SMC Calling Convention

Document number:	ARM DEN 0028C
Release Quality:	Release
Issue Number:	1.2
Confidentiality:	Non-Confidential
Date of Issue:	August 2020

Contents

About this document

Release Information

The change history table lists the changes that have been made to this document.

Date	Version	Confidentiality	Change
June 2013	A	Non-Confidential	First release
November 2016	B	Non-Confidential	HVC calling convention. SMC calling convention clarifications and updates.
August 2020	C	Non-Confidential	Merge information contained in DEN 0070 (see [12.]). Add guidelines for SVE register context management and SoC ID Arm architecture call. Allow R4—R7 (SMC32/HVC32) to be used as result registers. Allow X8—X17 to be used as parameter registers in SMC64/HVC64. Allow X4—X17 to be used as result registers in SMC64/HVC64.

SMC Calling Convention

Arm Non-Confidential Document Licence ("Licence")

Any breach of this Licence by a Subsidiary shall entitle Arm to terminate this Licence as if you were the party in breach. Any termination of this Licence shall be effective in respect of all Subsidiaries. Any rights granted to any Subsidiary hereunder shall automatically terminate upon such Subsidiary ceasing to be a Subsidiary.

The Document consists solely of commercial items. Licensee shall be responsible for ensuring that any use, duplication or disclosure of the Document complies fully with any relevant export laws and regulations to assure that the Document or any portion thereof is not exported, directly or indirectly, in violation of such export laws.

This Licence may be translated into other languages for convenience, and Licensee agrees that if there is any conflict between the English version of this Licence and any translation, the terms of the English version of this Licence shall prevail.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

Arm document reference: LES-PRE-21585 version 4.0

References

This document refers to the following documents:

Ref	Document Number	Title
[1.]	ARM DDI 0406	Arm® Architecture Reference Manual Armv7-A and Armv7-R edition
[2.]	ARM DDI 0487	Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile
[3.]	ARM IHI 0042	Procedure Call Standard for the Arm 32-bit Architecture
[4.]	ARM IHI 0055	Procedure Call Standard for the Arm 64-bit Architecture
[5.]	ARM DEN 0022	Power State Coordination Interface
[6.]	-	RFC 4122 - A Universally Unique IDentifier (UUID) URN Namespace http://tools.ietf.org/html/rfc4122
[7.]	-	Arm Security Update https://developer.arm.com/support/security-update
[8.]	ARM_DEN 0054	Software Delegated Exception Interface (SDEI)
[9.]	JEP106AZ	Standard Manufacturer's Identification Code
[10.]	ARM DEN 0060	Arm Management Mode Interface Specification
[11.]	ARM DDI 0584A	Arm Architectural Reference Manual Supplement - The Scalable Vector Extension (SVE), for Armv8-A
[12.]	ARM DEN 0070	Firmware interfaces for mitigating cache speculation vulnerabilities
[13.]	-	Cache Speculation Side-channels https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability
[14.]	ARM DEN 0091	SVE impact on Secure firmware

Terms and abbreviations

This document uses the following terms and abbreviations:

Term	Meaning
AArch32 state	The Arm 32-bit Execution state that uses 32-bit general purpose registers, and a 32-bit Program Counter (PC), Stack Pointer (SP), and Link Register (LR). AArch32 Execution state provides a choice of two instruction sets, A32 and T32, previously called the Arm and Thumb instruction sets.

AArch64 state	The Arm 64-bit Execution state that uses 64-bit general purpose registers, and a 64-bit program counter (PC), stack pointer (SP), and exception link registers (ELR). AArch64 Execution state provides a single instruction set, A64.
ACPI	The Advanced Configuration and Power Interface specification. This defines a standard for device configuration and power management by an OS.
CPU	A hardware implementation of the Arm Architecture.
EL0	The lowest Exception level. The Exception level that is used to execute user applications in Non-secure state.
EL1	The privileged Exception level. The Exception level that is used to execute operating systems in Non-secure state.
EL2	The hypervisor Exception level. The Exception level that is used to execute hypervisor code in Non-secure state.
EL3	The Secure Monitor Exception level. The Exception level that is used to execute Secure Monitor code, which handles the transitions between Non-secure and Secure states. EL3 is always in Secure state.
Execution context	The PE that is state associated with a thread of execution, including register state, exception level and security state. Usually an execution context is managed by another execution context at a higher exception level or an exception level in the Secure state. For example, firmware manages one or more system software execution contexts. However, the managing and managed execution contexts may reside at the same exception level and security state. For example, a runtime environment manages one or more interpreted applications.
Firmware	Software that provides platform specific services. Firmware typically operates at an exception level higher than the operating system or Hypervisor which makes use of the firmware services.
Function Identifier	A 32-bit integer that identifies which function is being invoked by an SMC or HVC call. Passed in R0 or W0 into every SMC or HVC call.
HVC	Hypervisor Call, an Arm assembler instruction that causes an exception that is taken synchronously into EL2.
Hypervisor	The hypervisor runs at the EL2 Exception level, and supports the execution of multiple EL1 operating systems.
Non-secure state	The Arm Execution state that restricts access to only the Non-secure system resources, for example memory, peripherals, and System registers.
OEM	Original Equipment Manufacturer. In this document, the final device manufacturer.
OS	Application operating system, for example Linux or Windows. This also includes a virtualized OS running under a hypervisor.

PE	Processing element. The abstract machine that is defined in the Arm architecture, see [2.]
Rx	Register; A32 native 32-bit register, A64 architectural register
S-EL0	The Secure EL0 Exception level. The Exception level that is used to execute Trusted application code in Secure state.
S-EL1	The Secure EL1 Exception level. The Exception level that is used to execute Trusted OS code in Secure state.
S-EL2	The Secure EL2 Exception level. The Exception level that is used to execute hypervisor code in Secure state.
Secure Monitor	The Secure Monitor is software that executes at the EL3 Exception level. The Secure Monitor receives and handles Secure Monitor exceptions, and provides transitions between Secure state and Non-secure state.
Secure state	The Arm Execution state that enables access to the Secure and Non-secure systems resources, for example memory, peripherals, and System registers.
SiP	Silicon Partner. In this document, the silicon manufacturer.
SMC	Secure Monitor Call. An Arm assembler instruction that causes an exception that is taken synchronously into EL3.
SMCCC	SMC Calling Convention, this document.
SMCCC Implementation	The firmware executing at either EL2, Secure EL2, or EL3 that is handling the SMC or HVC call in a manner that is compliant with this document.
SMC32/HVC32	32-bit SMC and HVC calling convention
SMC64/HVC64	64-bit SMC and HVC calling convention
SoC	System on Chip
Wx	A64 32-bit register view
Xx	A64 64-bit register view
Trusted OS	The Secure operating system that is running in the Secure EL1 Exception level. Trusted OS supports the execution of Trusted applications in Secure EL0.
Unknown Function Identifier	A reserved return code defined by SMCCC that indicates that the function is not implemented. The Unknown Function Identifier is declared as NOT_SUPPORTED in the interface specification and takes the value -1.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

If you have comments on the content of this book, send an e-mail to errata@arm.com. Give:

- The title (SMC Calling Convention)
- The number and issue (ARM DEN 0028 1.2 REL C)
- The page numbers to which your comments apply
- The rule identifiers to which your comments apply, if applicable
- A concise explanation of your comments

Arm also welcomes general suggestions for additions and improvements.

1 Introduction

This document defines a common calling mechanism to be used with the Secure Monitor Call (SMC) and Hypervisor Call (HVC) instructions in both the Armv7 and Armv8 architectures.

The SMC instruction is used to generate a synchronous exception that is handled by Secure Monitor code running in EL3. Arguments and return values are passed in registers. After being handled by the Secure Monitor, calls that result from the instructions can be passed on to a Trusted OS or some other entity in the Secure software stack.

The HVC instruction is used to generate a synchronous exception that is handled by a hypervisor running in EL2. Arguments and return values are passed in registers. Hypervisors can also trap SMC calls that are made by guest operating systems (at EL1), which allows the calls to be emulated, passed through, or denied as appropriate.

This specification aims to ease integration and reduce fragmentation between software layers, for example operating systems, hypervisors, Trusted OSs, Secure monitors, and system firmware.

Note: This document is defined for the Armv8-A Exception levels, EL0 to EL3.
The relationship between these Exception levels and the 32-bit Armv7 Exception levels is described in [2.]

2 SMC and HVC calling conventions

2.1 Secure Monitor Calls

In the Arm architecture, synchronous control is transferred between the normal Non-secure state and the Secure state through Secure Monitor Call (SMC) exceptions [1.][2.]. SMC exceptions are generated by the SMC instruction [1.][2.], and are handled by the Secure Monitor. The operation of the Secure Monitor is determined by the parameters that are passed in through registers.

2.2 Hypervisor Calls

Hypervisor Calls (HVCs) that are made by an operating system at EL1 result in a synchronous transfer of control to an EL2 hypervisor, and are regarded as HVC exceptions. The operation of the hypervisor is determined by the parameters that are passed in through registers.

2.3 Fast Calls and Yielding Calls

Two types of calls are defined:

- **Fast Calls** execute atomic operations. The call appears to be atomic from the perspective of the calling PE, and returns when the requested operation has completed
- **Yielding Calls** start operations that can be pre-empted by a Non-secure interrupt. The call can return before the requested operation has completed. Appendix A: Example implementation of Yielding Service calls provides an example of handling Yielding Calls.

2.4 32-bit and 64-bit conventions

For the SMC and HVC, two calling conventions instructions are defined:

- **SMC32/HVC32**: A 32-bit interface that can be used by either a 32-bit or a 64-bit client code, and passes up to seven 32-bit arguments. Because only SMC32 and HVC32 calls are used for the identification of Function Identifier ranges, the 32-bit calling convention is mandatory for all compliant systems, whether they are 32-bit or 64-bit systems. For more information, see Section 6.2.
- **SMC64/HVC64:** A 64-bit interface that can be used only by 64-bit client code, and passes up to seventeen 64-bit arguments. SMC64/HVC64 calls are expected to be the 64-bit equivalent to the 32-bit call, where applicable.

2.5 Function Identifiers

The **Function Identifier** is passed on W0 on every SMC and HVC call. Its 32-bit integer value indicates which function is being requested by the caller. It is always passed as the first argument to every SMC or HVC call in R0 or W0. The bit W0[31] determines if the call is Fast (W0[31]==1) or Yielding (W0[31]==0).

2.5.1 Fast Calls

In the Fast Call case (W0[31]==1), the bits W0[30:0] determine:

- The service to be invoked
- The function to be invoked

- The calling convention (32-bit or 64-bit) that is in use

Several bits within the 32-bit value have defined meanings valid for Fast Calls, as shown in Table 2-1.

Table 2-1 Bit usage within the SMC and HVC Function Identifier for Fast Call

<table>
<tr><th>Bit Numbers</th><th>Bit Mask</th><th>Description</th></tr>
<tr><td>31</td><td>0x80000000</td><td>Always set to 1 for Fast Calls.</td></tr>
<tr><td>30</td><td>0x40000000</td><td>If set to 0, the SMC32/HVC32 calling convention is used.
If set to 1, the SMC64/HVC64 calling convention is used.</td></tr>
<tr><td>29:24</td><td>0x3F000000</td><td>Service Call ranges. For more information, see Section 6.
<table>
<tr><th>Owning Entity Number</th><th>Bit Mask</th><th>Description</th></tr>
<tr><td>0</td><td>0x00000000</td><td>Arm Architecture Calls</td></tr>
<tr><td>1</td><td>0x01000000</td><td>CPU Service Calls</td></tr>
<tr><td>2</td><td>0x02000000</td><td>SiP Service Calls</td></tr>
<tr><td>3</td><td>0x03000000</td><td>OEM Service Calls</td></tr>
<tr><td>4</td><td>0x04000000</td><td>Standard Secure Service Calls</td></tr>
<tr><td>5</td><td>0x05000000</td><td>Standard Hypervisor Service Calls</td></tr>
<tr><td>6</td><td>0x06000000</td><td>Vendor Specific Hypervisor Service Calls</td></tr>
<tr><td>7-47</td><td>0x07000000 – 0x2F000000</td><td>Reserved for future use</td></tr>
<tr><td>48-49</td><td>0x30000000 – 0x31000000</td><td>Trusted Application Calls</td></tr>
<tr><td>50-63</td><td>0x32000000 – 0x3F000000</td><td>Trusted OS Calls</td></tr>
</table></td></tr>
<tr><td>23:16</td><td>0x00FF0000</td><td>Must be zero (MBZ), for all Fast Calls, when bit[31] == 1.
All other values are reserved for future use.
Note: Some Armv7 legacy Trusted OS Fast Call implementations have all bits set to 1.</td></tr>
<tr><td>15:0</td><td>0x0000FFFF</td><td>Function number within the range call type that is defined by bits[29:24].</td></tr>
</table>

2.5.2 Yielding Calls

In the Yielding Call case (W0[31]==0), Trusted OS Yielding Calls are placed in the 0x02000000-0x1FFFFFFF range. For more details on the Yielding Call Function ranges, see Table 6-2.

2.5.3 Conduits

Service calls are expected to be invoked through SMC instructions, except for Standard Hypervisor Calls and Vendor Specific Hypervisor Calls. On some platforms, however, SMC instructions are not available, and the

services can be accessed through HVC instructions. The method that is used to invoke the service is referred to as the conduit.

Table 2-2 describes which conduits are available, and how they depend on the Exception levels that are implemented.

Table 2-2: Dependence of conduits on implemented Exception levels

EL3 Implemented	EL2 Implemented	Conduits	Notes
Yes	Yes	SMC, HVC	
Yes	No	SMC	
No	Yes	HVC	Only permitted on Armv8-A
No	No	N/A	No conduit required

The SMC calling convention, however, does not specify which instruction, either SMC or HVC, to use to invoke a particular service.

2.6 SMC32/HVC32 argument passing

When the SMC32/HVC32 convention is used, an SMC or HVC instruction takes a Function Identifier and up to seven 32-bit register values as arguments, and returns the status and up to seven 32-bit register values.

When an SMC32/HVC32 call is made from AArch32:

- A Function Identifier is passed in register R0.
- Arguments are passed in registers R1-R7.
- Results are returned in R0-R7.
- The registers R4-R7 must be preserved unless they contain results, as specified in the function definition.
- Registers R8-R14 are saved by the function that is called, and must be preserved over the SMC or HVC call.

When an SMC32/HVC32 call is made from AArch64:

- A Function Identifier is passed in register W0.
- Arguments are passed in registers W1-W7.
- Results are returned in W0-W7.
- Registers W4-W7 must be preserved unless they contain results, as specified in the function definition.
- Registers X8-X30 and stack pointers SP_EL0 and SP_ELx are saved by the function that is called, and must be preserved over the SMC or HVC call.

Note: Unused result and scratch registers can leak information after an SMC or HVC call. An implementation can mitigate this risk by either preserving the register state over the call, or returning a constant value, such as zero, in each register.

Note: SMC32/HVC32 calls from AArch32 and AArch64 use the same physical registers for arguments and results, since register names W0-W7 in AArch64 map to register names R0-R7 in AArch32.

2.7 SMC64/HVC64 argument passing

When the SMC64/HVC64 convention is used, the SMC or HVC instruction takes a Function Identifier, up to seventeen 64-bit arguments in registers, and returns the status and up to seventeen 64-bit values in registers.

When an SMC64/HVC64 call is made from AArch64:

- A Function Identifier is passed in register W0.
- Arguments are passed in registers X1-X17.
- Results are returned in X0-X17.
- Registers X4-X17 must be preserved unless they contain results, as specified in the function definition.
- Registers X18-X30 and stack pointers SP_EL0 and SP_ELx are saved by the function that is called, and must be preserved over the SMC or HVC call.

This calling convention cannot be used by code executing in AArch32 state.

- Any SMC64/HVC64 call from AArch32 state receives the "Unknown Function Identifier" result, see section 5.2.

Note: Unused result and scratch registers can leak information after an SMC or HVC call. An implementation can mitigate against this risk by either preserving the register state over the call, or returning a constant value, such as zero, in each register.

2.8 SVE, SIMD and floating-point registers

SVE, SIMD, and floating-point registers must not be used to pass arguments to or receive results from any SMC or HVC call that complies with this specification.

The SMCCC implementation must ensure that the live state, belonging to the calling context, on all SVE [11.], Advanced SIMD and floating-point registers, is preserved over all SMC and HVC calls.

The live state to be preserved is present in the following registers:

- Z0-Z31, P0-P15, FFR, FPSR and FPCR, if SVE is implemented
- V0-V31, FPSR and FPCR, if SVE is not implemented

The SMCCC implementation is responsible for ensuring that information is not disclosed between execution contexts through SVE, SIMD, and floating-point registers.

Arm recommends that the SMCCC implementation adopts a design pattern, for SVE state preservation, from the set of patterns that are described in [14.].

2.9 SMC and HVC immediate value

The SMC and HVC instructions encode an immediate value, as defined by the Arm architecture [1.][2.]. The size of this immediate value, and mechanisms to access the value, differ between the Arm instruction sets. Also, it is time-consuming for 32-bit Secure Monitor code to access this immediate value.

Therefore:

- For all compliant calls, an SMC or HVC immediate value of zero must be used.
- Nonzero immediate values in SMC instructions are reserved.
- Nonzero immediate values in HVC instructions are designated for use by hypervisor vendors.

2.10 Client ID (optional)

Provisions have been made for Secure software to track and index client IDs.

2.10.1 SMC calls

If an implementation includes a hypervisor or similar supervisory software that executes at EL2, it might be necessary to identify the client operating system from which the SMC call originated.

- A 16-bit client ID parameter is optionally defined for SMC calls.
- In AArch32, the client ID is passed in the R7 register. For more information, see Table 4-1.
- In AArch64, the client ID is passed in the W7 register. For more information, see Table 3-1.
- The client ID of `0x0000` is designated for SMC calls from the hypervisor itself.

The client ID is expected to be created within the hypervisor and used to register, reference, and de-register client operating systems to a Trusted OS. It is not expected to correspond to the VMIDs used by the MMU.

If a client ID is implemented, all SMC calls that are generated by software executing at EL1 must be trapped by the hypervisor. Identification information must be inserted into R7 or W7 register before forwarding any SMC call on to the Secure Monitor.

If no hypervisor is implemented, the Guest OS is not required to set the client ID value.

2.10.2 HVC calls

The Client ID is ignored by the HVC calling convention.

2.11 Secure OS ID (optional)

In the presence of multiple Secure operating systems at S-EL1, the caller must specify the Secure OS for which the call is intended:

- An optional 16-bit Secure OS ID parameter can be defined for SMC calls.
- In AArch32, the Secure OS ID is passed in the R7 register. For more information, see Table 4-1.
- In AArch64 state, the Secure OS ID is passed in the W7 register. For more information, see Table 3-1.

2.12 Session ID (optional)

To support multiple sessions within a Trusted OS or hypervisor, it might be necessary to identify multiple instances of the same SMC or HVC call:

- An optional 32-bit Session ID can be defined for SMC and HVC calls.
- In AArch32, the Session ID is passed in the R6 register, see Table 4-1.
- In AArch64, the Session ID is passed in the W6 register, see Table 3-1.

The Session ID is expected to be provided by the Trusted OS or hypervisor, and is used by its clients in subsequent calls.

3 AArch64 SMC and HVC calling conventions

This specification defines common calling mechanisms for use with the SMC and HVC instructions from the AArch64 state. These calling mechanisms are referred to as SMC32/HVC32 and SMC64/HVC64.

For Arm AArch64:

- All Trusted OS and Secure monitor implementations must conform to this specification.
- All hypervisors must implement the Standard Secure and Hypervisor Service calls.

3.1 Register use in AArch64 SMC and HVC calls

For the AArch64 calling conventions, usage of the architectural registers is defined in Table 3-1.

The working size of the register is identified by its name:

Xn All 64-bits are used.

Wn The least significant 32-bits are used, and the most significant 32-bits are zero. Implementations must ignore the most significant bits.

Table 3-1 Register Usage in AArch64 SMC32, HVC32, SMC64, and HVC64 calls

Register Name		Role during SMC or HVC call		
SMC32/HVC32	SMC64/HVC64	Calling values	Modified	Return state
SP_ELx		ELx stack pointer	No	Register values are preserved.
SP_EL0		EL0 stack pointer	No	
X30		The Link Register	No	
X29		The Frame Pointer	No	
X19…X28		Registers that are saved by the called function	No	
X18		The Platform Register	No	
X17		Parameter register The second intra-procedure-call scratch register	Dependent[1]	Registers values are preserved or contain call results.
X16		Parameter register The first intra-procedure-call scratch register	Dependent[1]	
X9…X15		Parameter registers Temporary registers	Dependent[1]	
X8		Parameter register Indirect result location register	Dependent[1]	

W7	X7 (or W7)	Parameter register Optional Client ID in bits[15:0] (ignored for HVC calls) Optional Secure OS ID in bits[31:16]	Dependent[1]	
W6	X6 (or W6)	Parameter register Optional Session ID register	Dependent[1]	
W4…W5	X4…X5	Parameter registers	Dependent[1]	
W1…W3	X1…X3	Parameter registers	Yes	SMC and HVC result registers
W0	W0	Function Identifier	Yes	

For more information, see [4.].

[1] An SMC call or HVC call can return results in this register. Otherwise the call must preserve the value in the register. Refer to the documentation for the defined behavior of each SMC or HVC call.

4 AArch32 SMC and HVC calling convention

This specification defines a common calling mechanism for use with the SMC and HVC instructions from the AArch32 state, which are referred to as SMC32/HVC32.

Note: Arm recognizes that some vendors already use a proprietary calling convention and are not able to meet all the following requirements.

4.1 Register use in AArch32 SMC and HVC Calls

Table 4-1 Register usage in AArch32 SMC and HVC Calls

Register SMC32/HVC32	Role during SMC or HVC call		
	Calling values	Modified	Return state
R15	The Program Counter	Yes	Next instruction
R14	The Link Register	No	Unchanged registers are saved or restored.
R13	The stack pointer	No	
R12	The Intra-Procedure-call scratch register	No	
R11	Variable-register 8	No	
R10	Variable-register 7	No	
R9	Platform register	No	
R8	Variable-register 5	No	
R7	Parameter register 7 Optional Client ID in bits[15:0] (ignored for HVC calls) Optional Secure OS ID in bits[31:16]	Dependent[2]	Register values are preserved or contain call results.
R6	Parameter register 6 Optional Session ID	Dependent[2]	
R5	Parameter register 5	Dependent[2]	
R4	Parameter register 4	Dependent[2]	
R3	Parameter register 3	Yes	SMC and HVC results registers
R2	Parameter register 2	Yes	
R1	Parameter register 1	Yes	
R0	Function Identifier	Yes	

For more information, see [3.].

All architecturally-banked registered must be preserved over AArch32 calls.

[2] An SMC call or HVC call can return results in this register. Otherwise the call must preserve the value in the register. Refer to the documentation for the defined behavior of each SMC or HVC call.

5 SMC and HVC results

5.1 Error codes

Errors codes that are returned in R0, W0 and X0 are signed integers of the appropriate size:

- In AArch32:
  - When using the SMC32/HVC32 calling convention, error codes, which are returned in R0, are 32-bit signed integers.
- In AArch64:
  - When using the SMC64/HVC64 calling convention, error codes, which are returned in X0, are 64-bit signed integers.
  - When using the SMC32/HVC32 calling convention, error codes, which are returned in W0, are 32-bit signed integers. X0[63:32] is UNDEFINED.

5.2 Unknown Function Identifier

The Unknown SMC Function Identifier is a sign-extended value of (-1) that is returned in the R0, W0 or X0 registers. An implementation must return this error code when it receives:

- An SMC or HVC call with an unknown Function Identifier
- An SMC or HVC call for a removed Function Identifier
- An SMC64/HVC64 call from AArch32 state

Note: The Unknown Function Identifier must not be used to discover the presence of an SMC or HVC function, or that lack of a function. Function Identifiers must be determined from the UID and Revision information. For the Arm Architecture Call range, Function Identifiers can be determined using SMCCC_ARCH_FEATURES as described in Section 7.3 and Appendix B: Discovery of Arm Architecture Service functions.

5.3 Unique Identification format

This value identifies the implementer of a subrange (see 6.2) of the API, and therefore what controls the actions of SMCs in that subrange.

The Unique Identification UID is a UUID as defined by RFC 4122 [6.]. These UUIDs must be generated by any method that is defined by RFC 4122 [6.], and are 16-byte strings.

UIDs are returned as a single 128-bit value using the SMC32 calling convention. This value is mapped to argument registers as shown in Table 5-1.

Table 5-1: UUID register mapping

Register		Value
AArch32	AArch64	
R0	W0	Bytes 0…3 with byte 0 in the low-order bits
R1	W1	Bytes 4…7 with byte 4 in the low-order bits
R2	W2	Bytes 8…11 with byte 8 in the low-order bits

R3	W3	Bytes 12…15 with byte 12 in the low-order bits

UIDs with the least significant 32 bits set to `0xFFFFFFFF` must not be used, because they are indistinguishable from Unknown Function Identifiers.

There can be many implementers of standard APIs. The API compatibility is determined by revision numbers.

5.4 Revision information format

The revision information for a subrange (see 6.2) is defined by a 32-bit major version and a 32-bit minor version.

Different major version values indicate a possible incompatibility between SMC and HVC APIs for the affected SMC and HVC range.

For two revisions, A and B, where the major version values are identical, and the minor version value of revision B is greater than the minor version value of revision A, every SMC and HVC instruction in the affected range that works in revision A must also work in revision B, with a compatible effect.

When returned by a call, the major version is returned in R0 or W0 and the minor version is returned in R1 or W1. Such an SMC or HVC instruction must use the SMC32 or HVC32 calling conventions.

The rules for interface updates are:

- A Function Identifier, when issued, must never be reused.
- Subsequent SMC or HVC calls must take a new unused Function Identifier.
- Calls to Function Identifiers that have been removed must return the Unknown Function Identifier value.
- Incompatible argument changes cannot be made to an existing SMC or HVC call. A new call is required.
- Major revision numbers must be incremented when:
    - Any SMC or HVC call is removed.
- Minor revision numbers must be incremented when:
    - Any SMC or HVC call is added.
    - Backwards compatible changes are made to existing function arguments.

6 Function Identifier Ranges

Arm defines the SMC and HVC Fast Call services that are listed in Table 6-1.

Table 6-1: SMC and HVC Services

Service	Owning Entity Number	Comment
Arm Architecture Service	0	Provides interfaces to generic services for the Arm Architecture.
CPU Service	1	Provides interfaces to CPU implementation-specific services for this platform, for example access to errata workarounds.
SiP Service	2	Provides interfaces to SoC implementation-specific services on this platform, for example Secure platform initialization, configuration, and some power control services.
OEM Service	3	Provides interfaces to OEM-specific services on this platform.
Standard Secure Service	4	Standard Service Calls for the management of the overall system. By standardizing such calls, the job of implementing operating systems on Arm is made easier. Section 6.3 lists Secure Services that are already defined.
Standard Hypervisor Service	5	Standardized Hypervisor Service Calls allow for common hypervisor discovery mechanism from any Guest OS.
Vendor Specific Hypervisor Service	6	Proprietary Hypervisor Service Calls.
Trusted Applications	48—49	
Trusted OS	50—63	

The existing Arm Architecture Services are listed in Section 7. An example of a set of services residing in the Standard Secure Services range is PSCI, defined in [5.].

6.1 Allocation of values

Table 6-2 shows the recommended allocation of Function Identifier value ranges for different entities and purposes. The owner of a range is the entity that is responsible for that function in a specific SoC. Any subranges, in the 0x0000 0000 – 0xFFFF FFFF range, that are not covered by the table are reserved.

Table 6-2: Allocated subranges of Function Identifier to different services

SMC Function Identifier	Service type
0x00000000-0x0100FFFF	Reserved for existing APIs This region is already in use by the existing Armv7 devices.
0x02000000-0x1FFFFFFF	Trusted OS Yielding Calls
0x20000000-0x7FFFFFFF	Reserved for future expansion of Trusted OS Yielding Calls
0x80000000-0x8000FFFF	SMC32: Arm Architecture Calls
0x81000000-0x8100FFFF	SMC32: CPU Service Calls

0x82000000-0x8200FFFF	SMC32: SiP Service Calls
0x83000000-0x8300FFFF	SMC32: OEM Service Calls
0x84000000-0x8400FFFF	SMC32: Standard Service Calls
0x85000000-0x8500FFFF	SMC32: Standard Hypervisor Service Calls
0x86000000-0x8600FFFF	SMC32: Vendor Specific Hypervisor Service Calls
0x87000000-0xAF00FFFF	Reserved for future expansion
0xB0000000-0xB100FFFF	SMC32: Trusted Application Calls
0xB2000000-0xBF00FFFF	SMC32: Trusted OS Calls
0xC0000000-0xC000FFFF	SMC64: Arm Architecture Calls
0xC1000000-0xC100FFFF	SMC64: CPU Service Calls
0xC2000000-0xC200FFFF	SMC64: SiP Service Calls
0xC3000000-0xC300FFFF	SMC64: OEM Service Calls
0xC4000000-0xC400FFFF	SMC64: Standard Service Calls
0xC5000000-0xC500FFFF	SMC64: Standard Hypervisor Service Calls
0xC6000000-0xC600FFFF	SMC64: Vendor Specific Hypervisor Service Calls
0xC7000000-0xEF00FFFF	Reserved for future expansion
0xF0000000-0xF100FFFF	SMC64: Trusted Application Calls
0xF2000000-0xFF00FFFF	SMC64: Trusted OS Calls

6.2 General service queries

The following general queries are optional:

- **Call Count Query**[3]– Returns a 32-bit count of the available service calls. The count includes both 32 and 64 calling convention service calls and both Fast Calls and Yielding Calls.
- **Call UID Query** – Returns a unique identifier of the service provider, as specified in section 5.3.
- **Revision Query** – Returns revision details of the service implementor, as specified in section 5.4.

All queries listed in Table 6-3 are SMC32 or HVC32 Fast Calls.

When implemented, the general service queries must use the reserved function IDs that are defined in Table 6-3. The reserved function IDs must only be used for the calls that are listed in Table 6-3.

If the service or the query is not implemented, general service queries must return the Unknown Function Identifier error.

Table 6-3 Function Identifier values of general queries

Service	Call Count	Call UID	Revision
Arm Architecture Service	0x8000_FF00[3]	0x8000_FF01[3]	0x8000_FF03[3]

[3] Query deprecated from SMCCC v1.2 onward

CPU Service	0x8100_FF00[3]	0x8100_FF01	0x8100_FF03
SiP Service	0x8200_FF00[3]	0x8200_FF01	0x8200_FF03
OEM Service	0x8300_FF00[3]	0x8300_FF01	0x8300_FF03
Standard Secure Service	0x8400_FF00[3]	0x8400_FF01	0x8400_FF03
Standard Hypervisor Service	0x8500_FF00[3]	0x8500_FF01	0x8500_FF03
Vendor Specific Hypervisor Service	0x8600_FF00[3]	0x8600_FF01	0x8600_FF03
Trusted Applications[4]	-	-	-
Trusted OS	0xBF00_FF00[3]	0xBF00_FF01	0xBF00_FF03

In addition to the values in the table above, the other Function Identifiers in the `0xXXXXFF00 – 0xXXXXFFFF` range, for example `0x8000FF02` and `0x8000FF04-0x8000FFFF` for Arm Architecture Service, are reserved for future expansion. The Call Count Query of all services, the Call UID Query and the Revision Query from the Arm Architecture Service, are deprecated from SMCCC v1.2 onward.

6.3 Implemented Standard Secure Service Calls

Arm defines a set of Standard Secure Service Calls for the management of the overall system. Standard calls are intended to provide system management services to operating systems.

The following Function Identifier values are reserved for the following Standard Secure Service Calls:

Table 6-4: Reserved Standard Secure Service Call range

Function Identifier	Reserved for	Notes
0x84000000-0x8400001F	PSCI 32-bit calls	32-bit Power Secure Control Interface. For details of functions and arguments, see [5.].
0xC4000000-0xC400001F	PSCI 64-bit calls	64-bit Power Secure Control Interface. For details of functions and arguments, see [5.].
0x84000020-0x8400003F	SDEI 32-bit calls	32-bit Software Delegated Exception Interface. For details of functions and arguments, see [8.].
0xC4000020-0xC400003F	SDEI 64-bit calls	64-bit Software Delegated Exception Interface. For details of functions and arguments, see [8.].
0x84000040-0x84000041	MM 32-bit calls	32-bit Management Mode. For details of functions and arguments, see [10.].
0xC4000040-0xC4000041	MM 64-bit calls	64-bit Management Mode. For details of functions and arguments, see [10.].

[4] It is the responsibility of a Trusted OS to identify and describe the services that are provided by Trusted applications.

Note: When a hypervisor traps SMC calls, it must be able to determine from the Standard Service identifiers which SMC calls are for power control and similar operations, so that it can emulate these calls for its clients. Sometimes the standards defining these service calls might allow use of HVC instead of SMC, either to support platforms that do not implement EL3, or to allow more flexibility for the hypervisor implementation, in which case the identifiers remain the same.

7 Arm Architecture Calls

7.1 Return codes

Table 7-1 defines the possible values for the error codes that are used with the interface functions. The error return type is signed integer. Zero and positive values denote success, and negative values indicate error.

Table 7-1 Return code and values

Name	Description	Value
SUCCESS	The call completed successfully.	0
NOT_SUPPORTED	The call is not supported by the implementation.	-1
NOT_REQUIRED	The call is deemed not required by the implementation.	-2
INVALID_PARAMETER	One of the call parameters has a non-supported value.	-3

7.2 SMCCC_VERSION

Dependency	MANDATORY from SMCCC v1.1 OPTIONAL for SMCCC v1.0		
Description	Retrieve the implemented version of the SMC Calling Convention		
Parameters	uint32 Function ID	0x80000000	
Return	int32	NOT_SUPPORTED	Treat as v1.0
		major:minor	Bit[31] must be zero Bits [30:16] Major version Bits [15:0] Minor version

7.2.1 Usage

This call is used by system software to determine the version of SMCCC that is implemented. The version that is implemented indicates the calling convention for AArch64 callers and the presence of the SMCCC_ARCH_FEATURES function.

7.2.2 Discovery

This function was not defined in SMCCC version 1.0, and might not be safe on all platforms. Calling software can detect the implementation of this function by one of the following methods:

- Built-in knowledge of the firmware implementation
- Discovery via PSCI_FEATURES with the psci_func_id parameter set to 0x8000 0000. For more information, see [5.], Appendix C: Changes to PSCI and Appendix B: Discovery of Arm Architecture Service functions.
- Information from firmware tables like the Flattened Device Tree table or ACPI tables.

See Appendix B: Discovery of Arm Architecture Service functions for a description of the full discovery sequence.

If SMCCC_VERSION is implemented, calling SMCCC_ARCH_FEATURES with arch_func_id equal to 0x8000 0000 will return SUCCESS.

7.2.3 Caller responsibilities

Before calling this function, Arm recommends that the caller determines whether it is safe to do so on the platform. This is because some firmware implementations do not implement this function safely. See Appendix B: Discovery of Arm Architecture Service functions for the recommended discovery protocol.

The caller must interpret a NOT_SUPPORTED response as indicating the presence of firmware that implements SMCCC v1.0.

7.2.4 Implementation responsibilities

For the values that must be returned by this call, see Table 11-1 in Appendix D: Changelog.

7.3 SMCCC_ARCH_FEATURES

Dependency	MANDATORY from SMCCC v1.1 OPTIONAL for SMCCC v1.0		
Description	Determine the availability and capability of Arm Architecture Service functions		
Parameters	uint32 Function ID	0x80000001	
	uint32 arch_func_id	Function ID of an Arm Architecture Service Function	
Return	int32	< 0	Function not implemented or arch_func_id not in Arm Architecture Service range. The reason is indicated by an error code specific to the function.
		SUCCESS	Function implemented.
		> 0	Optional Function implemented. Function capabilities are indicated using feature flags specific to the function.

7.3.1 Usage

This call is used by system software to determine whether a specific Arm Architecture Service function is implemented in the firmware. This function might also provide information about the capabilities of the function.

7.3.2 Discovery

The implementation of this function can be detected by checking the SMCCC version. This function is mandatory if SMCCC_VERSION indicates that version 1.1 or later is implemented.

See Appendix B: Discovery of Arm Architecture Service functions for the full discovery sequence.

If SMCCC_ARCH_FEATURES is implemented, calling SMCCC_ARCH_FEATURES with arch_func_id equal to 0x8000 0001 will return SUCCESS.

7.3.3 Parameters

The arch_func_id parameter is the Function ID in the Arm Architecture Service range: 0x8000 0000-0x8000 FFFF and 0xC000 0000-0xC000 FFFF. Values outside of this range are invalid.

7.3.4 Return

If the result is non-negative it indicates that the return function is implemented.

Some functions provide information about their capabilities in the result.

A description of how to interpret the result of calling SMCCC_ARCH_FEATURES is provided in the Discovery section of the documentation for each function.

7.3.5 Caller responsibilities

The caller must only call SMCCC_ARCH_FEATURES on implementations that are compliant with SMCCC version 1.1 or later.

7.3.6 Implementation responsibilities

This function must return SUCCESS when arch_func_id is the SMCCC_VERSION or SMCCC_ARCH_FEATURES function id.

7.4 SMCCC_ARCH_SOC_ID

Dependency	OPTIONAL from SMCCC v1.0		
Description	Obtain a SiP defined SoC identification value		
Parameters	uint32 Function ID	0x80000002	
	uint32 SoC_ID_type	0: SoC version 1: SoC revision 2 – 0xFFFF FFFF: Reserved	
Return	int32	INVALID_PARAMETER	
		> 0	SoC_ID_type == 0 JEP-106 code for the SiP Bit[31] must be zero Bits [30:24] JEP-106 bank index for the SiP (see [9.])[5] Bits [23:16] JEP-106 identification code with parity bit for the SiP (see [9.])[5] Bits [15:0] Implementation defined SoC ID

[5] As an example, the value of JEP-106 manufacturer code in Arm's case is:
Bits[30:24] = 0x04 (the bank index is equal to the for continuation code bank number -1)
Bits [23:16] = 0x3B

			SoC_ID_type == 1 Bit[31] must be zero Bits [30:0] SoC revision

7.4.1 Usage

This call is used by system software to obtain the SiP defined SoC identification details.

7.4.2 Discovery

The implementation of this function can be detected by calling SMCCC_ARCH_FEATURES (see 7.3) with arch_func_id equal to 0x8000 0002. The result of that call should be interpreted as follows:

NOT_SUPPORTED	Function is not implemented
0	Function is implemented

7.4.3 Parameters

The SoC_ID_type parameter value identifies the type of SoC identification that the function returns. The valid values for the SoC_ID_type parameter are:

- 0: SoC version: Function returns the SiP defined SoC version.
- 1: SoC revision: Function returns the SiP defined SoC revision.

Any other SoC_ID_type parameter value is invalid.

7.4.4 Return

If the call returns NOT_IMPLEMENTED the function is not present in the SMCCC implementation. If the call returns INVALID_PARAMETER, the SoC_ID_type parameter is outside of the {0,1} set.

On success the return value is a positive quantity which represents:

- SoC version, if SoC_ID_type == 0
- SoC revision, if SoC_ID_type == 1

7.4.5 Caller responsibilities

The caller must not call this function unless it has determined that it is implemented in the firmware, see Section 7.4.2.

7.4.6 Implementation responsibilities

If implemented, the firmware must provide discovery of this function as defined in the Section 7.4.2.

7.5 SMCCC_ARCH_WORKAROUND_1

Dependency	OPTIONAL from SMCCC v1.1 NOT SUPPORTED in SMCCC v1.0	
Description	Execute the mitigation for CVE-2017-5715 on the calling PE	
Parameters	uint32 Function ID	0x8000 8000

Return	void	This function has no return value.

7.5.1 Usage

This call is used by system software to execute a firmware workaround that is required to mitigate CVE-2017-5715.

7.5.2 Discovery

The implementation of this function can be detected by calling SMCCC_ARCH_FEATURES, as described in Section 7.3, with arch_func_id equal to 0x8000 8000. The result of that call should be interpreted as:

NOT_SUPPORTED	The discovery call will return NOT_SUPPORTED on every PE in the system. SMCCC_ARCH_WORKAROUND_1 must not be invoked on any PE in the system. Either: None of the PEs in the system require firmware mitigation for CVE-2017-5715. The system contains at least 1 PE affected by CVE-2017-5715 that has no firmware mitigation available. The firmware does not provide any information about whether firmware mitigation is required.
0	SMCCC_ARCH_WORKAROUND_1 can be invoked safely on all PEs in the system. The PE on which SMCCC_ARCH_FEATURES is called requires firmware mitigation for CVE-2017-5715.
1	SMCCC_ARCH_WORKAROUND_1 can be invoked safely on all PEs in the system. The PE on which SMCCC_ARCH_FEATURES is called does not require firmware mitigation for CVE-2017-5715.

7.5.3 Caller responsibilities

The caller must not call this function unless it has determined that it is implemented in the firmware, see Section 7.5.2.

Arm recommends that the caller only call this function on PEs that are affected by CVE-2017-5715 when a firmware-based mitigation is required and a local workaround is infeasible. Calling this on other PEs is wasted execution.

For more information, see [7.] and [13.].

7.5.4 Implementation responsibilities

This function must not be provided in firmware implementations that are not compliant with SMCCC version 1.1 or later.

If implemented, the firmware must provide discovery of this function as defined in the Section 7.5.2.

Arm recommends that firmware does not provide an implementation of this function on systems that return a negative error code in the discovery call above.

If implemented, the firmware must fully implement this function for all PEs in the system that require firmware mitigation for CVE-2017-5715.

In heterogeneous systems with some PEs that require mitigation and others that do not, the firmware must provide a safe implementation of this function on all PEs. This allows callers to call the function on all PEs in a system where the firmware implements the workaround, without risking functional stability. In such systems, on

PEs that do not require firmware mitigation, the firmware must provide an implementation that has no effect. For more information, see [7.] and [13.].

7.6 SMCCC_ARCH_WORKAROUND_2

Dependency	OPTIONAL from SMCCC v1.1 NOT SUPPORTED in SMCCC v1.0	
Description	Enable or disable the mitigation for CVE-2018-3639 on the calling PE	
Parameters	uint32 Function ID	0x8000 7FFF
	uint32 enable	A non-zero value indicates that the mitigation for CVE-2018-3639 must be enabled. A value of zero indicates that it must be disabled.
Return	Void	This function has no return value.

7.6.1 Usage

This call is used by system software to enable or disable a firmware workaround that is required to mitigate CVE-2018-3639. The call only affects the mitigation state (enabled or disabled) for the calling execution context. The workaround is enabled by default for all execution contexts that are managed by the firmware. Once the workaround is disabled, it remains disabled until explicitly re-enabled by a subsequent call to this function.

7.6.2 Discovery

The implementation of this function can be detected by calling SMCCC_ARCH_FEATURES, as described in Section 7.3, with arch_func_id equal to 0x8000 7FFF. The result of that call should be interpreted as:

NOT_SUPPORTED	The discovery call will return NOT_SUPPORTED on every PE in the system. SMCCC_ARCH_WORKAROUND_2 must not be invoked on any PE in the system. Either: The system contains at least one PE affected by CVE-2018-3639 that has no firmware mitigation available, or The firmware does not provide any information about whether firmware mitigation is required or enabled.
NOT_REQUIRED	The discovery call will return NOT_REQUIRED on every PE in the system. SMCCC_ARCH_WORKAROUND_2 must not be invoked on any PE in the system. For all PEs in the system, firmware mitigation for CVE-2018-3639 is either permanently enabled or not required.
0	SMCCC_ARCH_WORKAROUND_2 can be invoked safely on all PEs in the system. The PE on which SMCCC_ARCH_FEATURES is called requires dynamic firmware mitigation for CVE-2018-3639 using SMCCC_ARCH_WORKAROUND_2.
1	SMCCC_ARCH_WORKAROUND_2 can be invoked safely on all PEs in the system. The PE on which SMCCC_ARCH_FEATURES is called does not require dynamic firmware mitigation for CVE-2018-3639 using SMCCC_ARCH_WORKAROUND_2.

7.6.3 Caller responsibilities

The caller must not call this function unless it has determined that this function is implemented in the firmware, see 7.6.2.

Arm recommends that the caller only call this on PEs that are affected by CVE-2018-3639 when a dynamic firmware-based mitigation is required, and a local workaround is infeasible. Calling this on other PEs is wasted execution.

Arm recommends that Secure world software does not use this call so that it always remains protected.

For more information, see the [7.] and [13.].

7.6.4 Implementation responsibilities

This function must not be provided in firmware implementations that are not compliant with SMCCC version 1.1 or later.

If implemented, the firmware must provide discovery of this function as defined in the Discovery section above.

Arm recommends that firmware does not provide an implementation of this function on systems that return a negative error code in the discovery call above.

If implemented, the firmware must fully implement this function for all PEs in the system that require dynamic firmware mitigation for CVE-2018-3639.

In heterogeneous systems with some PEs that require dynamic firmware mitigation and others that do not, the firmware must provide a safe implementation of this function on all PEs. This permits callers to call the function on all PEs in a system where the firmware implements the workaround, without risking functional stability. In such systems, on PEs that do not require dynamic firmware mitigation, the firmware must provide an implementation that has no effect.

If implemented, the firmware must separately maintain the logical mitigation state (enabled or disabled) for each execution context that it manages. The default mitigation state (enabled) must be applied:

- To the primary PE following cold boot
- To a PE when it starts up following a CPU_ON call, as defined by the PSCI specification [5.]
- To a PE when it wakes up from a powerdown state (for example, following a CPU_SUSPEND call), as defined by the PSCI specification [5.]

If implemented, Arm recommends that the firmware enables mitigation during its own execution.

If the firmware implements this function and the Software Delegated Exception Interface (SDEI) specification [8.], then the firmware must apply the default mitigation state (enabled) to the execution context of each SDEI client handler following each triggered event, irrespective of the mitigation state of the interrupted or client execution contexts. The firmware must restore the mitigation state of the interrupted or client execution context following a call to SDEI_EVENT_COMPLETE or SDEI_EVENT_COMPLETE_AND_RESUME respectively.

For more information, see [7.] and [13.].

8 Appendix A: Example implementation of Yielding Service calls

Many aspects of Yielding Calls are specific to the internal implementation of Secure services, for example:

- Might a Yielding Call be resumed at another PE?
- Might there be more outstanding Yielding Calls per PE?

One approach for implementing Yielding Calls is for the Secure service to return a specific error code when that call is pre-empted by an interrupt, as shown in the following example. The caller can then resume the operation after the interrupt is serviced.

To allow Secure services to match the original Yielding Call after it resumes, the service might return opaque handlers that can be passed back in resume_call():

```
(X0, X1, X2) = any_yielding_call(…);
while (X0 == SMCCC_PREEMPTED)
{
  (X0, X1, X2) = resume_call(X1, X2);
}
```

9 Appendix B: Discovery of Arm Architecture Service functions

System software needs to run safely on any existing platform, but should make use of the mitigation functionality whenever it is available. The following approach to discovery should maximize the ability to detect this functionality without causing unsafe behavior on existing platforms.

Step 1: Determine if SMCCC_VERSION is implemented

The following pseudocode summarizes the proposed discovery flow using PSCI 1.0:

```
if (FirmwareTablesLookup(PSCI) == SUCCESS)
{
    if (invoke_psci_version() >= 0x10000)
    {
        if (invoke_psci_features(SMCCC_VERSION) == SUCCESS)
            return SUCCESS;
    }
}
return NOT_SUPPORTED
```

The steps are:

1. Use firmware data, device tree PSCI node, or ACPI FADT PSCI flag, to determine whether a PSCI implementation is present.
2. Use PSCI_VERSION to learn whether PSCI_FEATURES is provided. PSCI_FEATURES is mandatory from version 1.0 of PSCI.
3. Use PSCI_FEATURES with the SMCCC_VERSION Function Identifier as a parameter to determine that SMCCC_VERSION is implemented.

In future, the ACPI and device tree might also be extended to indicate the compliance to the SMCCC directly.

Step 2: Determine if Arm Architecture Service function is implemented

The following pseudocode summarizes the proposed discovery flow of an Arm Architecture Service function, using SMCCC_ARCH_WORKAROUND_1 as an example:

```
if (invoke_smccc_version() >= 0x10001)
{
    if (invoke_smccc_arch_features(SMCCC_ARCH_WORKAROUND_1) >= 0)
        return SUCCESS;
}
return NOT_SUPPORTED
```

The steps are:

1. Use SMCCC_VERSION to learn whether the calling convention complies to version 1.1 or above.

2. Use SMCCC_ARCH_FEATURES to query whether *the Arm Architecture Service* function is implemented on this system.

If the software is running on a heterogenous system, for example big.LITTLE, it can optimize use of an Arm Architecture Service function by invoking SMCCC_ARCH_FEATURES on each PE and eliminating the calls to the function on PEs that indicate the function call is not required, for example on PEs that return one (1) in the case of `SMCCC_ARCH_WORKAROUND_1`.

10 Appendix C: Changes to PSCI

10.1 Overview

10.1.1 Discoverability of SMCCC implementation

On platforms that fully implement the SMC Calling Convention version 1.0 (SMCCCv1.0) the functions that are described in Section 7.2 and Section 7.3 are adequate for detecting the presence of the Arm Architecture Service functionality.

However, some platform and virtualization firmware only implement a subset of SMCCC. Specifically, such firmware might only implement the PSCI specification in order to meet operating system requirements, but not provide a safe implementation of unimplemented SMCCC functions. A safe implementation is one that conforms to the SMCCC calling convention and returns NOT_SUPPORTED (-1) for functions that are not defined or not implemented.

System software that needs to use the mitigation functions that are described above, but must also run correctly on such platforms, requires additional mechanisms to discover whether SMCCC is implemented, and whether it is safe to call SMCCC_VERSION.

One mechanism is to add discoverability of this to the firmware description, for example Device Tree or ACPI tables.

To accelerate adoption of these mitigations and protect more systems more rapidly from this vulnerability, Arm strongly recommends that the firmware implementations also provide an additional discovery mechanism though the firmware PSCI implementation of PSCI_FEATURES.

10.2 Interface

This is an updated excerpt from the PSCI specification for PSCI_FEATURES. The significant changes are highlighted in gray. Note that the changes will apply from PSCI v1.0.

10.2.1 PSCI_FEATURES

Dependency	**Introduced in PSCI v1.0**		
Description	Query API that allows discovering whether a specific PSCI function is implemented and its features. See the PSCI specification [5.] for more details.		
Parameters	uint32 Function ID	0x8400000A	
	uint32 psci_func_id	Function ID for a PSCI Function or SMCCC_VERSION	
Return	int32	NOT_SUPPORTED	Function identified by psci_func_id not is not implemented or not valid
		SUCCESS	Function implemented
		> 0	Optional A set of feature flags associated with an implemented function identified by psci_func_id. Feature flags are specific to each function. In all cases the format is:

			Bit[31] is zero Bits[0:30] represent the feature flags. See PSCI [5.] for details.

Usage

The usage of this function is as defined for PSCI v1.0 and later. For more information, see [5.].

In addition, this interface can be used to detect the implementation of SMCCC_VERSION in the firmware.

Parameters

The psci_func_id parameter is valid if it is any of:

- A PSCI SMC32 Function Identifier, in the range 0x8400 0000-0x8400 001F
- A PSCI SMC64 Function Identifier, in the range 0xC400 0000-0xC400 001F
- The SMCCC_VERSION Function Identifier, 0x8000 0000

Return

For valid PSCI Function Identifiers, see PSCI for details of the return value.

When psci_func_id is SMCCC_VERSION, a return value of SUCCESS (zero) indicates that SMCCC_VERSION is implemented.

Implementation responsibilities

Arm recommends that this function reports the presence of SMCCC_VERSION for any firmware that implements SMCCC v1.1 as described in this specification.

11 Appendix D: Changelog

Table 11-1 relates the SMCCC version to the return of the SMCCC_VERSION Arm Architecture Call and to the changes introduced on each SMCCC version release.

For further details on the SMCCC_VERSION Arm Architecture Call return values, see Section 7.2.

Table 11-1 SMCCC Changelog

SMCCC version	SMCCC_VERSION call return	Changes
1.0	-1 or 0x10000	Introduces: General Service Queries (Section 6.2)
1.1	0x10001	Introduces: SMCCC_VERSION (Section 7.2) SMCCC_ARCH_FEATURES (Section 7.3) SMCCC_ARCH_WORKAROUND_1 (Section 7.5) SMCCC_ARCH_WORKAROUND_2 (Section 7.6)
1.2	0x10002	Argument/Result register set: Permits calls to use R4—R7 as return register (Section 4.1). Permits calls to use X4—X17 as return registers (Section 3.1). Permits calls to use X8—X17 as argument registers (Section 3.1). Introduces: SMCCC_ARCH_SOC_ID (Section 7.4) Deprecates: UID, Revision Queries on Arm Architecture Service (Section 6.2) Count Query on all services (Section 6.2)