SMC Calling Convention

Document number	DEN0028
Document quality	BET0
Document version	1.6 G
Document confidentiality	Non-confidential

Notice

This document is a Beta version of a specification undergoing review by Arm partners. It is provided to give advanced information only.

SMC Calling Convention

Release information

Date	Version	Changes
2025/Jan/28	1.6 G BET0	• Add SoC name to SMCCC_ARCH_SOC_ID • Add SMCCC_ARCH_WORKAROUND_4 • Allocated FIDs for the RHI in the Standard Hypervisor range • Update CCA FID reservation • Clarify that SMC32/HVC32 and SMC64/HVC64 function definitions can only differ in the range of the integer arg/returns.
2024/Apr/01	1.5 F	• Add Vendor Specific EL3 Monitor Service range. • Clarify that SMC/HVC callers from any Security state are supported. • Clarify that the RMM restricts the services exposed to Realms. • Rename "Secure Monitor" as "EL3 Monitor". • Minor text clarifications. • Clarify that the term hypervisor refers to Non-secure EL2 Software. • Add the SMCCC_ARCH_FEATURE_AVAILABILITY interface.
2022/May/01	E	• Add SME state management guidelines. • Update Standard Secure Service call list. • Add Standard Hypervisor Service call list. • Add SMCCC_ARCH_WORKAROUND_3 definition. • Add SMCCC deployment model description.
2021/Jan/01	D	• Add SVE absence of live state hint bit (FID[16]). • Minor text clarifications.
2020/Aug/01	C	• Merge information contained in DEN 0070 (see @ARMDEN0070). • 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.
2016/Nov/01	B	• HVC calling convention. • SMC calling convention clarifications and updates.
2013/Jun/01	A	• First release

Arm Non-Confidential Document License (“License”)

This License is a legal agreement between you and Arm Limited (“**Arm**”) for the use of Arm’s intellectual property (including, without limitation, any copyright) embodied in the document accompanying this License (“**Document**”). Arm licenses its intellectual property in the Document to you on condition that you agree to the terms of this License. By using or copying the Document you indicate that you agree to be bound by the terms of this License.

“**Subsidiary**” means any company the majority of whose voting shares is now or hereafter owner or controlled, directly or indirectly, by you. A company shall be a Subsidiary only for the period during which such control exists.

This Document is **NON-CONFIDENTIAL** and any use by you and your Subsidiaries (“Licensee”) is subject to the terms of this License between you and Arm.

Subject to the terms and conditions of this License, Arm hereby grants to Licensee under the intellectual property in the Document owned or controlled by Arm, a non-exclusive, non-transferable, non-sub-licensable, royalty-free, worldwide License to:

(i) use and copy the Document for the purpose of designing and having designed products that comply with the Document;

(ii) manufacture and have manufactured products which have been created under the License granted in (i) above; and

(iii) sell, supply and distribute products which have been created under the License granted in (i) above.

Licensee hereby agrees that the Licenses granted above shall not extend to any portion or function of a product that is not itself compliant with part of the Document.

Except as expressly licensed above, Licensee acquires no right, title or interest in any Arm technology or any intellectual property embodied therein.

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm’s view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

Reference by Arm to any third party’s products or services within this document is not an express or implied approval or endorsement of the use thereof.

THE DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. Arm may make changes to the Document at any time and without notice. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

NOTWITHSTANDING ANYTHING TO THE CONTRARY CONTAINED IN THIS LICENSE, TO THE FULLEST EXTENT PERMITTED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, IN CONTRACT, TORT OR OTHERWISE, IN CONNECTION WITH THE SUBJECT MATTER OF THIS LICENSE (INCLUDING WITHOUT LIMITATION) (I) LICENSEE’S USE OF THE DOCUMENT; AND (II) THE IMPLEMENTATION OF THE DOCUMENT IN ANY PRODUCT CREATED BY LICENSEE UNDER THIS LICENSE). THE EXISTENCE OF MORE THAN ONE CLAIM OR SUIT WILL NOT ENLARGE OR EXTEND THE LIMIT. LICENSEE RELEASES ARM FROM ALL OBLIGATIONS, LIABILITY, CLAIMS OR DEMANDS IN EXCESS OF THIS LIMITATION.

This License shall remain in force until terminated by Licensee or by Arm. Without prejudice to any of its other rights, if Licensee is in breach of any of the terms and conditions of this License then Arm may terminate this License immediately upon giving written notice to Licensee. Licensee may terminate this License at any time. Upon termination of this License by Licensee or by Arm, Licensee shall stop using the Document and destroy all copies of the Document in its possession. Upon termination of this License, all terms shall survive except for the License grants.

Any breach of this License by a Subsidiary shall entitle Arm to terminate this License as if you were the party in breach. Any termination of this License 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 License may be translated into other languages for convenience, and Licensee agrees that if there is any conflict between the English version of this License and any translation, the terms of the English version of this License shall prevail.

The validity, construction and performance of this License shall be governed by English Law.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

Arm document reference: PRE-21585

version 5.0, March 2024

Contents

SMC Calling Convention

Conventions

Typographical conventions

The typographical conventions are:

italic

Introduces special terminology, and denotes citations.

bold

Denotes signal names, and is used for terms in descriptive lists, where appropriate.

`monospace`

Used for assembler syntax descriptions, pseudocode, and source code examples.

Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.

SMALL CAPITALS

Used for some common terms such as IMPLEMENTATION DEFINED.

Used for a few terms that have specific technical meanings, and are included in the Glossary.

Red text

Indicates an open issue.

Blue text

Indicates a link. This can be

- A cross-reference to another location within the document
- A URL, for example http://developer.arm.com

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`. In both cases, the prefix and the associated value are written in a monospace font, for example `0xFFFF0000`. To improve readability, long numbers can be written with an underscore separator between every four characters, for example `0xFFFF_0000_0000_0000`. Ignore any underscores when interpreting the value of a number.

Pseudocode descriptions

This book uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in a monospace font. The pseudocode language is described in the Arm Architecture Reference Manual.

Assembler syntax descriptions

This book contains numerous syntax descriptions for assembler instructions and for components of assembler instructions. These are shown in a `monospace` font.

References

This section lists publications by Arm and by third parties.

See Arm Developer (http://developer.arm.com) for access to Arm documentation.

[1] *ARM DEN 0070 Firmware interfaces for mitigating cache speculation vulnerabilities*. Arm.

[2] *ARM DDI 0487 Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile*. Arm.

[3] *ARM DDI 0406 Arm® Architecture Reference Manual Armv7-A and Armv7-R edition*. Arm.

[4] *ARM DEN 0091 SVE impact on Secure firmware*. Arm.

[5] *ARM IHI 0055 Procedure Call Standard for the Arm 64-bit Architecture*. Arm.

[6] *RFC 4122 - A Universally Unique IDentifier (UUID) URN Namespace*. See http://tools.ietf.org/html/rfc4122

[7] *ARM DEN 0022 Power State Coordination Interface*. Arm.

[8] *Arm Security Update*. See https://developer.arm.com/support/security-update

[9] *Cache Speculation Side-channels*. See https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability

[10] *ARM DEN 0054 Software Delegated Exception Interface (SDEI)*. Arm.

[11] *ARM DEN 0044 Arm® Base Boot Requirements*. Arm.

[12] *ARM DEN 0137 Realm Management Monitor specification*. Arm.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

If you have any comments or suggestions for additions and improvements create a ticket at https://support.developer.arm.com. As part of the ticket include:

- The title (SMC Calling Convention).
- The number (DEN0028 1.6 G).
- 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.

Note

Arm tests PDFs only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the appearance or behavior of any document when viewed with any other PDF reader.

Inclusive terminology commitment

Arm values inclusive communities.

Arm recognizes that we and our industry have used terms that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive terms. If you find offensive terms in this document, please contact terms@arm.com.

Terms and abbreviations

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.

EL1 Software The software running at EL1 in a particular Security state. If the Security state implements EL2 [2], then there may be multiple EL1 Software instances managed by the EL2 Software in that Security state. Otherwise, the EL1 Software is managed by the EL3 Monitor.

EL2 Software The software running at EL2 in a particular Security state. There is an EL2 Software instance for every implemented Security state that has an EL2 Exception level. The EL2 Software, in any Security state, is managed by the EL3 Monitor.

EL3 Monitor The EL3 Monitor is software that executes at the EL3 Exception level. The EL3 Monitor receives and handles Secure Monitor exceptions, and provides transitions between distinct security states.

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, in the Non-secure Security state, and supports the execution of multiple 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

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 at a managing EL that handles the SMC or HVC calls, made from a SMCCC Caller, in a manner that is compliant with this document. A SMCCC implementation complies with a particular version of the SMCCC.

Managing EL The firmware in the more privileged EL that is immediate to the SMCCC Caller. Note that in the case of a type 2 Hypervisor, both the Hypervisor and its guest may be in the same EL. In that case, the term Managing EL, of these guests, refers to the type 2 Hypervisor.

SMCCC Caller The entity that invokes a SMC or HVC call.

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.

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, Armv8, and Armv9 architectures.

The SMC instruction is used to generate a synchronous exception that is handled by the EL3 Monitor, or trapped and handled by EL2 Software. Arguments and return values are passed in registers. After being handled by the EL3 Monitor, calls that result from the instructions can be passed to a target software component that may reside in a Security state [2] distinct from the caller.

The HVC instruction is used to generate a synchronous exception that is handled by the EL2 Software.

Arguments and return values are passed in registers. EL2 Software can also trap SMC calls that are made by EL1 Software, which allows the calls to be emulated, passed through, or denied as appropriate.

Note

The HVC instruction is defined in [2], it can be issued by a caller in any Security state and is handled by the EL2 Software running in the same Security state as the caller. Throughout this document the term hypervisor refers to the Non-Secure EL2 Software.

This specification aims to ease integration and reduce fragmentation between software layers running at different Exception levels and Security states, for example operating systems, hypervisors, Trusted OSs, EL3 Monitors, and system firmware in general.

The calling mechanism defined in this document applies to callers in all Arm A-profile Security states [2] managed by EL3 (i.e. Non-secure, Secure, Realm).

Note

This document is defined for the Armv8-A and Armv9-A Exception levels, EL0 to EL3 [2]. 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, control is synchronously transferred between a caller in one Security state [2] and a callee, in a potentially distinct Security state, through Secure Monitor Call (SMC) exceptions [3][2]. SMC exceptions are generated by the SMC instruction [3][2], and are handled by the EL3 Monitor, or trapped and handled by EL2 Software. The operation of the callee is determined by the parameters that are passed in through registers.

2.2 Hypervisor Calls

Hypervisor Calls (HVCs) that are made by EL1 Software, in any Security state, result in a synchronous transfer of control to EL2 Software, and are regarded as HVC exceptions. The operation of the callee 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 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</td><td>0x07000000</td><td>Vendor Specific EL3 Monitor Calls</td></tr>
<tr><td>8-47</td><td>0x08000000 - 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>23:17</td><td>0x00FE0000</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>16</td><td>0x00010000</td><td>From SMCCCv1.3:
Hint bit denoting the absence of SVE specific live state.
If set to 1, the caller asserts that the registers P0-P15, FFR and the bits with index greater than 127 in the Z0-Z31 registers do not contain any live state.
This bit is not part of the function identification. The SMCCC implementation must disregard this bit and consider it to be 0 for the purpose of function identification.
Before SMCCCv1.3:
Must be zero (MBZ)</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[1], see Table **6-2**.

2.5.3 Conduits

The mechanism or instruction that is used to perform a call is referred to as the conduit. The conduit can be either an SMC or an HVC.

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

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

[1] Arm recognizes that some TOS vendors use calls with FID in the 0x2000_0000—0x7FFF_FFFF range, calls in this range should not be blocked by FW unless strictly needed.

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.

Note

The SMCCCv1.0 interface defines the return state of the X4—X17 registers to be unpredictable. If the SMCCC version is 1.0, a caller must accommodate an unpredictable return on X4—X17.

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 Equivalence of SMC32/HVC32 and SMC64/HVC64 function definitions

W0[30] in the **Function Identifier** specifies whether a call is SMC32/HVC32 or SMC64/HVC64 (see Table **2-1**). In the case where 2 **Function Identifiers** only differ in W0[30], Arm strongly recommends that the SMC32/HVC32 and SMC64/HVC64 definitions of the function are semantically equivalent.”

These two definitions should:

- use the same set of registers for arguments and return values.
- have the same bit definitions for each argument and return value.

For example, a 32-bit bit-field argument located in W1[31:0] for the SMC32/HVC32 function definition, should also be a 32-bit bit-field in X1[31:0] for the SMC64/HVC64 function definition, and bits X1[63:32] should be 0.

The two definitions may differ in trivial ways, for example:

- a 32-bit integer argument/return in the SMC32/HVC32 definition may become a sign-extended 64-bit integer argument/return in the SMC64/HVC64 function definition.
- a 32-bit address argument/return in the SMC32/HVC32 definition may become a 64-bit address argument/return in the SMC64/HVC64 function definition.

2.9 SME, SVE, SIMD and floating-point registers

SME, 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.

If the calling context does not have live state in any SVE registers (P0-P15, FFR and the bits with index greater than 127 in Z0-Z31), the caller can set the FID[16] bit, see Table **2-1**.

The SMCCC implementation must ensure that the live state, belonging to the calling context, on all SME (ZA array) [2], SVE (Z0-Z31, P0-P15, FFR) [2], Advanced SIMD and floating-point registers (V0-V31, FPCR, FPSR), is preserved over all SMC and HVC calls.

The live state to be preserved, by the SMCCC implementation, depends on the architectural implemented features (FEAT_SVE, FEAT_SVE2, FEAT_SME, FEAT_SME_FA64) and caller provided input (PSTATE.SM, PSTATE.ZA, FID[16]). A detailed list of registers to be preserved is provided in Appendix C .

If SME is implemented and the calling context does not have live state in any SME registers (ZA[0]-ZA[31]), the caller should set PSTATE.ZA to 0 before invoking an SMC or HVC call.

Additionally, if FID[16] is 1, the SMCCC implementation must either preserve, or set to zero, the P0-P15 and FFR registers, and the bits with index greater than 127 in the Z0-Z31 registers if these are architecturally accessible from the context of the caller.

The SMCCC implementation is responsible for ensuring that information is not disclosed between execution contexts through SME, 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 [4].

2.10 SMC and HVC immediate value

The SMC and HVC instructions encode an immediate value, as defined by the Arm architecture [3][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 EL3 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.

2.11 Client ID (optional)

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

2.11.1 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 EL3 Monitor.

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

2.11.2 HVC calls

The Client ID is ignored by the HVC calling convention.

2.12 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**.

2.13 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 EL3 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.

For more information, see [5]

[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. **Note:** on SMCCCv1.0 compliant implementations these are scratch registers.

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		
X30		The Link Register		
X29		The Frame Pointer		
X19...X28		Registers that are saved by the called function		
X18		The Platform Register		
X17		Parameter register. The second intra-procedure-call scratch register.	Dependent[2]	Registers values are preserved or contain call results.
X16		Parameter register. The first intra-procedure-call scratch register.		
X9...X15		Parameter registers. Temporary registers.		
X8		Parameter register. Indirect result location register.		
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]		
W6	X6 (or W6)	Parameter register. Optional Session ID register		
W4...W5	X4...X5	Parameter registers.		
W1...W3	X1...X3	Parameter registers.	Yes	SMC and HVC result registers.
W0	W0	Function Identifier.		

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 Name	Role during SMC or HVC call		
SMC32/HVC32	**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		
R12	The Intra-Procedure-call scratch register.		
R11	Variable-register 8		
R10	Variable-register 7		
R9	Platform Register		
R8	Variable-register 5		
R7	Parameter register 7 Optional Client ID in bits[15:0] (ignored for HVC calls) Optional Secure OS ID in bits[31:16]	Dependent[3]	Register values are preserved or contain call results.
R6	Parameter register 6 Optional Session ID		
R5	Parameter register 5		
R4	Parameter register 4		
R3	Parameter register 3	Yes	SMC and HVC results registers
R2	Parameter register 2		
R1	Parameter register 1		
R0	Function Identifier		

[3] 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 .

5.3 Unique Identification format

This value identifies the implementer of a subrange (see Section **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**.

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

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

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 Section **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.
Vendor Specific EL3 Monitor Service	7	Calls defined by the vendor of the EL3 Monitor. It is the responsibility of the vendor of the EL3 Monitor to maintain this space and ensure FID uniqueness within its codebase. In cases where the EL3 Monitor software is derived from a common codebase, for example the TF-A project, that project may choose to standardize some services in this range. It is strongly recommended that the services in this range are only called from software that is tightly integrated with the SMCCC implementation. General purpose EL1 or EL2 software should not call services in this range.
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 [7].

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[4]
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
0x87000000-0x8700FFFF	SMC32: Vendor Specific EL3 Monitor Service Calls
0x88000000-0xAF00FFFF	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
0xC7000000-0xC700FFFF	SMC64: Vendor Specific EL3 Monitor Service Calls
0xC8000000-0xEF00FFFF	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**[5] – 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[5]	0x8000_FF01[5]	0x8000_FF03[5]
CPU Service	0x8100_FF00[5]	0x8100_FF01	0x8100_FF03
SiP Service	0x8200_FF00[5]	0x8200_FF01	0x8200_FF03
OEM Service	0x8300_FF00[5]	0x8300_FF01	0x8300_FF03
Standard Secure Service	0x8400_FF00[5]	0x8400_FF01	0x8400_FF03
Standard Hypervisor Service	0x8500_FF00[5]	0x8500_FF01	0x8500_FF03
Vendor Specific Hypervisor Service	0x8600_FF00[5]	0x8600_FF01	0x8600_FF03
Vendor Specific EL3 Monitor Service	-	0x8700_FF01	0x8700_FF03
Trusted Applications[6]	-	-	-
Trusted OS	0xBF00_FF005	0xBF00_FF01	0xBF00_FF03

In addition to the values in the table above, the other Function Identifiers in the 0xXXXX_FF00 – 0xXXXX_FFFF range, for example 0x8000_FF02 and 0x8000_FF04-0x8000_FFFF 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:

Note

[4]Arm recognizes that some TOS vendors use calls with FID in the 0x2000_0000—0x7FFF_FFFF range, calls in this range should not be blocked by FW unless strictly needed.
[5]Query deprecated from SMCCC v1.2 onward
[6]It is the responsibility of a Trusted OS to identify and describe the services that are provided by Trusted applications.

Table 6-4: Reserved Standard Secure Service Call range

Function Identifier	Reserved for	Notes
0x8400_0000-0x8400_001F	PSCI 32-bit calls	32-bit Power Secure Control Interface.
0xC400_0000-0xC400_001F	PSCI 64-bit calls	64-bit Power Secure Control Interface.
0x8400_0020-0x8400_003F	SDEI 32-bit calls	32-bit Software Delegated Exception Interface.
0xC400_0020-0xC400_003F	SDEI 64-bit calls	64-bit Software Delegated Exception Interface.
0x8400_0040-0x8400_004F	MM 32-bit calls	32-bit Management Mode.
0xC400_0040-0xC400_004F	MM 64-bit calls	64-bit Management Mode.
0x8400_0050-0x8400_005F	TRNG 32-bit calls	32-bit TRNG FW Interface.
0xC400_0050-0xC400_005F	TRNG 64-bit calls	64-bit TRNG FW Interface.
0x8400_0060-0x8400_00EF	FF-A 32-bit calls	32-bit PSA Firmware Framework A.
0xC400_0060-0xC400_00EF	FF-A 64-bit calls	64-bit PSA Firmware Framework A.
0x8400_00F0-0x8400_010F	Errata FW 32-bit calls	32-bit Errata FW interface.
0xC400_00F0-0xC400_010F	Errata FW 64-bit calls	64-bit Errata FW interface.
0x8400_0150-0x8400_02CF	CCA 32-bit calls	32-bit Arm CCA function ID range reservation.
0xC400_0150-0xC400_02CF	CCA 64-bit calls	64-bit Arm CCA function ID range reservation.

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.

6.4 Implemented Standard Hypervisor Service Calls

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

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

Table 6-5: Reserved Standard Hypervisor Service Call range

Function Identifier	Reserved for	Notes
0xC500_0020-0xC500_003F	PV Time 64-bit calls	64-bit Paravirtualized Time for Arm-based Systems.
0xC500_0040-0xC500_013F	RHI 64-bit calls	64-bit Realm Host Interface.

7 Arm Architecture Calls

7.1 Return codes

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	0x8000_0000	
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 [7].
- Information from firmware tables like the Flattened Device Tree table or ACPI tables.

See Appendix B 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 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 **F0-1** in Appendix F .

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	0x8000_0001	
Return	uint32 arch_func_id	Function ID of an Arm Architecture Service Function	
	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 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.
- The Standard Hypervisor Service range: 0x8500_0000-0x8500_FFFF and 0xC500_0000-0xC500_FFFF.

Any arch_func_id values outside of these ranges 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 FID	0x8000_0002	
	uint32 SoC_ID_type	0: SoC version 1: SoC revision 2: SoC name (optionally implemented for SMC64 calls, not implemented for SMC32 calls) 3 – 0xFFFF FFFF: Reserved	
Return	W0/R0 (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.])[7] - Bits [23:16] JEP-106 identification code with parity bit for the SiP (see [9.])[7] - Bits [15:0] Implementation defined SoC ID SoC_ID_type == 1 - Bit[31] must be zero - Bits [30:0] SoC revision

	X1 – X17		Unused by SoC_ID_type = {0, 1}. SoC_ID_type == 2 (SoC name), defined for SMC64 only. SoC name is encoded as an UTF-8 string, terminated with a null byte, with no byte order mark (BOM). The string occupies at most 136 bytes, including the null byte. The bytes from the end of the string until byte 135 (inclusive) must be set to null. For compatibility with older parsers, US-ASCII characted should be used. The mapping between bytes and return registers is the following: - X<a>[7:0]: byte $(a-1).8$ - X<a>[15:8]: byte $(a-1).8+1$ - X<a>[23:16]: byte $(a-1).8+2$ - X<a>[31:24]: byte $(a-1).8+3$ - X<a>[39:32]: byte $(a-1).8+4$ - X<a>[47:40]: byte $(a-1).8+5$ - X<a>[55:48]: byte $(a-1).8+6$ - X<a>[63:56]: byte $(a-1).8+7$ for <a> in 1, ..., 17. For example, X1[7:0] corresponds to byte0, X1[15:8] corresponds to byte 2, and X2[7:0] corresponds to byte 8.

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 Section **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.

[7] 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

- 2: SoC name: Function returns the SiP-defined SoC name represented as a null-terminated sequence of UTF-8 characters. The SoC name implementation is optional and is only present for SMC64 invocations.

Any other SoC_ID_type parameter value is invalid.

7.4.4 Return

If the call returns NOT_SUPPORTED the function is not present in the SMCCC implementation. If the call returns INVALID_PARAMETER, the SoC_ID_type parameter is either outside of the {0,1, 2} set, or SoC_ID_type==2 is not implemented. On success, the return value is present in the following registers:

- W0, representing the SoC version, for SoC_ID_type == 0
- W0, representing the SoC revision, for SoC_ID_type == 1
- X1–X17, representing the SoC name, for SoC_ID_type == 2

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** .

The caller can infer that the INVALID_PARAMETER return status, to a call with SoC_ID_type == 2, means SoC name is not implemented.

7.4.6 Implementation responsibilities

If implemented, the firmware:

- must provide discovery of this function as defined in the Section **7.4.2** .
- must Implement SoC_ID_type == {0, 1},
- must not implement SoC_ID_type == 2 for SMC32.
- can optionally implement SoC_ID_type == 2 for SMC64 (Function ID 0xC000_0002),
- must return INVALID_PARAMETER if SoC_ID_type == 2 is not implemented.
- must ensure that SoC version and revision uniquely identify the SoC,
  - SoC name must not contain SoC identifying information not captured by <SoC version, SoC revision>.

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. This result does not imply that the calling PE is unaffected by 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 [8] and [9].

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 [8] and [9].

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. This result does not imply that the calling PE is unaffected by CVE-2018-3639.

7.6.3 Caller responsibilities

The caller must not call this function unless it has determined that this function is implemented in the firmware, Section **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 [8] and [9].

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 [7]
- To a PE when it wakes up from a powerdown state (for example, following a CPU_SUSPEND call), as defined by the PSCI specification [7]

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 [10], 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 [8] and [9].

7.7 SMCCC_ARCH_WORKAROUND_3

Dependency	OPTIONAL from SMCCC v1.1.	
Description	Execute the mitigation for CVE-2017-5715 and CVE-2022-23960 on the calling PE	
Parameters	uint32 Function ID	0x8000_3FFF
Return	void	This function has no return value.

7.7.1 Usage

This call is used by system software to execute a firmware workaround that is required to mitigate CVE-2017-5715 and CVE-2022-23960.

7.7.2 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 3FFF. The result of that call should be interpreted as:

7.7.3 Caller responsibilities

The caller must not call this function unless it has determined that it is implemented in the firmware, see Section **7.7.2** . Arm recommends that the caller only call this function on PEs that are affected by CVE-2017-5715 or CVE-2022-23960 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 [8] and [9].

Note

CVE-2022-23960 extends CVE-2017-5715 – it is natural for a workaround that mitigates CVE-2022-23960 to also mitigate CVE-2017-5715. Any SMCCC_ARCH_WORKAROUND_3 implementation mitigates both.

7.7.4 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 **7.7.2** . Arm

NOT_SUPPORTED	The discovery call will return NOT_SUPPORTED on every PE in the system. SMCCC_ARCH_WORKAROUND_3 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 or CVE-2022-23960. • The system contains at least 1 PE affected by CVE-2017-5715 or CVE-2022-23960 that has no firmware mitigation available. • The firmware does not provide any information about whether firmware mitigation is required.
0	SMCCC_ARCH_WORKAROUND_3 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 or CVE-2022-23960.
1	SMCCC_ARCH_WORKAROUND_3 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 or CVE-2022-23960. This result does not imply that the PE is unaffected by CVE-2017-5715 or CVE-2022-23960.

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 or CVE-2022-23960. If the PE is affected by both CVE-2017-5715 and CVE-2022-23960, the SMCCC_ARCH_WORKAROUND_3 implementation must mitigate both. 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 [8] and [9].

Note

If SMCCC_ARCH_WORKAROUND_3 is implemented then SMCCC_ARCH_WORKAROUND_1 is superfluous. The SMCCC_ARCH_WORKAROUND_1 should nevertheless be reported as implemented. This is to support legacy clients that are unaware of SMCCC_ARCH_WORKAROUND_3.

7.8 SMCCC_ARCH_FEATURE_AVAILABILITY

The PEs implement a set of architectural features defined by the Arm Architecture [2]. Some architectural features define controls to enable aspects of that feature during system operation (via fields in System Registers). Firmware must have awareness of all architectural features present in the SoC it manages. In the ideal case, firmware enables all features and handles all feature related events.

On platforms that deviate from the ideal case, software running at lower ELs must identify features implemented by the PE, but which firmware has not enabled.

This interface allows firmware to declare the features that it makes available for callers to use. The interface reports the features that firmware is aware of and has catered for suitable usage by callers. Callers to this function can use this information to avoid using a feature which may lead to an unhandled trap at EL3.

Note

The existence of any feature, not requiring EL3 firmware support, is fully described by the corresponding field in the ID register.

Note

Some features (e.g. FEAT_FGT and FEAT_FGT2), when present in the PE, must be enabled and fully supported for correct and safe system operation. The exhaustive list of architectural features that firmware must enable and support is out of scope of this document. Further information can be found in the Base Boot Requirements specification [11].

Dependency	OPTIONAL from SMCCC v1.1. NOT SUPPORTED in SMCCC v1.0.	
Description	Discover architectural features enabled by EL3 firmware for use by callers.	
Parameters	uint32 Function ID	0x8000_0003
	uint64 bitmask_selector	The identifier of the feat_bitmask to be returned. See Table **7-3** for a list of bitmask_selectors and the mapping of features to each bit in the feat_bitmask. When there is a related system register, the value of the bitmask selector is the opcode of that system register, as defined by: opcode = (op0 «18) | (op1 « 16) | (CRn « 12) | (CRm « 8) | (op2 « 5).
Return	int32 Status	The status of the call: - SUCCESS - INVALID PARAMETER: the bitmask_selector is unknown. All other values are reserved.
	uint64 feat_bitmask	A bitmask describing the features enabled by EL3 firmware. Each feature maps to a feat_bitmask_selector, bit offset The feature_bitmask_selector determines the meaning of each bit, see Table **7-3** for a list of bitmask_selectors and the meaning of each bit in the feat_bitmask.

7.8.1 Usage

SMCCC_ARCH_FEATURE_AVAILABILITY returns a bitmask of features enabled by EL3 firmware for use by callers. The return of SMCCC_ARCH_FEATURE_AVAILABILITY applies to the PE on which the call was made.

7.8.2 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_0003. The result of that call should be interpreted as:

7.8.3 Caller responsibilities

The caller must determine the function is present before calling it (see 7.8.2). The return applies to the calling PE, the caller must invoke SMCCC_ARCH_FEATURE_AVAILABILITY on each PE. If the call returns an INVALID_PARAMETER status, the caller should assume that any architectural features for that bitmask_selector are not enabled by firmware.

Table 7-3: Bitmask_selector and feature bit offset assignment

Bitmask_selector	Feature offsets in feat_bitmask
0x1e_1100 (SCR_EL3 opcode)	The register support obtained for this bitmask_selector is directly related to the trap controls existing in SCR_EL3. If feat_bitmask[x]==1 this means that the system registers enumerated in SCR_EL3[x] can be accessed, from EL2 and EL1, and will not lead to an unhandled exception at EL3. If feat_bitmask[x]==0, the EL3 firmware may be unaware of the system registers enumerated by SCR_EL3[x]. Interactions with these system registers may have unintended consequences. The list of feat_bitmask values is present in Appendix E (see Table **E0-1**).
0x1e_1140 (CPTR_EL3 opcode)	The register support obtained for this bitmask_selector is directly related to the trap controls existing in CPTR_EL3. If feat_bitmask[x]==1 this means that the system registers enumerated in CPTR_EL3[x] can be accessed, from EL2 and EL1, and will not lead to an unhandled exception at EL3. If feat_bitmask[x]==0, the EL3 firmware may be unaware of the system registers enumerated by CPTR_EL3 [x]. Interactions with these system registers may have unintended consequences. The list of feat_bitmask values is present in Appendix E (see Table **E0-1**).
0x1e_1320 (MDCR_EL3 opcode)	The register support obtained for this bitmask_selector is directly related to the trap controls existing in MDCR_EL3. If feat_bitmask[x]==1 this means that the system registers enumerated in MDCR_EL3[x] are can be accessed, from EL2 and EL1, and will not lead to an unhandled exception at EL3. If feat_bitmask[x]==0, the EL3 firmware may be unaware of the system registers enumerated by MDCR_EL3 [x]. Interactions with these system registers may have unintended consequences. The list of feat_bitmask values is present in Appendix E (see Table **E0-1**).
0x1e_a500 (MPAM3_EL3 opcode)	The register support obtained for this bitmask_selector is directly related to the trap controls existing in MPAM3_EL3. If feat_bitmask[x]==1 this means that the MPAM system registers associated to MPAM3_EL3[x] can be accessed, from EL2 and EL1, and will not lead to an unhandled exception at EL3. If feat_bitmask[x]==0, the EL3 firmware may be unaware of the system registers enumerated by MPAM_EL3 [x]. Interactions with these system registers may have unintended consequences. The list of feat_bitmask values is present in Appendix E (see Table **E0-1**).
All other values are reserved	–

NOT_SUPPORTED	The discovery call will return NOT_SUPPORTED on every PE in the system. SMCCC_ARCH_FEATURE_AVAILABILITY must not be invoked on any PE in the system.
0	SMCCC_ARCH_FEATURE_AVAILABILITY is present and can be safely invoked on any PE in the system.

7.8.4 Implementation responsibilities

This function can only be provided by SMCCC implementations that are 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. When present, the function must be implemented for all PEs. Firmware should enable and fully support any architectural features that it is aware of, for callers. For every feature that firmware reports feat_bitmask[x] == 1, firmware must implement a behaviour that complies with the bit description from Table **7-3**. The value of feat_bitmask[x] must be constant from system initialization until reset/power off.

7.9 SMCCC_ARCH_WORKAROUND_4

Dependency	OPTIONAL from SMCCC v1.1 Not supported in SMCCC v1.0 or earlier	
Description	Signals the presence of CVE-2024-7881 mitigation on the calling PE.	
Parameters	uint32 Function ID	0x8000_0004
Return	void	This function has no return value.

7.9.1 Usage

The presence of this call is used to determine that CVE-2024-7881 is mitigated by a higher EL. SMCCC_ARCH_WORKAROUND_4 is a NOP. The function SMCCC_ARCH_WORKAROUND_4 should never be called. The information about CVE-2024-7881 mitigation by a higher EL is specified by the SMCCC_ARCH_WORKAROUND_4 presence alone.

7.9.2 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_0004. The result of that call should be interpreted as:

NOT_SUPPORTED	The SMCCC_ARCH_WORKAROUND_4 is not implemented on the calling CPU, and CVE-2024-7881 is not mitigated by a higher EL. The firmware does not provide any information about whether firmware mitigation is required.
0	SMCCC_ARCH_WORKAROUND_4 is implemented on the calling PE, and CVE-2024-7881 is mitigated by a higher EL.

7.9.3 Caller responsibilities

The caller infers that a higher EL mitigates CVE-2024-7881 by detecting the presence of SMCCC_ARCH_WORKAROUND_ see Section **7.3** .

Note

The SMCCC_ARCH_WORKAROUND_4 call is a NOP. Its invocation does not change the state of CVE-2024-7881 mitigation by a higher EL. An OS should never invoke this call.

7.9.4 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 **7.3** . The

call implementation must not impact the CVE-2024-7881 mitigation. If implemented, the firmware must mitigate CVE-2024-7881

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);
}
```

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.

Appendix C: SME, SVE, SIMD and FP live state preservation by the SMCCC implementation

1 SVE state

SVE is implemented if ID_AA64PFR0_EL1.SVE==1 [2].

SVE introduces the following registers:

- Z0-Z31, P0-P15, FFR

2 SME state

SME is implemented if ID_AA64PFR1_EL1.SME==1 [2].

SME introduces the following registers:

- ZA array

If SME is implemented, the processor state (PSTATE) is augmented with the ZA and SM bits. The ZA array is only architecturally visible when PSTATE.ZA==1.

Note

The PE sets the values in the ZA array to zero on a PSTATE.ZA transition from 0 to 1 [.15].

2.1 Streaming SVE processor state

SME introduces the streaming SVE processor state, which is enabled when PSTATE.SM==1. The streaming SVE mode is required on any SME implementation even if SVE is not implemented [2].

When the processor is in streaming SVE mode the Z0-Z31 and P0-P15 registers are architecturally visible. When implementing SME, a processor can implement a full A64 mode. The presence of full A64 mode is indicated by ID_AA64SMFR0_EL1.FA64==1. The full A64 mode is enabled if SCMR_ELx.FA64==1 for the current and higher Els. The IsFullA64Enabled() pseudocode, listed in [2], is used to determine if the full A64 mode is enabled. When enabled, the full A64 mode makes the FFR SVE register architecturally visible in streaming SVE mode, if SVE is implemented [2].

Note

If the full A64 mode is not implemented, the FFR register is not architecturally accessible when PSTATE.SM==1.

Note

If PSTATE.SM==1 then the length of the SVE Z0-Z31, P0-P15 registers is specified by SMCR_ELx.LEN [2].

Note

Any PSTATE.SM transition resets all the Z0-Z31, P0-P15, FFR and FPSR registers to an architecturally defined constant, see [.15] for more information.

3 Register list to be preserved

The procedures to create the set of registers to preserve are listed in the pseudocode below. A single pseudocode listing is applicable to a platform. The applicability of a pseudocode listing to a platform depends on whether that platform implements SVE and/or SME. The Preserve function, used below, takes a variable length list of registers. The registers passed as arguments to the function Preserve are appended to the set of registers to be preserved across the next SMC/HVC call.

- If neither SVE nor SME are implemented

```
Preserve(V0-V31, FPSR, FPCR)
```

- If SVE is implemented and SME is not implemented

```
Preserve(FPSR, FPCR)

If FID[16]==1
    Preserve(V0-V31)

Else
    Preserve(Z0-Z31, P0-P15, FFR)
```

- If SVE is **not** implemented and SME is implemented

```
Preserve(FPSR, FPCR)

If FID[16]==1 or PSTATE.SM==0
    Preserve(V0-V31)

Else
    Preserve(Z0-Z31, P0-P15)

If PSTATE.ZA==1
    Preserve(ZA array)
```

- If both SVE and SME are implemented

```
Preserve (FPSR, FPCR)

If FID[16]==1
    Preserve(V0-V31)

Else
    Preserve(Z0-Z31, P0-P15)
```

3. Register list to be preserved

```
If FID[16]==0 AND (IsFullA64Enabled() OR PSTATE.SM==0)
    Preserve(FFR)

If PSTATE.ZA==1
    Preserve(ZA array)
```

Note

The V0-V31 registers are contained in the Z0-Z31 registers. The registers V0-V31 are implicitly preserved when the Z0-Z31 registers are preserved.

Appendix D: SMCCC deployment model

A platform can contain multiple distinct SMCCC implementations.

In the general case, a SMCCC Caller interacts only with the SMCCC implementation provided by its managing EL, see Figure 1. The managing EL informs each SMCCC Caller of the conduit that it must use, via an IMPLEMENTATION DEFINED discovery method. Examples of conduit discovery methods are the ACPI FADT or suitable FDT binding.

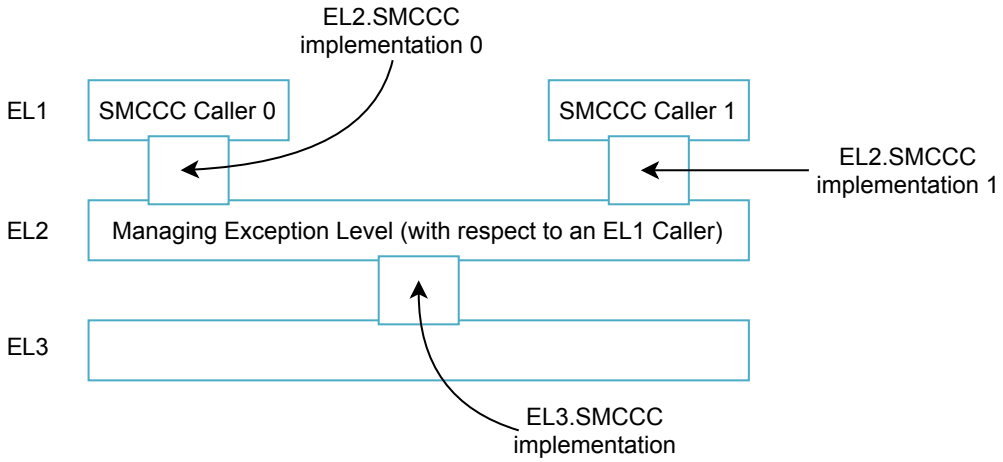

Figure 1: SMCCC Caller relationship to its managing EL

Arm recommends that firmware at EL2 always traps SMC calls.

A hypervisor may allow some callers to use both the SMC and HVC conduits. The meaning attributed to each conduit is defined by that hypervisor.

1 SMCCC implementations and security states

The software/firmware executing in any suitable EL (EL3 or EL2 in any Security state), can expose SMCCC implementations to a caller at a lower EL.

The SMCCC FID space partitioning (Table **2-1**, Table **6-3**) applies to all Security states managed by EL3. EL2 Software may restrict the services/functions available to the callers under its management. For example, the RMM specification restricts the SMC services available to Realm EL1 Software, see [12] for the list of allowed services.

Appendix E: Feature Discovery bitmask

The SMCCC_ARCH_FEATURE_AVAILABILITY (Section **7.8**) allows a caller to determine the system registers that can be safely accessed.

The system register that can be safely accessed are described in bitfields, returned by the function SMCCC_ARCH_FEATURE_AVAILABILITY.

The complete definition of the bitfields covering the relevant system registers defined in the Arm ARM [2], issue J.a, are listed below. Additional features corresponding to more recent architecture versions may be discovered using this version of the SMCCC.

Bitmask_selector	Feature offsets in feat_bitmask
0x1e_1100 (SCR_EL3 opcode)	feat_bitmask [61] == 1: Accesses to registers listed in SCR_EL3.HACDBSE are emulated by firmware or access directly the physical register. feat_bitmask [60] == 1: Accesses to registers listed in SCR_EL3.HDBSSE are emulated by firmware or access directly the physical register. feat_bitmask [59] == 1: FEAT_FGT2 is enabled, any EL2 access to any of the EL2 system registers, enumerated in the SCR_EL3.FGTEn2 definition, are emulated by firmware or access directly the physical register. feat_bitmask [53] == 1: Accesses to registers listed in SCR_EL3.PFAREn are emulated by firmware or access directly the physical register. feat_bitmask [52] == 1: Accesses to registers listed in SCR_EL3.TWERR are emulated by firmware or access directly the physical register. feat_bitmask [50] == 1: Accesses to the FPMR register are emulated by firmware or access directly the physical register. feat_bitmask [47] == 1: Accesses to registers listed in SCR_EL3.D128 are emulated by firmware or access directly the physical register. feat_bitmask [46] == 1: Accesses to registers listed in SCR_EL3.AIEn are emulated by firmware or access directly the physical register. feat_bitmask [45] == 1: Accesses to registers listed in SCR_EL3.PIEn are emulated by firmware or access directly the physical register. feat_bitmask [44] == 1: Accesses to registers listed in SCR_EL3.SCTLR2En are emulated by firmware or access directly the physical register. feat_bitmask [43] == 1: Accesses to registers listed in SCR_EL3.TCR2En are emulated by firmware or access directly the physical register.

0x1e_1100 (SCR_EL3 opcode)	feat_bitmask [42] == 1: Accesses to registers listed in SCR_EL3.RCWMASKEn are emulated by firmware or access directly the physical register. feat_bitmask [41] == 1: Accesses to register TPIDR2_EL0 are emulated by firmware or access directly the physical register. feat_bitmask [40] == 1: Accesses to registers listed in SCR_EL3.TRNDR are emulated by firmware or access directly the physical register. feat_bitmask [39] == 1: Accesses to registers listed in SCR_EL3.GCSEn are emulated by firmware or access directly the physical register. feat_bitmask [38] == 1: Accesses to register HCRX_EL2 are emulated by firmware or access directly the physical register. feat_bitmask [37] == 1: Accesses to register ACCDATA_EL1 are emulated by firmware or access directly the physical register. feat_bitmask [36] == 1: Accesses to registers listed in SCR_EL3.AMVOFFEN are emulated by firmware or access directly the physical register. feat_bitmask [28] == 1: Accesses to register CNTPOFF_EL2 are emulated by firmware or access directly the physical register. feat_bitmask [27] == 1: FEAT_FGT is enabled, any EL2 access to the EL2 system registers, enumerated in the SCR_EL3.FGTEn definition, are emulated by firmware or access directly the physical register. feat_bitmask [26] == 1: Accesses to registers listed in SCR_EL3.ATA are emulated by firmware or access directly the physical register. feat_bitmask [25] == 1: Accesses to registers listed in SCR_EL3.EnSCXT are emulated by firmware or access directly the physical register. feat_bitmask [21] == 1: Accesses to registers listed in SCR_EL3.FIEN are emulated by firmware or access directly the physical register. feat_bitmask [16] == 1: Accesses to registers listed in SCR_EL3.APK are emulated by firmware or access directly the physical register. feat_bitmask [15] == 1: Accesses to registers listed in SCR_EL3.TERR are emulated by firmware or access directly the physical register. feat_bitmask [14] == 1: Accesses to registers listed in SCR_EL3.TLOR are emulated by firmware or access directly the physical register. All other bits in feat_bitmask are reserved and must be 0.

0x1e_1140 (CPTR_EL3 opcode)	feat_bitmask [31] == 1: Accesses to registers listed in CPTR_EL3.TCPAC are emulated by firmware or access directly the physical register. feat_bitmask [30] == 1: Accesses to registers listed in CPTR_EL3.TAM are emulated by firmware or access directly the physical register. feat_bitmask [20] == 1: Accesses to registers listed in CPTR_EL3.TTA are emulated by firmware or access directly the physical register. feat_bitmask [12] == 1: Accesses to registers listed in CPTR_EL3.ESM are emulated by firmware or access directly the physical register. feat_bitmask [10] == 1: Accesses to registers listed in CPTR_EL3.TFP are emulated by firmware or access directly the physical register. feat_bitmask [8] == 1: Accesses to registers listed in CPTR_EL3.EZ are emulated by firmware or access directly the physical register. All other bits in feat_bitmask are reserved and must be 0.
0x1e_1320 (MDCR_EL3 opcode)	feat_bitmask [50] == 1: Accesses to register MDSTEPOP_EL1 are emulated by firmware or access directly the physical register. feat_bitmask [47] == 1: Accesses to registers listed in MDCR_EL3.EnITE are emulated by firmware or access directly the physical register. feat_bitmask [44] == 1: Accesses to registers listed in MDCR_EL3.EnPMSS are emulated by firmware or access directly the physical register. feat_bitmask [42] == 1: Accesses to register PMSDFR_EL1 are emulated by firmware or access directly the physical register. feat_bitmask [39] == 1: Accesses to register TRBMPAM_EL1 are emulated by firmware or access directly the physical register. feat_bitmask [36] == 1: Accesses to register PMSNEVFR_EL1 are emulated by firmware or access directly the physical register. feat_bitmask [32] == 1: Accesses to registers listed in MDCR_EL3.SBRBE are emulated by firmware or access directly the physical register. feat_bitmask [27] == 1: Accesses to registers listed in MDCR_EL3.TDCC are emulated by firmware or access directly the physical register. feat_bitmask [24] == 1: Accesses to registers listed in MDCR_EL3.NSTB are emulated by firmware or access directly the physical register. feat_bitmask [19] == 1: Accesses to registers listed in MDCR_EL3.TTRF are emulated by firmware or access directly the physical register.

0x1e_1320 (MDCR_EL3 opcode)	feat_bitmask [12] == 1: Accesses to registers listed in MDCR_EL3.NSPB are emulated by firmware or access directly the physical register. feat_bitmask [10] == 1: Accesses to registers listed in MDCR_EL3.TDOSA are emulated by firmware or access directly the physical register. feat_bitmask [9] == 1: Accesses to registers listed in MDCR_EL3.TDA are emulated by firmware or access directly the physical register. feat_bitmask [7] == 1: Accesses to registers listed in MDCR_EL3.EnPM2 are emulated by firmware or access directly the physical register. feat_bitmask [6] == 1: Accesses to registers listed in MDCR_EL3.TPM are emulated by firmware or access directly the physical register. All other bits in feat_bitmask are reserved and must be 0.
0x1e_a500 (MPAM3_EL3 opcode)	feat_bitmask [62] == 1: Accesses to the following registers: • MPAM0_EL1 • MPAM1_EL1 • MPAM2_EL1 • MPAM3_EL1 • MPAM1_EL12 • MPAMCR_EL2 • MPAMVPMV_EL2 • MPAMVMP0_EL2 • MPAMVMP1_EL2 • MPAMVMP2_EL2 • MPAMVMP3_EL2 • MPAMVMP4_EL2 • MPAMVMP5_EL2 • MPAMVMP6_EL2 • MPAMVMP7_EL2 • MPAMIDR_EL1 • MPAMSM_EL1 are emulated by firmware or access directly the physical register. All other bits in feat_bitmask are reserved and must be 0.
All other values are reserved	–

Table E0-1: Detailed Feature Discovery Arguments

Appendix F: Changelog

Table Table **F0-1** 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 F0-1: Changelog

SMCCC version	SMCCC_VERSION call return	Changes
1.0	-1 or 0x10000	Introduces: • General Service Queries (Section **6.2**)
1.1	0x10001	Result register set: • Mandates preservation of registers X4–X17 across a SMC or HVC call. 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**) • Requirement for the SMCCC implementation to preserve SVE live state across an SMC/HVC. Deprecates: • UID, Revision Queries on Arm Architecture Service (Section **6.2**) • Count Query on all services (Section **6.2**)
1.3	0x10003	Introduces: • SVE absence of live state hint bit (Section **2.5**)
1.4	0x10004	Introduces: • Requirement for the SMCCC implementation to preserve SME live state across an SMC/HVC. • SMCCC_ARCH_WORKAROUND_3 (Section **7.7**)
1.5	0x10005	Introduces: • Vendor Specific EL3 Monitor Service range. • SMCCC_ARCH_FEATURE_AVAILABILITY (Section **7.8**).
1.6	0x10006	Introduces: • SoC_ID_type == 2 to 7.4 SMCCC_ARCH_SOC_ID (Section **7.4**). • SMCCC_ARCH_WORKAROUND_4 (Section **7.9**)