SMC CALLING CONVENTION
System Software on ARM® Platforms

Document number: ARM DEN 0028B

ARM®

SMC Calling Convention
System Software on ARM

Release information

The Change History table lists the changes that are made to this document.

Table 1-1Change history

Date	Issue	Confidentiality	Change
June 2013	A	Non-Confidential	First release
November 2016	B	Non-Confidential	HVC calling convention. SMC calling convention clarifications and updates.

1 ABOUT THIS DOCUMENT

1.1 Introduction

This document defines a common calling mechanism that is to be used with the ***Secure Monitor Call (SMC)*** and ***Hypervisor Call (HVC)*** instructions in both the ARMv7 and ARMv8 architectures.

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

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

This specification aims to ease integration and reduce fragmentation between software layers, for example Operating Systems, hypervisors, Trusted OSs, Secure Monitors, and System Firmware.

Note: This document is defined for the ARMv8-A Exception levels, EL0 to EL3.
The relationship between these Exception levels and the 32-bit ARMv7 Exception levels is described in [2.]

1.2 References

This document refers to the following documents.

Reference	Doc No	Author	Title
[1.]	ARM DDI 0406	ARM	ARM® Architecture Reference Manual ARMv7-A and ARMv7-R edition
[2.]	ARM DDI 0487	ARM	ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile
[3.]	ARM IHI 0042	ARM	Procedure Call Standard for the ARM 32-bit Architecture
[4.]	ARM IHI 0055	ARM	Procedure Call Standard for the ARM 64-bit Architecture
[5.]	ARM DEN 0022	ARM	Power State Coordination Interface
[6.]	http://tools.ietf.org/html/rfc4122	IETF	RFC 4122 - A Universally Unique IDentifier (UUID) URN Namespace

1.3 Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
AArch32 state	The ARM 32-bit Execution state that uses 32-bit general purpose registers, and a 32-bit program counter (PC), stack pointer (SP), and link register (LR). AArch32 Execution state provides a choice of two instruction sets, A32 and T32, previously called the ARM and Thumb instruction sets.
AArch64 state	The ARM 64-bit Execution state that uses 64-bit general purpose registers, and a 64-bit program counter (PC), stack pointer (SP), and exception link registers (ELR). AArch64 Execution state provides a single instruction set, A64.
EL0	The lowest Exception level. The Exception level that is used to execute user applications, in Non-secure state.
EL1	Privileged Exception level. The Exception level that is used to execute operating systems, in Non-secure state.
EL2	Hypervisor Exception level. The Exception level that is used to execute hypervisor code. EL2 is always in Non-secure state.
EL3	Secure Monitor Exception level. The Exception level that is used to execute Secure Monitor code, which handles the transitions between Non-secure and Secure states. EL3 is always in Secure state.
Function Identifier	A 32-bit integer that identifies which function is being invoked by this 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. It supports the execution of multiple EL1 Operating Systems.
Non-secure state	The ARM Execution state that restricts access to only the Non-secure system resources such as: memory, peripherals, and System registers.
OEM	Original Equipment Manufacturer. In this document, the final device manufacturer.
PE	Processing element. The abstract machine that is defined in the ARM architecture, see [2.]
Rx	Register; A32 native 32-bit register, A64 architectural register
S-EL0	The Secure EL0 Exception level, the Exception level that is used to execute trusted application code in Secure state.
S-EL1	The Secure EL1 Exception level, the Exception level that is used to execute Trusted OS code in Secure state.
Secure Monitor	The Secure Monitor is software that executes at the EL3 Exception level. It receives and handles Secure Monitor exceptions, and provides transitions between Secure state and Non-secure state.
Secure state	The ARM Execution state that enables access to the Secure and Non-secure systems resources, such as: 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.
SMC32/HVC32	32-bit SMC and HVC calling convention.
SMC64/HVC64	64-bit SMC and HVC calling convention.
Wx	A64 32-bit register view.
Xx	A64 64-bit register view.
Trusted OS	The secure operating system running in the Secure EL1 Exception level. It supports the execution of trusted applications in Secure EL0.

2 SMC AND HVC CALLING CONVENTIONS

2.1 SMC calls

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

2.2 HVC Calls

HVC calls that are made by an Operating System at EL1 result in a synchronous transfer of control to an EL2 hypervisor, and are regarded as HVC exceptions. The operation of the hypervisor is determined by the parameters that are passed in through registers.

2.3 Fast and Yielding types of 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 preempted 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 wholly 32-bit interface that can be used by either 32-bit or 64-bit client code, and passes up to six 32-bit arguments.

  Because only SMC32/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. See also 6.2.

- **SMC64**/**HVC64:** A 64-bit interface that can be used only by 64-bit client code, and passes up to six 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 every SMC and HVC call and it determines:

- The service to be invoked.
- The function to be invoked.
- The calling convention (32-bit or 64-bit) that is in use.
- The call type (fast or yielding) that is in use.

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.

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

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

<table>
<tr><th>Bit Numbers</th><th>Bit Mask</th><th>Description</th></tr>
<tr><td>31</td><td>0x80000000</td><td>If set to 0, the call is a Yielding Call.
If set to 1, the call is a Fast Call (atomic).</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. They are further defined in section 6.
<table>
<tr><th>Owning Entity Number</th><th>Bit Mask</th><th>Description</th></tr>
<tr><td>0</td><td>0x00000000</td><td>ARM Architecture Calls</td></tr>
<tr><td>1</td><td>0x01000000</td><td>CPU Service Calls</td></tr>
<tr><td>2</td><td>0x02000000</td><td>SiP Service Calls</td></tr>
<tr><td>3</td><td>0x03000000</td><td>OEM Service Calls</td></tr>
<tr><td>4</td><td>0x04000000</td><td>Standard Secure Service Calls</td></tr>
<tr><td>5</td><td>0x05000000</td><td>Standard Hypervisor Service Calls</td></tr>
<tr><td>6</td><td>0x06000000</td><td>Vendor Specific Hypervisor Service Calls</td></tr>
<tr><td>7-47</td><td>0x07000000 – 0x2F000000</td><td>Reserved for future use</td></tr>
<tr><td>48-49</td><td>0x30000000 – 0x31000000</td><td>Trusted Application Calls</td></tr>
<tr><td>50-63</td><td>0x32000000 – 0x3F000000</td><td>Trusted OS Calls</td></tr>
</table>
</td></tr>
<tr><td>23:16</td><td>0x00FF0000</td><td>Must be zero (MBZ), for all Fast Calls, when bit[31] == 1.
All other values are reserved for future use.
Note: Some ARMv7 legacy Trusted OS Fast Call implementations have all bits set to 1.</td></tr>
<tr><td>15:0</td><td>0x0000FFFF</td><td>Function number within the range call type that is defined by bits[29:24].</td></tr>
</table>

2.5.1 Conduits

Service calls are expected to be invoked through SMC instructions, except for Standard Hypervisor Calls and Vendor Specific Hypervisor Calls. On some platforms, however, SMC instructions are not available, and the services can be accessed through HVC instructions. The method that is used to invoke the service is referred to as the conduit.

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

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

EL3 Implemented	EL2 Implemented	Conduits	Notes
Yes	Yes	SMC, HVC	
Yes	No	SMC	
No	Yes	HVC	Only permitted on ARMv8-A
No	No	N/A	No conduit required

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

2.6 SMC32/HVC32 Argument passing

When the SMC32/HVC32 convention is used, an SMC or HVC instruction takes a Function Identifier and up to six 32-bit register values as arguments, and can return up to four 32-bit register values. For SMC calls that originate from EL2, an optional seventh 32-bit argument can be added, as described in sections 2.10 and 2.11.

When an SMC32/HVC32 call is made from AArch32:

- A Function Identifier is passed in register R0.
- Arguments are passed in registers R1-R6.
- Results are returned in R0-R3.
- Registers R4-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-W6.
- Results are returned in W0-W3.
- 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: Unused result and scratch registers can leak information after an SMC or HVC call. An implementation can mitigate this risk by either preserving the register state over the call, or returning a constant value, such as zero, in each register.

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

2.7 SMC64/HVC64 Argument passing

When the SMC64/HVC64 convention is used, the SMC or HVC instruction takes up to six 64-bit arguments in registers and can return up to four 64-bit values in registers. For SMC calls originating from EL2 an optional seventh 32-bit argument can be added, as described in sections 2.10 and 2.11.

When an SMC64/HVC64 call is made from AArch64:

- A Function Identifier is passed in register W0.
- Arguments are passed in registers X1-X6.
- Results are returned in X0-X3.

- Registers X18-X30 and stack pointers SP_EL0 and SP_ELx are saved by the function that is called, and must be preserved over the SMC or HVC call.

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

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

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

2.8 SIMD and floating-point registers

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.

All SIMD and floating-point registers are saved by the function that is called, and must be preserved over all SMC and HVC calls.

2.9 SMC and HVC immediate value

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

Therefore:

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

2.10 Client ID (optional)

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

2.10.1 SMC calls

If an implementation includes a hypervisor or similar supervisory software that executes at EL2, it might be necessary to identify from which client operating system 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, see Table 4-1.
- In AArch64, the client ID is passed in the W7 register, 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 generated by software executing at EL1 must be trapped by the hypervisor. Identification information must be inserted into R7 or W7 register before forwarding any SMC call on to the Secure Monitor.

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

2.10.2 HVC calls

The Client ID is ignored by the HVC calling convention.

2.11 Secure OS ID (optional)

In the presence of multiple secure operating systems at S-EL1, the caller must specify for which secure OS 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, see Table 4-1
- In AArch64 state, the secure OS ID is passed in the W7 register, see Table 3-1.

2.12 Session ID (optional)

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

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

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

3 AARCH64 SMC AND HVC CALLING CONVENTIONS

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

For ARM AArch64:

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

3.1 Register use in AArch64 SMC and HVC calls

For the AArch64 calling conventions, the architectural registers, R0-R7, are used.

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 least significant bits.

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

Register Name		Role during SMC or HVC call		
SMC32/HVC32	**SMC64/HVC64**	**Calling values**	**Modified**	**Return state**
SP_ELx		ELx stack pointer	No	Unchanged; registers are saved or restored
SP_EL0		EL0 stack pointer	No	
X30		The Link Register	No	
X29		The Frame Pointer	No	
X19…X28		Registers that are saved by the called function	No	
X18		The Platform Register	No	
X17		The second intra-procedure-call scratch register	Yes	Unpredictable; scratch registers
X16		The first intra-procedure-call scratch register	Yes	
X9…X15		Temporary registers	Yes	
X8		Indirect result location register	Yes	
W7	W7	Optional Client ID in bits[15:0] (ignored for HVC calls) Optional Secure OS ID in bits[31:16]	Yes	
W6	X6 (or W6)	Parameter register Optional Session ID register	Yes	
W4…W5	X4…X5	Parameter registers	Yes	
W1…W3	X1…X3	Parameter registers	Yes	SMC and HVC result registers
W0	W0	Function Identifier	Yes	

For more information, see [4.] *Procedure Call Standard for the ARM 64-bit Architecture*.

4 AARCH32 SMC AND HVC CALLING CONVENTION

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

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

4.1 Register use in AArch32 SMC and HVC calls

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

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

For more information, see [3.] *Procedure Call Standard for the ARM 32-bit Architecture*.

All architecturally banked registered must be preserved over AArch32 calls.

5 SMC AND HVC RESULTS

5.1 Error Codes

Errors codes that are returned in R0, W0 and X0 are considered to be sign-extended to 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 X0 or W0, are sign-extended to 64-bit signed integers, because bits[63-32] are UNDEFINED.

5.2 Unknown Function Identifier

The Unknown SMC Function Identifier is a sign-extended value of (-1) that is returned in R0, W0 or X0 register. 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, or lack of, an SMC or HVC Function. Function Identifiers must be determined from the UID and Revision information.

5.3 Unique Identification (UID) format

This value identifies the implementer of a particular subrange of the API, and therefore what controls the actions of SMCs in that subrange.

The 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 single128-bit value using the SMC32 calling convention. This value is mapped to argument registers as shown in Table 5-1.

Table 5-1: UUID register mapping

Register		Value
AArch32	**AArch64**	
R0	W0	Bytes 0…3 with byte 0 in the low-order bits
R1	W1	Bytes 4…7 with byte 4 in the low-order bits
R2	W2	Bytes 8…11 with byte 8 in the low-order bits
R3	W3	Bytes 12…15 with byte 12 in the low-order bits

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

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

5.4 Revision information format

The revision information for a subrange 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 services that are listed in Table 6-1.

Table 6-1: SMC and HVC Services

Service	Comment
ARM Architecture Service	Provides interfaces to generic services for the ARM Architecture.
CPU Service	Provides interfaces to CPU implementation-specific services for this platform, for example access to errata workarounds.
SiP Service	Provides interfaces to SoC implementation-specific services on this platform, for example secure platform initialization, configuration, and some power control services.
OEM Service	Provides interfaces to OEM-specific services on this platform.
Standard Secure Service	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	Standardized Hypervisor Service Calls allow for common hypervisor discovery mechanism from any Guest OS.
Vendor Specific Hypervisor Service	Proprietary Hypervisor Service Calls.
Trusted Applications	
Trusted OS	

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 who is responsible for that function in a specific SoC. The same entity can be responsible for multiple subranges. Values that are not covered by the table are reserved.

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

SMC Function Identifier	Trusted World subrange ownership
0x00000000-0x0100FFFF	Reserved for existing APIs This region is already in use by the existing ARMv7 devices.
0x02000000-0x1FFFFFFF	Trusted OS Yielding Calls
0x20000000-0x7FFFFFFF	Reserved for future expansion of Trusted OS Yielding Calls
0x80000000-0x8000FFFF	SMC32: ARM Architecture Calls
0x81000000-0x8100FFFF	SMC32: CPU Service Calls
0x82000000-0x8200FFFF	SMC32: SiP Service Calls
0x83000000-0x8300FFFF	SMC32: OEM Service Calls
0x84000000-0x8400FFFF	SMC32: Standard Service Calls
0x85000000-0x8500FFFF	SMC32: Standard Hypervisor Service Calls
0x86000000-0x8600FFFF	SMC32: Vendor Specific Hypervisor Service Calls

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

6.2 General Service Queries

The following general queries are defined for each service:

- **Call Count Query** – Returns a 32-bit count of the available service calls. The count includes both 32 and 64 calling convention service calls and both fast 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, as specified in section 5.4.

If the service is not implemented, general service queries must return the “Unknown Function Identifier” error. The following table recommends the following Function Identifier values to use for general service queries:

Table 6-3: Function Identifier values of general queries

Service	Query[1]		
	Call Count	Call UID	Revision
ARM Architecture Service	0x8000_FF00	0x8000_FF01	0x8000_FF03
CPU Service	0x8100_FF00	0x8100_FF01	0x8100_FF03
SiP Service	0x8200_FF00	0x8200_FF01	0x8200_FF03
OEM Service	0x8300_FF00	0x8300_FF01	0x8300_FF03
Standard Secure Service	0x8400_FF00	0x8400_FF01	0x8400_FF03
Standard Hypervisor Service	0x8500_FF00	0x8500_FF01	0x8500_FF03
Vendor Specific Hypervisor Service	0x8600_FF00	0x8600_FF01	0x8600_FF03
Trusted Applications[2]	-	-	-
Trusted OS	0xBF00_FF00	0xBF00_FF01	0xBF00_FF03

[1] Note that the queries are SMC32/HVC32 Fast calls.

[2] It is the responsibility of a Trusted OS to identify and describe services provided by Trusted Applications.

Besides the values in the table above, the other Function Identifiers in the 0xXXXXFF00 – 0xXXXXFFFF range, for example 0x0800FF02 and 0x0800FF04-0x0800FFFF for ARM Architecture Service, are reserved for future expansion.

6.3 Implemented Standard Secure Service Calls

ARM intends to define a set of Standard Secure Service Calls for the management of the overall system. By standardizing such calls, the job of implementing Operating Systems on ARM is made easier.

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

Table 6-4: Reserved Standard Secure Service Call range

Function Identifier	Reserved for	Notes
0x84000000-0x8400001F	PSCI 32-bit calls	32-bit Power Secure Control Interface. For details of functions and arguments, see [5.].
0xC4000000-0xC400001F	PSCI 64-bit calls	64-bit Power Secure Control Interface. For details of functions and arguments, See [5.].

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

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 preempted by an interrupt, as illustrated 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);
```