values. For each attribute with attribute_id = base_attribute_id + i, the corresponding value MUST be placed at the shared memory offset 4 * i.

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

Table 114. MPXY Write Channel Attributes Errors

Error code	Description
SBI_SUCCESS	Message channel attributes has been written successfully.
SBI_ERR_INVALID_PARAM	`attribute_count` is `0`. Or the `attribute_count > (shared memory size)/4`. Or the `base_attribute_id` is not valid.
SBI_ERR_NOT_SUPPORTED	`channel_id` is not supported or invalid.
SBI_ERR_BAD_RANGE	One of the attributes in the range specified by the `base_attribute_id` and `attribute_count` do not exist or the attribute is read-only (RO). Or `base_attribute_id` and `attribute_count` result into a range which overlaps with standard and message protocol specific attributes.
SBI_ERR_NO_SHMEM	The shared memory setup is not done or disabled for calling hart.
SBI_ERR_DENIED	If any attribute write dependency is not satisfied.
SBI_ERR_FAILED	Failed due to other unspecified errors.

20.9. Function: Send Message with Response (FID #4)

```
struct sbiret
sbi_mpxy_send_message_with_response(uint32_t channel_id,
                                    uint32_t message_id,
```

```
                                        unsigned long message_data_len)
```

Send a message to the MPXY channel specified by the `channel_id` parameter. The `message_id` parameter specifies a message specific to a message protocol to be sent whereas the `message_data_len` parameter represents the length of message data in bytes which is located at the offset `0x0` in the shared memory setup by the calling hart.

After sending the message, this function waits for the message response from the SBI implementation. This function only succeeds upon receipt of the response. Some messages may require sending multiple times for complete data transfer so the supervisor software is responsible for doing multiple requests in such cases. Details of such cases can be found in respective message protocol specifications.

Upon calling this function the SBI implementation MUST write the response message data at the offset `0x0` in the shared memory setup by the calling hart and the number of bytes written will be returned through `sbiret.uvalue`. The layout of data in case of both request and response is according to the respective message protocol specification message format.

Upon success, this function:
1) Writes the message response data at offset `0x0` of the shared memory setup by the calling hart.
2) Returns `SBI_SUCCESS` in `sbiret.error`.
3) Returns message response data length in `sbiret.uvalue`.

This function is optional. If this function is implemented, the corresponding bit in the `CHANNEL_CAPABILITY` attribute is set to 1.

The possible error codes returned in `sbiret.error` are below.

Table 115. MPXY Send Message with Response Errors

Error code	**Description**
SBI_SUCCESS	Message sent and response received successfully.
SBI_ERR_INVALID_PARAM	The `message_data_len > max_message_data_len` for specified `channel_id`. Or the `message_data_len` is greater than the size of shared memory on the calling hart.
SBI_ERR_NOT_SUPPORTED	`channel_id` is not supported or invalid. Or the message represented by the `message_id` is not supported or invalid. Or this function is not supported.
SBI_ERR_NO_SHMEM	The shared memory setup is not done or disabled for calling hart.
SBI_ERR_TIMEOUT	Waiting for response timeout.
SBI_ERR_IO	Failed due to I/O error.
SBI_ERR_FAILED	Failed due to other unspecified errors.

20.10. Function: Send Message without Response (FID #5)

```
struct sbiret
```

```
sbi_mpxy_send_message_without_response(uint32_t channel_id,
                                       uint32_t message_id,
                                       unsigned long message_data_len)
```

Send a message to the MPXY channel specified by the `channel_id` parameter. The `message_id` parameter specifies a message specific to a message protocol to be sent whereas the `message_data_len` parameter represents the length of message data in bytes which is located at the offset `0x0` in the shared memory setup by the calling hart.

This function does not wait for response and returns after successful message transmission.

In cases where complete data transfer requires multiple transmissions, the supervisor software shall issue multiple requests as necessary. Details of such cases can be found in the respective message protocol specification.

This function is optional. If this function is implemented, the corresponding bit in the `CHANNEL_CAPABILITY` attribute is set to 1.

The possible error codes returned in `sbiret.error` are below.

Table 116. MPXY Send Message without Response Errors

Error code	Description
SBI_SUCCESS	Message sent successfully.
SBI_ERR_INVALID_PARAM	The `message_data_len > max_message_data_len` for specified `channel_id`. Or the `message_data_len` is greater than the size of shared memory on the calling hart.
SBI_ERR_NOT_SUPPORTED	`channel_id` is not supported or invalid. Or the message represented by the `message_id` is not supported or invalid. Or this function is not supported.
SBI_ERR_NO_SHMEM	The shared memory setup is not done or disabled for calling hart.
SBI_ERR_TIMEOUT	Message send timeout.
SBI_ERR_IO	Failed due to I/O error.
SBI_ERR_FAILED	Failed due to other unspecified errors.

20.11. Function: Get Notifications (FID #6)

```
struct sbiret sbi_mpxy_get_notification_events(uint32_t channel_id)
```

Get the message protocol specific notification events on the MPXY channel specified by the `channel_id` parameter. The events are message protocol specific and MUST be defined in the respective message protocol specification. The SBI implementation may support notification indication mechanisms like MSI or SSE to indicate the availability of events to the supervisor software.

If the message channel does not support or is not configured for a notification *indication mechanism, such as MSI or SSE, the supervisor software can periodically invoke the poll operation* `sbi_mpxy_get_notification_events`.

Notifications are asynchronous from the perspective of the supervisor software. Any caching or buffering mechanism is specific to the SBI implementation. The supervisor software may periodically poll for notification events using this function, provided that the polling frequency is sufficient to prevent the loss of events due to potential buffer limitations in the SBI implementation.

Depending on the message protocol implementation, a channel may support events state which includes data like number of events `RETURNED`, `REMAINING` and `LOST`. Events state data is optional and if the message protocol implementation supports then the channel will have a corresponding bit set in `CHANNEL_CAPABILITY` attribute.
By default the events state is disabled and clients can explicitly enable it through the `EVENTS_STATE_CONTROL` attribute.

Only after enabling the events state reporting through `EVENTS_STATE_CONTROL` *attribute, the events state data will start getting accumulated by the SBI implementation. A client may enable the* `EVENTS_STATE_CONTROL` *attribute in the initialization phase if it is supported.*

In the shared memory, 16 bytes starting from offset `0x0` are used to return this state data.

Shared memory layout with events state data (each field is of 4 bytes):

```
Offset 0x0: REMAINING
Offset 0x4: RETURNED
Offset 0x8: LOST
Offset 0xC: RESERVED
Offset 0x10: Start of message protocol specific notification events data
```

The `RETURNED` field represents the number of events which are returned in the shared memory when this function is called. The `REMAINING` field represents the number of events still remaining with SBI implementation and the client may need to call this function again until the `REMAINING` field becomes `0`.

The `LOST` field represents the number of events which are lost due to limited buffer size managed by the message protocol implementation. Details of buffering/caching of events is specific to message protocol implementation.

Upon calling this function the received notification events are written by the SBI implementation at the offset `0x10` in the shared memory setup by the calling hart irrespective of events state data reporting. If events state data reporting is disabled or not supported, then the values in events state fields are undefined. The number of the bytes written to the shared memory will be returned through `sbiret.uvalue` which is the number of bytes starting from offset `0x10`. The layout and encoding of notification events are defined by the message protocol specification associated with the message proxy channel (`channel_id`).

This function is optional. If this function is implemented, the corresponding bit in the `CHANNEL_CAPABILITY` attribute is set to 1.

The possible error codes returned in `sbiret.error` are below.

Table 117. MPXY Get Notifications Errors

Error code	Description
SBI_SUCCESS	Notifications received successfully.
SBI_ERR_NOT_SUPPORTED	`channel_id` is not supported or invalid. Or this function is not supported.
SBI_ERR_NO_SHMEM	The shared memory setup is not done or disabled for calling hart.
SBI_ERR_IO	Failed due to I/O error.
SBI_ERR_FAILED	Failed due to other unspecified errors.

20.12. Function Listing

Table 118. MPXY Function List

Function Name	SBI Version	FID	EID
sbi_mpxy_set_shmem	3.0	0	0x4D505859
sbi_mpxy_get_channel_ids	3.0	1	0x4D505859
sbi_mpxy_read_attributes	3.0	2	0x4D505859
sbi_mpxy_write_attributes	3.0	3	0x4D505859
sbi_mpxy_send_message_with_response	3.0	4	0x4D505859
sbi_mpxy_send_message_without_response	3.0	5	0x4D505859
sbi_mpxy_get_notification_events	3.0	6	0x4D505859

Chapter 21. Experimental SBI Extension Space (EIDs #0x08000000 - #0x08FFFFFF)

The SBI specification doesn't define any rules for the EID management for experimental SBI extensions.

Chapter 22. Vendor Specific Extension Space (EIDs #0x09000000 - #0x09FFFFFF)

The lower 24 bits of vendor specific EID must match the lower 24 bits of the `mvendorid` value.

Chapter 23. Firmware Specific Extension Space (EIDs #0x0A000000 - #0x0AFFFFFF)

The lower 24 bits of the firmware EID must match the lower 24 bits of the SBI implementation ID. The firmware specific SBI extensions space is reserved for SBI implementations. It provides firmware specific SBI functions which are defined in the external firmware specification.

References

[1] “The RISC-V Instruction Set Manual, Volume II: Privileged Architecture.” 2021, [Online]. Available: github.com/riscv/riscv-isa-manual/releases/tag/Priv-v1.12.

[2] “The RISC-V Debug Specification.” 2024, [Online]. Available: github.com/riscv/riscv-debug-spec/releases/tag/1.0.0-rc3.

[3] “The RISC-V Platform Management Interface Specification.” 2024, [Online]. Available: github.com/riscv-non-isa/riscv-rpmi/blob/main/riscv-rpmi.adoc.