Fundamentals of ARMv8-A

Version 1.0

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

Revision Information

The following revisions have been made to this User Guide.

Date	Issue	Confidentiality	Change
03 March 2017	0100	Non-Confidential	First release

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM® in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Where the term ARM is used it means "ARM or any of its subsidiaries as appropriate".

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Unrestricted Access is an ARM internal classification.

Product Status

The information in this document is final, that is for a developed product.

Web Address

http://www.arm.com

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

Contents

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

1 Fundamentals of ARMv8-A

In ARMv8-A, a program executes at one of four Exception levels. In the 64-bit Execution state, the Exception level determines the level of execution privilege, in a similar way to the privilege levels defined in ARMv7-A.

The concept of the Exception level is fundamental to the ARMv8-A architecture. All operations take place at a defined Exception level, and a register can exist in one or more Exception levels. Changing a bit in a register at one Exception level can have a different effect at another Exception level.

Exception levels provide a logical separation of software execution privilege that applies across all operating states of the ARMv8-A architecture. System software determines the Exception level, and therefore the level of privilege, at which software runs. Exception levels are similar to, and support the concept of, hierarchical protection domains common in computer science.

The type of software that typically runs at each of the Exception levels is:

EL0 Normal user applications. EL0 corresponds to the lowest privilege level and is often described as unprivileged, whereas execution at any Exception level above EL0 is often referred to as privileged execution.

EL1 An operating system kernel typically described as privileged.

EL2 Hypervisor.

EL3 Low-level firmware, including the Secure Monitor.

An Exception level (EL*n*) with a larger value of n than another one is said to be at a higher Exception level. An Exception level with a smaller value of n than another is described as being at a lower Exception level.

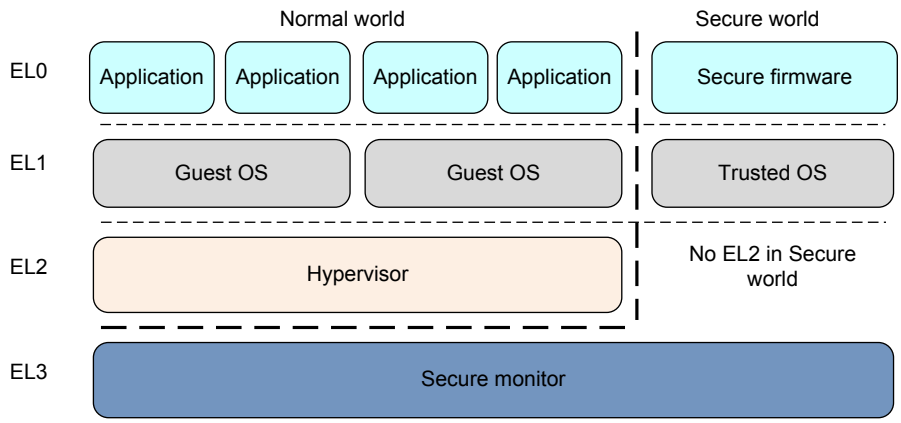
In general, a piece of software, such as an application, the kernel of an operating system, or a hypervisor, occupies a single Exception level. An exception to this is in-kernel hypervisors such as KVM, which operates across both EL2 and EL1.

ARMv8-A also provides two Security states. The *ARM® Architecture Reference Manual* uses the terms Secure and Non-secure to refer to these System security states. Here, the Non-secure state is referred to as the Normal world. Non-secure state does not indicate any security vulnerability, but rather refers to normal operation, and is therefore the same as the Normal world. The word ‘world’ is used to emphasize the relationship between the Secure world and other states that the device is capable of.

The Operating System (OS) runs in the Normal world, in parallel with a trusted OS running in the Secure world on the same hardware. ARM TrustZone® technology enables the system to be partitioned between the Normal and Secure worlds. This provides protection against certain software attacks and hardware attacks. The Secure monitor acts as a gateway for moving between the Normal and Secure worlds. The Secure monitor in the ARMv8-A architecture is at a higher Exception level than all other software.

The following figure shows the Exception levels in the Normal and Secure worlds.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.



ARMv8-A also provides hardware support for virtualization. In the Normal world, virtualization enables more than one OS to co-exist and operate on the same system. This means that a hypervisor or Virtual Machine Manager (VMM) can run on the system and host multiple guest operating systems. Each of the guest operating systems is then, running on a virtual machine. Each OS is unaware that it is sharing time on the system with other guest operating systems.

This means that the Normal world has the following components:

Applications Applications running in the Normal world.

Guest Operating Systems These include Linux or Windows running in Non-secure EL1. When running under a hypervisor, the OS kernels can be running either as a guest or a host, depending on the hypervisor model.

Hypervisor This runs at EL2. The hypervisor, when present and enabled, switches operation between multiple Guest operating systems.

The Secure world has the following components:

Secure firmware On an application processor, Secure firmware must be the first thing that runs at boot time. It provides several services, including platform initialization, the installation of the Trusted OS, and routing of Secure monitor calls. The Secure firmware executes at EL3.

Trusted OS The Trusted OS provides Secure services to the Normal world and provides a runtime environment for executing Secure or trusted applications. It executes at Secure EL1 when EL3 is using AArch64 and at Secure EL3 when EL3 is using AArch32.

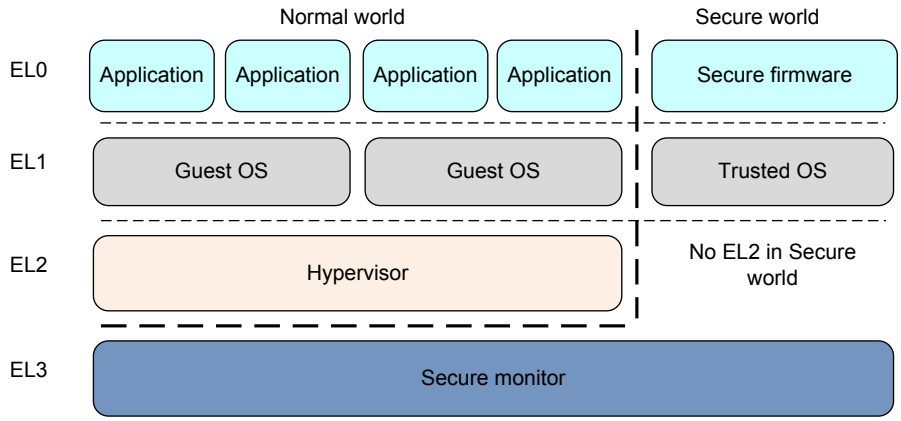
Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

2 Execution states

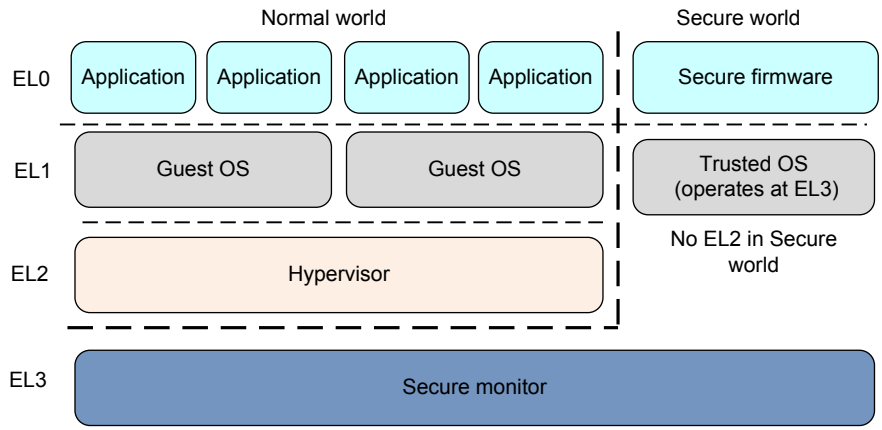
The ARMv8-A architecture also defines two Execution States, *AArch64* and *AArch32*. AArch64 state is unique to ARMv8-A, and uses 64-bit general-purpose registers, while AArch32 state provides backwards compatibility with ARMv7-A using 32-bit general-purpose registers. GNU and Linux documentation (except for Redhat and Fedora distributions) sometimes refers to AArch64 as ARM64.

The AArch32 Execution state is compatible with an ARMv7-A implementation that includes the Virtualization Extensions, the Security Extensions, and the Large Physical Address Extensions. The ARMv8-A architecture allows the execution of different software layers, such as an Application, or an Operating System Kernel or a Hypervisor layer using either AArch32 or AArch64. The ARMv8-A architecture defines how the execution in AArch32 and AArch64 interact.

The following diagram show the organization of the Exception levels in AArch64.



The following diagram show the organization of the Exception levels in AArch32.



In AArch32 state, Trusted OS software executes in Secure EL3, and in AArch64 state it primarily executes in Secure EL1.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

3 Changing Exception levels

Previous versions of the ARM architecture defined an Exception model based on processor modes. For each exception type, the architecture defines the mode to which the exception is taken. This mode is called the target mode for the exception. However, configurable traps, enables, and routing controls can often change the target mode for an exception. ARMv8-A AArch32 follows this model.

When the processor takes an exception it:

- Saves the current program state in the SPSR of the target mode.
- Saves the return address for the exception:
    - In the Link Register (LR) of the target mode if the target mode is not Hyp mode.
    - In ELR_hyp if the target mode is Hyp mode.
- Moves into the target mode. Unless the exception targets Monitor mode, it does so without changing Security state.

The ARMv7-A architecture used Privilege levels PL0 to PL2. In ARMv8-A, the Exception levels have replaced the Privilege levels, but this section explains how PL1 continues to have a particular use. The following table shows the full set of processor modes for an ARMv7-A processor that includes the Virtualization Extensions and the Security Extensions. It also shows the Privilege level that ARMv7-A assigns to each mode, which defines its execution privilege. Execution privilege is defined independently in each Security state.

Mode	Function	Security state	ARMv7-A Privilege level
User (USR)	Unprivileged mode in which most applications run	Both	PL0
FIQ	Entered on an FIQ interrupt exception	Both	PL1
IRQ	Entered on an IRQ interrupt exception	Both	
Supervisor (SVC)	Entered on reset or when a Supervisor Call instruction (SVC) is executed	Both	
Monitor (MON)	Entered when the SMC instruction (Secure Monitor Call) is executed or when the processor takes an exception that is configured to be taken to Monitor mode. Provided to support switching between Secure and Non-secure states.	Secure only	
Abort (ABT)	Entered on a memory access exception	Both	
Undef (UND)	Entered when an UNDEFINED instruction is executed	Both	
System (SYS)	Privileged mode, sharing the register view with User mode	Both	

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

Hyp (HYP)	Entered by the Hypervisor Call and Hyp Trap exceptions.	Non-secure only	PL2

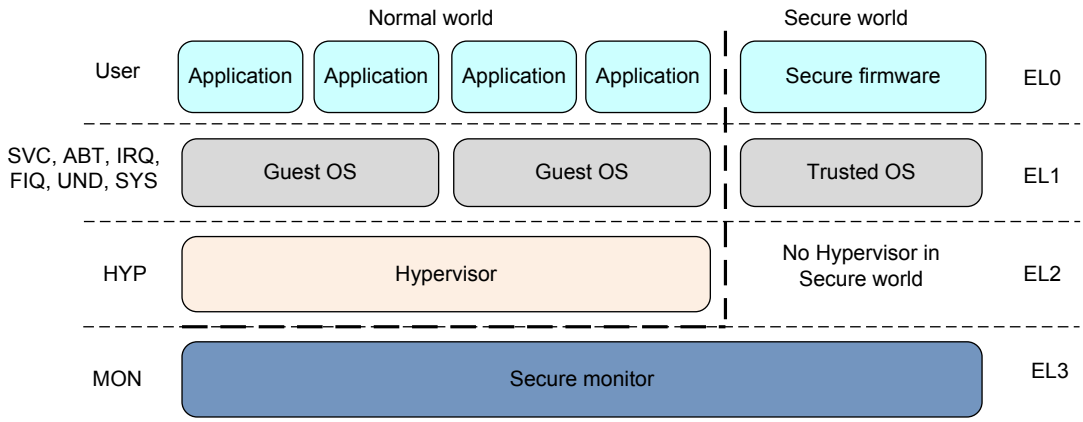
In the ARMv7-A architecture, the processor mode can change under privileged software control or automatically when taking an exception. When an exception occurs, the core saves the current Execution state and the return address, enters the mode that is required to deal with the exception, and possibly disables hardware interrupts. Applications operate at the lowest level of privilege, PL0, previously unprivileged User mode.

Operating systems run at PL1. In a system with the Virtualization Extensions the Hypervisor runs at PL2. The Secure monitor, which acts as a gateway for moving between the Secure and Normal worlds, also operates at PL1.

ARMv8-A does not change this Exception model, but adds the following rules to cover cases that were not possible in ARMv7-A:

- If EL2 is using AArch64, then any exception that targets Hyp mode is taken to EL2 using AArch64.
- If EL3 is using AArch64, then any exception that targets Monitor mode is taken to EL3 using AArch64.

In AArch64, the processor modes are mapped onto the Exception levels as in the following figure.



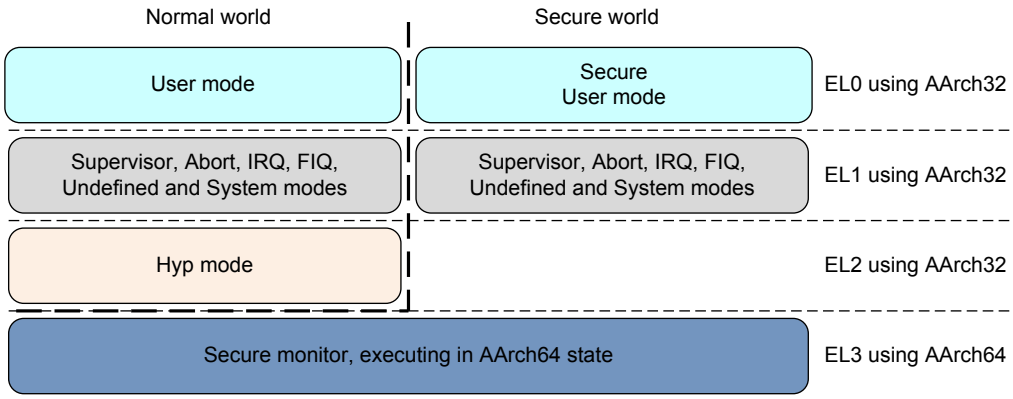
When an exception is taken, the processor changes to the Exception level (equivalent to processor mode in ARMv7-A) which supports the handling of that exception type. The Secure monitor, which operates at PL1 at AArch32, operates at EL3 in AArch64.

3.1 Mapping the processor modes onto the Exception levels

Exception levels that are present in Secure state depend on whether EL3 is using AArch64. This affects how the processor modes map onto the Exception levels.

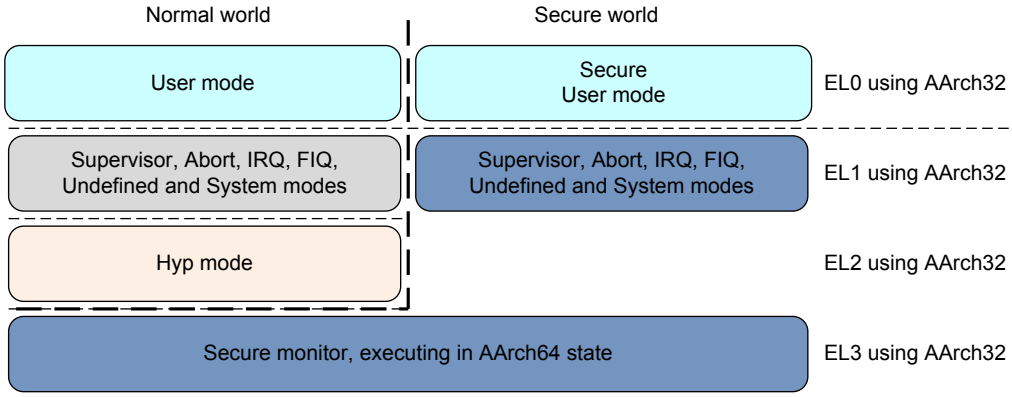
The following figure shows how the AArch32 processor modes map onto the Exception levels when EL3 is using AArch64:

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.



The Monitor mode that was used in ARMv7-A is not present in ARMv8-A. This is because EL3 provides the Secure Monitor functionality, and EL3 is using AArch64.

When EL3 is using AArch32, the mapping of the AArch32 processor modes onto the Exception levels is:



Comparing the two figures, the mapping is unchanged in the Normal world, but in the Secure world the Supervisor mode, Abort mode, IRQ mode, FIQ mode, Undefined mode, and System mode are promoted from EL1 to EL3. This happens because:

- EL3 provides the Secure Monitor functionality.
- The ARMv6 Security Extensions defined Monitor mode as a Secure state mode as peer of Supervisor mode, Abort mode, IRQ mode, FIQ mode, Undefined mode, and System mode. These modes therefore appear as EL3 along with the Secure Monitor functionality

This remapping has no effect on the operation of the processor. Operation within AArch32 state is defined completely in terms of interactions between the processor modes, without reference to any associated Privilege levels or Exception levels.

3.2 Privilege levels in ARMv8-A

The set of modes (Supervisor, Abort, IRQ, FIQ, Undefined, and System) are EL3 modes in Secure state when EL3 is using AArch32, and are EL1 modes under all other circumstances.

To avoid this complicated description, these modes can be described as PL1 modes, reflecting their Privilege level in ARMv7-A. Related to this:

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

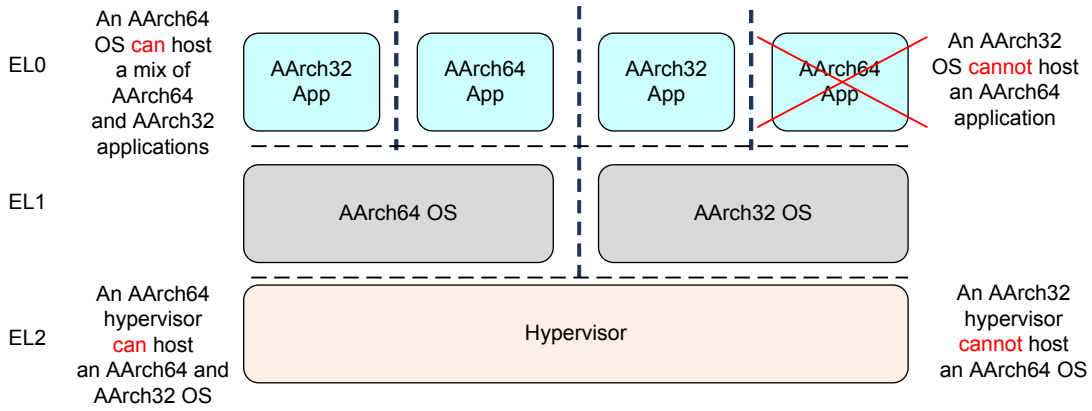
- Controls that affect execution in these modes, in both Security states, can be described as PL1 controls.
- The translation system that is used when executing in these modes or in User mode is called the PL1&0 translation regime.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

4 Changing Execution state

Sometimes the Execution state of your system has to change. This could be, for example, if you are running an AArch64 operating system, and want to run a 32-bit application at EL0. To do this, the system must switch to AArch32. You can only change Execution state by changing Exception level. Taking an exception can change Execution state from AArch32 to AArch64, and returning from an exception can change it from AArch64 to AArch32.

When the application has completed or execution returns to the OS, the system can switch back to AArch64. The following figure shows that you cannot do it the other way around. An AArch32 operating system cannot host a 64-bit application. This is shown in the following figure:



Moving between the two states is performed at the level of the Secure monitor, hypervisor or operating system. A hypervisor or operating system executing in AArch64 state can support AArch32 operation at lower privilege levels. This means that an OS running in AArch64 can host both AArch32 and AArch64 applications. Similarly, an AArch64 hypervisor can host both AArch32 and AArch64 guest operating systems. However, a 32-bit operating system cannot host a 64-bit application and a 32-bit hypervisor cannot host a 64-bit guest operating system.

To change between Execution states at the same Exception level, the system must switch to a higher Exception level and then return to the original Exception level.

As an example, you might have 32-bit and 64-bit applications running under a 64-bit OS. In this case, the 32-bit application can execute and generate a Supervisor Call (`SVC`) instruction, or receive an interrupt, causing a switch to EL1 and AArch64. The OS can then switch tasks and return to EL0 in AArch64. Practically speaking, this means that you cannot have a mixed 32-bit and 64-bit application, because there is no direct way of calling between them.

The main points when changing between AArch64 and AArch32 Execution states can be summarized as follows:

- Changing to AArch32 requires going from a higher to a lower Exception level. This is the result of exiting an exception handler by executing the `ERET` instruction.
- Changing to AArch64 requires going from a lower to a higher Exception level. The exception can be the result of an instruction execution or an external signal.
- If, when taking an exception or returning from an exception, the Exception level remains the same, then the Execution state also cannot change.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

- Both AArch64 and AArch32 Execution states have Exception levels that are similar, but there are some differences between Secure and Non-secure operation. The Execution state the processor is in when the exception is generated can limit the Exception levels available to the other Execution state.
- Where an ARMv8-A processor operates in AArch32 Execution state at a particular Exception level, it uses the same exception model as in ARMv7-A for exceptions that are taken to that Exception level.
- Code at EL3 cannot take an exception to a higher Exception level, so cannot change Execution state, except by going through a reset.

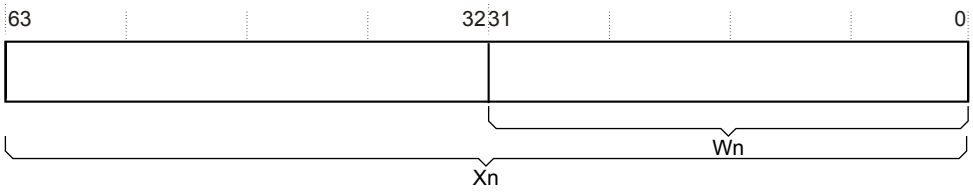
For the highest implemented Exception level (EL3 on most ARMv8-A processors), the Execution state to use for each Exception level when taking an exception is fixed. The Exception level can only be changed by resetting the processor. For EL2 and EL1, when not the highest implemented Exception level, this is controlled by a higher privilege level using System registers.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

5 Registers

ARMv8-A provides 31 × 64-bit general-purpose registers, always accessible, and accessible in all Exception levels. In the AArch64 Execution state, each register (X0-X30) is 64 bits wide. The increased width helps to reduce register pressure in most applications.

Each 64-bit general-purpose register (X0 - X30) also has a 32-bit form (W0 - W30).



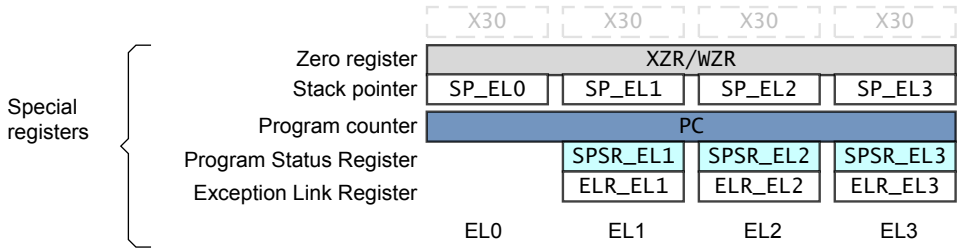
The 32-bit W register forms the lower half of the corresponding 64-bit X register. That is, W0 forms the lower word of X0, and W1 forms the lower word of X1.

Reads from W registers ignore the higher 32 bits of the corresponding X register and leave them unchanged. Writes to W registers set the higher 32 bits of the X register to zero. So, writing `0xFFFFFFFF` into W0 sets X0 to `0x00000000FFFFFFFF`.

Note
Occasionally Rn is used to designate an ARMv8-A register. This means that the register can be either Xn or Wn.

5.1 Special registers

In addition to the thirty one (X0 to X30) ARMv8-A core registers, there are also several special registers.



Note

There is no register that is called X31 or W31. Some instructions are encoded so that the number 31 represents the zero register, ZR (WZR/XZR). There is also a restricted group of instructions in which one or more of the arguments are encoded so that number 31 represents the stack pointer (SP).

Name	Size	Description
WZR	32 bits	Zero register
XZR	64 bits	Zero register
WSP	32 bits	Current stack pointer

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

SP	64 bits	Current stack pointer
PC	64 bits	Program counter

Table 2 Special registers in AArch64

Note

The 64-bit form of the stack pointer does not use an X prefix.

When executing in AArch64, the exception return state is held in the following dedicated registers for each Exception level:

- Exception Link Register (ELR).
- Saved Processor State Register (SPSR).

The following table identifies special registers by Exception level:

	EL0	EL1	EL2	EL3
Stack pointer (SP)	SP_EL0	SP_EL1	SP_EL2	SP_EL3
Exception Link Register (ELR)	-	ELR_EL1	ELR_EL2	ELR_EL3
Saved Process Status Register (SPSR)	-	SPSR_EL1	SPSR_EL2	SPSR_EL3

The *Procedure Call Standard* (PCS) also defines a dedicated Frame Pointer (FP), which makes debugging and call-graph profiling easier by making it possible to unwind the stack reliably.

The Zero register

The zero register does what its name implies.

It ignores all writes to it and all reads of the zero register return 0. You can use the zero register in most, but not all, instructions.

The stack pointer

The stack pointer (SP) is a register that points to the top of the stack. The choice of stack pointer to use is separated to some extent from the Exception level. By default, taking an exception selects the stack pointer for the target Exception level (SP_EL*n*). For example, taking an exception to EL1 selects SP_EL1. Each Exception level has its own stack pointer.

However, when in AArch64 at an Exception level other than EL0, the processor can use either:

- The 64-bit stack pointer that is associated with that Exception level (SP_EL*n*), or,
- The stack pointer that is associated with EL0 (SP_EL0). EL0 can only access SP_EL0.

The SP cannot be referenced by most instructions. However, some arithmetic instructions, for example, the `ADD` instruction, can read and write to the current stack pointer to adjust the stack pointer in a function. For example:

```
ADD SP, SP, #0x10          // Adjust SP to be 0x10 bytes before its current value

ADD SP, SP, #256           // SP = SP + 256
```

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

The Program Counter

The Program Counter (PC) holds the current program address. It cannot be referred to by number as if part of the general register file and therefore cannot be used as the source or destination of arithmetic instructions, or as the base, index or transfer register of load and store instructions.

The only instructions that read the PC are those whose function is to compute a PC-relative address (`ADR`, `ADRP`, literal load, and direct branches), and the branch-and-link instructions that store a return address in the link register (`BL` and `BLR`). The only way to modify the program counter is using branch, exception generation, and exception return instructions.

Where the PC is read by an instruction to compute a PC-relative address, then its value is the address of that instruction. Unlike ARMv7-A, there is no implied offset of 4 or 8 bytes.

The Exception Link Register (ELR)

The Exception Link Register holds the address to return to after an exception.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

6 Processor state

AArch64 does not have a direct equivalent of the ARMv7-A *Current Program Status Register* (CPSR). In AArch64, the components of the traditional CPSR are supplied as fields that can be accessed independently. These are referred to collectively as Processor State (PSTATE). There are also instructions that operate on elements of PSTATE.

The Processor State, or PSTATE fields, for AArch64 have the following definitions:

Name	Description
N	Negative condition flag.
Z	Zero condition flag.
C	Carry condition flag.
V	oVerflow condition flag.
D	Debug mask bit.
A	SError mask bit.
I	IRQ mask bit.
F	FIQ mask bit.
SS	Software Step bit.
IL	Illegal Execution state bit.
EL (2)	Exception level.
nRW	Execution state 0 = 64-bit 1 = 32-bit
SP	Stack pointer selector. 0 = SP_EL0 1 = SP_ELn

PSTATE fields are accessed using special-purpose registers. The Special-purpose registers are read directly using the MRS instruction, and written directly using MSR instructions.

The special registers are:

Special purpose register	Description	PSTATE fields
CurrentEL	Holds the current Exception level.	EL
DAIF	Specifies the current interrupt mask bits.	D, A, I, F
NZCV	Holds the condition flags.	N, Z, C, V
SPSel	At EL1 or higher, this selects between the SP for the current Exception level and SP_EL0.	SP

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

For example, to access the SPSel:

```
MRS X0, SPSel                    //Read SPSel into X0
MSR SPSel, X0                    //Write X0 to SPSel
```

Exception handler code, for example, can switch from using SP_ELn to SP_EL0.

SP_EL1 might point to a piece of memory that holds a small stack that the kernel can guarantee to always be valid. SP_EL0 might point to a kernel task stack that is larger, but not guaranteed to be safe from overflow. This switch is controlled by writing to the SPSel bit, as shown in the following code:

```
MSR SPSel, #0                    // switch to SP_EL0
MSR SPSel, #1                    // switch to the SP of the current Exception
                                 // level ELn
```

Further PSTATE fields can be accessed using the following operands.

Operand	PSTATE fields	Notes
DAIFSet	D, A, I, F	Sets any of the PSTATE.{D,A, I, F} bits to 1
DAIFClr	D, A, I, F	Sets any of the PSTATE.{D,A, I, F} bits to 0
SPSel	SP	Directly sets PSTATE.SP to either 1 or 0

For example:

```
MSR DAIFSet, #Imm4        // Used to set any or all of DAIF to 1

MSR DAIFClr, #Imm4        // Used to clear any or all of DAIF to 0

MSR SPSel, #Imm1          // Used to select the stack pointer, between SP_EL0

                          // and SP_ELn
```

In AArch64, return from an exception is by executing the ERET instruction. This causes the SPSR_ELn to be copied into PSTATE. The ALU flags, Execution state, Exception level, and the processor branches are all restored. From this point, execution continues from the address in ELR_ELn.

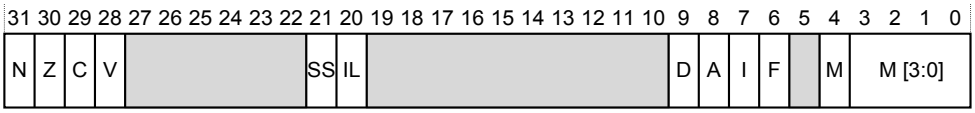
PSTATE.{N, Z, C, V} fields can be accessed at EL0. All other PSTATE fields can be accessed at EL1 or higher and are UNDEFINED at EL0.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

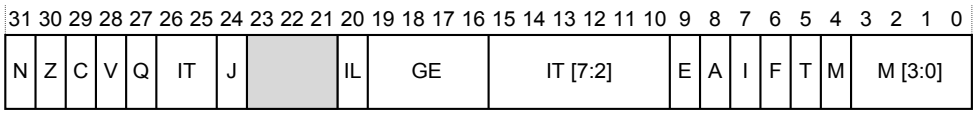
7 The Saved Process Status Register

When taking an exception, the processor state is stored in the relevant Saved Program Status Register (SPSR), in a similar way to the CPSR in ARMv7-A. The SPSR holds the value of PSTATE fields before taking an exception and is used to restore the value of PSTATE fields when executing an exception return.

The following figure shows the SPSR when exceptions are taken from AArch64:



The following figure shows the SPSR when exceptions are taken from AArch32:



The individual bits represent the following values for AArch64:

N Negative result (N flag).

Z Zero result (Z) flag.

C Carry over (C flag).

V Overflow (V flag).

SS Software Step. Indicates whether software step was enabled when an exception was taken.

IL Illegal Execution State bit. Shows the value of PSTATE.IL immediately before the exception was taken.

D Debug exception mask bit. On a reset or taking an exception to AArch64 state, this bit is set to 1.

A SError (System Error) mask bit.

I IRQ mask bit.

F FIQ mask bit.

M[4] Used to record the Execution state (0 indicates AArch64 and 1 indicates AArch32).

M[3:0] Mode or Exception level that an exception was taken from.

In ARMv8-A, the SPSR to be used depends on the Exception level. If the exception is taken in EL1, then SPSR_EL1 is used. If the exception is taken in EL2, then SPSR_EL2 is used, and if the exception is taken in EL3, SPSR_EL3 is used. The core populates the SPSR when taking an exception.

Note

The register pairs ELR_EL*n* and SPSR_EL*n* that are associated with an Exception level retain their state during execution at a lower Exception level.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

8 System registers

System configuration in AArch64 is controlled through system registers accessed using MSR and MRS instructions. This contrasts with ARMv7-A, where system registers are typically accessed through coprocessor 15 (CP15) operations.

The name of a register tells you the lowest Exception level that it can be accessed from. For example:

- TTBR0_EL1 is accessible from EL1, EL2, and EL3.
- TTBR0_EL2 is accessible from EL2 and EL3.

Registers that have the suffix _ELn have a separate, banked copy in some or all the levels, though not EL0. Few system registers are accessible from EL0, although the Cache Type Register (CTR_EL0) is an example of one that is.

Code to access a system register takes the following form:

```
MRS X0, TTBR0_EL1          // Move TTBR0_EL1 into X0

MSR TTBR0_EL1, X0          // Move X0 into TTBR0_EL1
```

Previous versions of the ARM architecture have used coprocessors for system configuration. However, AArch64 does not include support for coprocessors.

The following table shows the Exception levels that have separate copies of each register. For example, separate Auxiliary Control Registers (ACTLRs) exist as ACTLR_EL1, ACTLR_EL2 and ACTLR_EL3.

Name	Register	Description	Allowed values of n
ACTLR_ELn	Auxiliary Control Register	Controls processor-specific features.	1, 2, 3
CCSIDR_ELn	Current Cache Size ID Register	Provides information about the architecture of the currently selected cache.	1
CLIDR_ELn	Cache Level ID Register	The type of cache, or caches, which are implemented at each level. The Level of Coherency and Level of Unification for the cache hierarchy.	1, 2, 3
CNTFRQ_ELn	Counter-timer Frequency Register	Reports the frequency of the system timer.	0
CNTPCT_ELn	Counter-timer Physical Count Register	Holds the 64-bit current count value.	0

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

CNTKCTL_ELn	Counter-timer Kernel Control Register	Controls the generation of an event stream from the virtual counter. Also controls access from EL0 to the physical counter, virtual counter, EL1 physical timers, and the virtual timer.	1
CNTP_CVAL_ELn	Counter-timer Physical Timer Compare Value Register	Holds the compare value for the EL1 physical timer.	0
CPACR_ELn	Coprocessor Access Control Register	Controls access to trace, floating-point, and SIMD functionality.	1
CSSELR_ELn	Cache Size Selection Register	Selects the current Cache Size ID Register, CCSIDR_EL1, by specifying the required cache level and the cache type, either instruction or data cache.	1
CNTP_CTL_ELn	Counter-timer Physical Control Register	Control register for the EL1 physical timer.	0
CTR_ELn	Cache Type Register	Information about the architecture of the integrated caches.	0
DCZID_ELn	Data Cache Zero ID Register	Indicates the block size that is written with byte values of 0 by the Data Cache Zero by virtual address (DCZVA) system instruction.	0
ELR_ELn	Exception Link Register	Holds the address of the instruction which caused the exception.	1, 2, 3
ESR_ELn	Exception Syndrome Register	Includes information about the reasons for the exception.	1, 2, 3
FAR_ELn	Fault Address Register	Holds the virtual faulting address.	1, 2, 3
FPCR	Floating-point Control Register	Controls floating-point extension behavior. The fields in this register map to the equivalent fields in	.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

		the AArch32 FPSCR.	
FPSR	Floating-point Status Register	Provides floating-point system status information. The fields in this register map to the equivalent fields in the AArch32 FPSCR.	.
HCR_ELn	Hypervisor Configuration Register	Controls virtualization settings and trapping of exceptions to EL2.	2
MAIR_ELn	Memory Attribute Indirection Register	Provides the memory attribute encodings corresponding to the possible values in a Long-descriptor format translation table entry for stage 1 translations at ELn.	1, 2, 3
MIDR_ELn	Main ID Register	The type of processor the code is running on (part number and revision).	1
MPIDR_ELn	Multiprocessor Affinity Register	The processor and cluster IDs, in multi-core or cluster systems.	1
RVBAR_ELn	Reset Vector Based Address Register	Holds the reset vector base address for any exception that is taken to ELn.	1, 2, 3
SCR_ELn	Secure Configuration Register	Controls Secure state and trapping of exceptions to EL3.	3
SCTLR_ELn	System Control Register	Controls architectural features, for example the MMU, caches and alignment checking.	0, 1, 2, 3
SPSR_ELn	Saved Program Status Register	Holds the saved processor state when an exception is taken to this mode or Exception level.	abt, fiq, irq, und, 1,2, 3
TCR_ELn	Translation Control Register	Determines which of the Translation Table Base Registers define the base address for a translation table walk required for the stage 1 translation of a memory access from ELn. Also controls the	1, 2, 3

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

		translation table format and holds cacheability and shareability information.	
TPIDR_ELn	User Read/Write Thread ID Register	Provides a location where software executing at ELn can store thread identifying information, for OS management purposes.	0, 1, 2, 3
TPIDRRO_ELn	User Read-Only Thread ID Register	Provides a location where software executing at EL1 or higher can store thread identifying information. This information is visible to software executing at EL0, for OS management purposes.	0
TTBR0_ELn	Translation Table Base Register 0	Holds the base address of translation table 0, and information about the memory it occupies. This is one of the translation tables for the stage 1 translation of memory accesses at ELn.	1, 2, 3
TTBR1_ELn	Translation Table Base Register 1	Holds the base address of translation table 1, and information about the memory it occupies. This is one of the translation tables for the stage 1 translation of memory accesses at EL0 and EL1	1
VBAR_ELn	Vector Based Address Register	Holds the exception base address for any exception that is taken to ELn.	1, 2, 3
VTCR_ELn	Virtualization Translation Control Register	Controls the translation table walks required for the stage 2 translation of memory accesses from Non-secure EL0 and EL1. Also holds cacheability and shareability information for the accesses.	2
VTTBR_ELn	Virtualization Translation Table Base Register	Holds the base address of the translation table for the stage 2 translation of memory accesses from	2

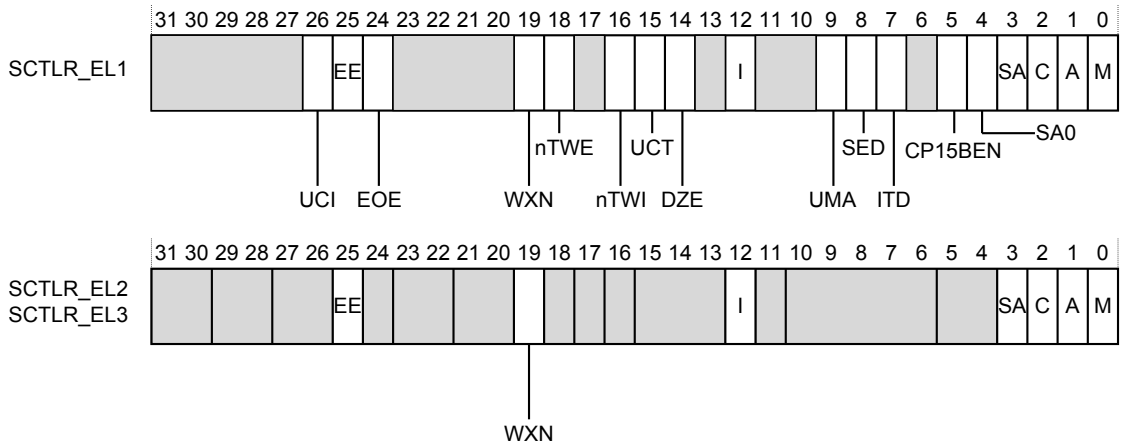
Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

Non-secure EL0 and EL1.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

9 The System Control Register

The System Control Register (SCTLR) is a register that controls standard memory, system facilities and provides status information for functions that are implemented in the core.



Not all bits are available above EL1. The individual bits represent the following:

UCI When this is set, EL0 access for DC CVAU, DC CIVAC, DC CVAC, and IC IVAU instructions is enabled in AArch64.

EE Exception endianness.

0 Little endian

1 Big endian.

EOE Endianness of data accesses at EL0. The possible values of this bit are:

0 Little-endian.

1 Big-endian.

WXN Write permission implies XN (eXecute Never)

0 Regions with write permission are not forced to XN.

1 Regions with write permission are forced to XN.

nTWE A value of 0 means that WFE instructions are trapped to EL1 if the instruction would have caused the core to sleep.

A value of 1 means that WFE instructions are executed as normal.

nTWI A value of 0 means that WFI instructions are trapped to EL1 if the instruction would have caused the core to sleep.

A value of 1 means that WFI instructions are executed as normal.

UCT A value of 1 means EL0 access to the CTR_EL0 register in AArch64 is enabled.

A value of 0 mean EL0 access to the CLR_ELO register in AArch64 is disabled.

DZE Access to DC ZVA instruction at EL0.

0 Execution not allowed.

1 Execution allowed.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

I This is an enable bit for instruction caches at EL0 and EL1.

0 Instruction accesses to Normal memory are not cached.

1 Instruction accesses to Normal memory are cached.

UMA User Mask Access. Controls access to interrupt masks from EL0, when EL0 is using AArch64.

0 Attempts to use an MSR or MSR instruction to access the DAIF is trapped at EL1.

1 Attempt to use an MSR or MRS instruction to access the DAIF is not trapped at EL1.

SED Disables SETEND instructions at EL0 using AArch32.

0 SETEND instructions are enabled.

1 The SETEND instruction is disabled.

ITD IT Disable bit. The possible values of this bit are:

0 The IT instruction is available.

1 The IT instruction is treated as a 16-bit instruction. Only another 16-bit instruction, or the first half of a 32-bit instruction, can follow. This depends on the implementation.

CP15BEN CP15 barrier enable. If implemented, it is an enable bit for the AArch32 CP15 DMB, DSB, and ISB barrier operations.

SA0 Stack Alignment Check Enable for EL0.

SA Stack Alignment Check Enable.

C Data cache enable. This is an enable bit for data caches at EL0 and EL1.

Data accesses to Cacheable Normal memory are cached.

A Alignment check enable bit.

M Enable the MMU.

9.1 Accessing the SCTLR

To access the SCTLR_ELn, use:

```
MRS <Xt>, SCTLR_ELn                    // Read SCTLR_ELn into Xt
MSR SCTLR_ELn, <Xt>                    // Write Xt to SCTLR_ELn
```

As in the following example:

```
MRS X0, SCTLR_EL1                  // Read System Control Register configuration
                                   // data
ORR X0, X0, #(1 << 2)              // Set [C] bit (bit [2]) to enable data caching
ORR X0, X0, #(1 << 12)             // Set [I] bit (bit [12]) to enable instruction
                                   // caching
MSR SCTLR_EL1, X0                  // Write System Control Register configuration
                                   // data
```

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

Note

Caches in the processor must be invalidated before data and instruction caches are enabled in any of the Exception levels.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

10 Changing Execution state (registers)

You can only change Execution state by changing Exception level. Taking an exception can change Execution state from AArch32 to AArch64, and returning from an exception can change it from AArch64 to AArch32.On entry to an Exception level using AArch64 from an Exception level using AArch32:

- The values of the upper 32 bits of registers that were accessible to any lower Exception level using AArch32 execution are UNKNOWN.
- The registers that are not accessible during AArch32 execution retain the state that they had before AArch32 execution.
- On exception entry to EL3, when EL2 was using AArch32, the values of the upper 32 bits of the ELR_EL2 are UNKNOWN.
- When entering an Exception level that is not accessible during AArch32 execution, AArch64 stack pointers (SPs) and Exception Link Registers (ELRs) at that Exception level, retain the state that they had before AArch32 execution. This applies to the following registers:
    - SP_EL0.
    - SP_EL1.
    - SP_EL2.
    - ELR_EL1.

In general, application programmers write applications for either AArch32 or AArch64. It is only the OS that must take account of the two Execution states and the switch between them.

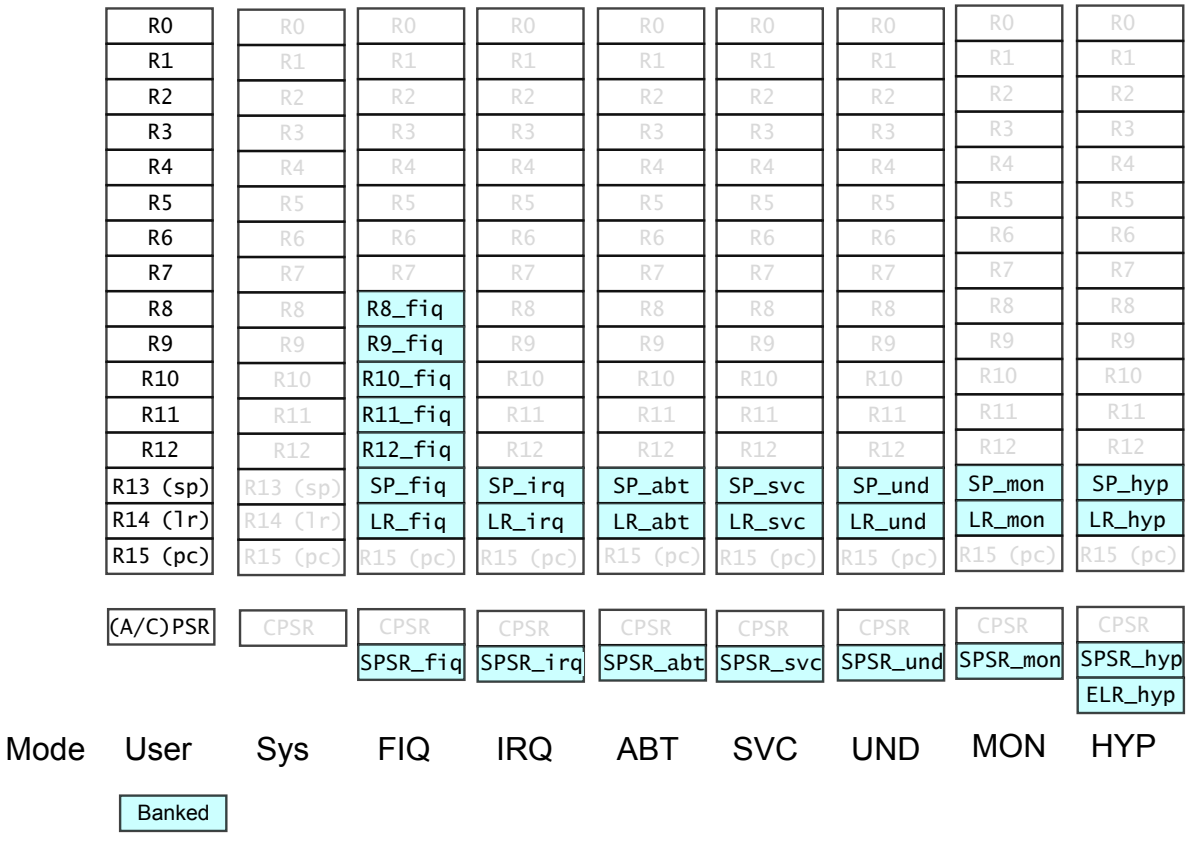
Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

11 Registers at AArch32

Being compatible with ARMv7-A means that, for a processor operating in the AArch32 Execution state, there must be some correspondence between the AArch32 state of the ARMv8-A architecture, and the view of it provided by the ARMv7-A general-purpose registers.

Remember that, in the ARMv7-A architecture, there are sixteen 32-bit general-purpose registers (R0-R15) for software use. Fifteen of them (R0-R14) can be used for general-purpose data storage. The remaining register, R15, is the program counter (PC) whose value is altered as the core executes instructions. Software can also access the CPSR, and the saved copy of the CPSR from the previously executed mode in the SPSR. On taking an exception, the CPSR is copied to the SPSR of the mode to which the exception is taken.

Which of these registers is accessed, and where, depends on the processor mode the software is executing in and the register itself. This is called banking. The shaded registers in Figure 4-7 on page 4-16 are banked. They use physically distinct storage and are usually accessible only when a process is executing in that particular mode.



Banking is used in ARMv7 to reduce the latency for exceptions. However, this also means that of a considerable number of possible registers, fewer than half can be used at any one time.

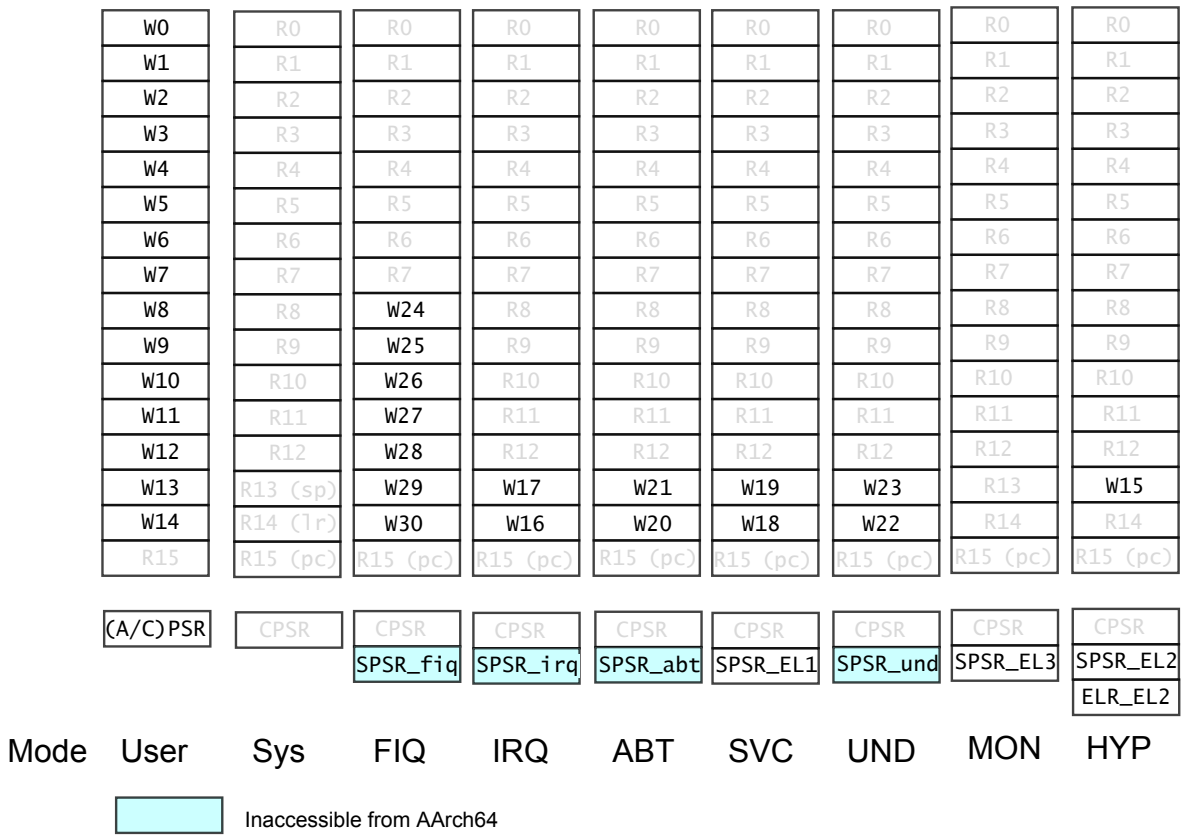
ARMv8-A has 31 ×64-bit general-purpose registers that are always accessible in all Exception levels.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

When taking an exception from AArch32 to AArch64, there are some special considerations. AArch64 handler code can require access to AArch32 registers and the architecture therefore defines mappings to allow access to AArch32 registers.

Bits [63:32] of the X registers are not available in AArch32 state and contain either 0 or the last value that is written in AArch64. There is no architectural guarantee on which value it is. It is therefore usual to access AArch32 registers as W registers.

This mapping is shown in the following figure:



AArch32 also maps the banked registers to AArch64 registers that would otherwise be inaccessible.

The SPSR and ELR_Hyp registers in AArch32 are extra registers that are only accessible using system instructions. They are not mapped into the AArch64 general-purpose register space of the AArch64 architecture. Some of these registers correspond between AArch32 and AArch64:

- SPSR_svc maps to SPSR_EL1.
- SPSR_hyp maps to SPSR_EL2.
- ELR_hyp maps to ELR_EL2.

The following registers are only used during AArch32 execution. However, during execution at EL1 using AArch64, they retain their state and are inaccessible during AArch64 execution at that Exception level.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

- SPSR_abt.
- SPSR_und.
- SPSR_irq.
- SPSR_fiq.

The SPSR registers are only accessible during AArch64 execution at higher Exception levels for context switching.

If an exception is taken to an Exception level using AArch64 from an Exception level using AArch32, the top 32 bits of the AArch64 ELR_ELn are all zero.

11.1 System registers at AArch32

In the ARMv7-A architecture the functionality provided by the System Registers was accessed through CP15 registers.

In much the same way as the mapping between 32-bit W registers at AArch64 map onto the AArch32 General Purpose registers, there is a defined mapping between CP15 registers and AArch64 system registers.

Many system registers are 32-bit, in which case there is a one to one mapping between the AArch32 and the AArch64 instance. For example, the AArch32 Hyp System Control Register (HSCTLR) maps to SCTLR_EL2

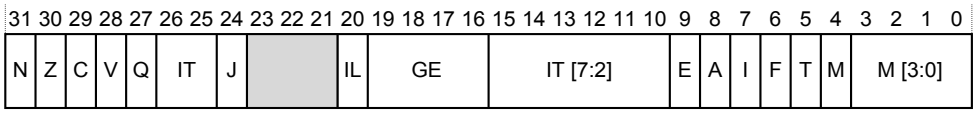
Some AArch64 system registers are 64 bits wide, and these often map to two AArch32 CP15 registers. For example:

- HCR maps to HCR_EL2[31:0].
- HCR2 maps to HCR_EL2[63:32].

11.2 PSTATE at AArch32

In ARMv8-A, the different components of the traditional CPSR are presented as Processor State (PSTATE) fields that can be accessed independently. PSTATE also includes fields that are specific to AArch32 state.

The following figure shows the CPSR bit assignments at AArch32;



Giving extra PSTATE bits which are accessible only at AArch32:

Name	Description
Q	Cumulative saturation (sticky) flag.
GE (4)	Greater than or Equal flags.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

IT (8)	If-Then execution bits.
J	J bit.
T	T32 bit.
E	Endianness bit.
M	Mode field.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

12 A64 instructions

One of the most significant changes introduced in the ARMv8-A architecture was the addition of a new instruction set for AArch64. This instruction set contains many of the same features as the existing AArch32 (ARMv7-A) 32-bit instruction set.

The addition of A64 provides access to 64-bit wide integer registers and data operations, and the ability to use 64-bit sized pointers to memory. The new instructions are called A64 and execute in the AArch64 Execution state. ARMv8-A also includes the original ARM instruction set, now called A32, and the Thumb® (T32) instruction set.

Both A32 and T32 execute in AArch32 state, and provide backward compatibility with ARMv7-A. Although they are similar in many respects, the A64 instruction set is different to the older ISA and is encoded differently. A64 adds some additional capabilities while also removing other features that would potentially limit the speed or energy efficiency of high performance implementations. The ARMv8-A architecture includes some enhancements to the 32-bit instruction sets (A32 and T32) as well. However, code that makes use of such features is not compatible with older ARMv7-A implementations. Instruction opcodes in the A64 instruction set, though, are still 32 bits long, not 64 bits.

The A64 instruction set also provides extra addressing modes with respect to A32, allowing a 64-bit index register to be added to the 64-bit base register, with optional scaling of the index by the access size. Also, it provides sign or zero-extension of a 32-bit value within an index register, again with optional scaling.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

13 The ARMv8-A instruction sets

The A64 instruction set is similar to the existing A32 instruction set. The instructions themselves are still 32 bits wide and have similar syntax.

The instruction sets use a generic naming convention within the ARMv8-A architecture, so that the original 32-bit instruction set states are now called:

A32 When in AArch32 state, the instruction set is largely compatible with ARMv7-A, though there are differences. It also provides some new instructions to align with some of the features that are introduced in the A64 instruction set.

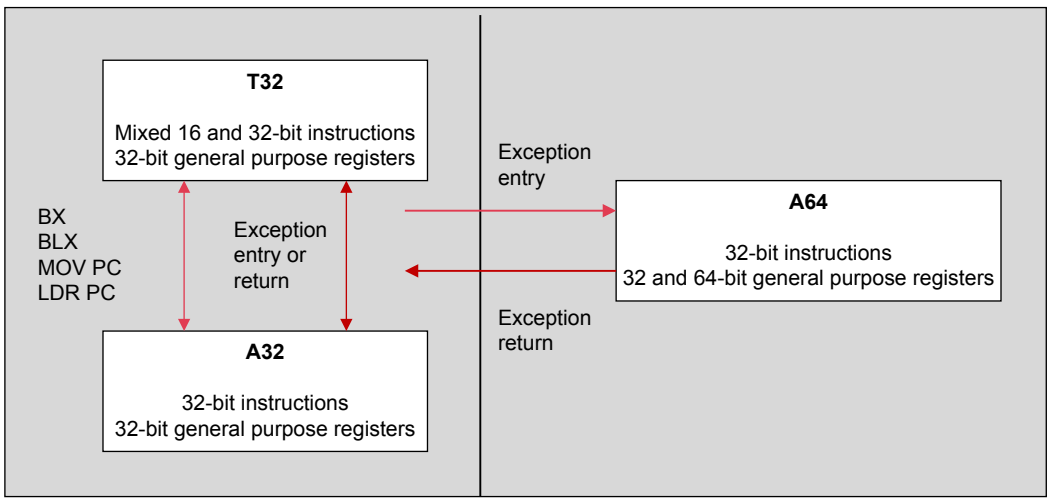
T32 The Thumb instruction set was first included in the ARM7TDMI processor and originally contained only 16-bit instructions. 16-bit instructions gave much smaller programs at the cost of some performance. ARMv7-A processors, including those in the Cortex-A series, support Thumb-2 technology, which extends the Thumb instruction set to provide a mix of 16-bit and 32-bit instructions. This gives performance similar to that of ARM, while retaining the reduced code size. Because of its size and performance advantages, it is increasingly common for all 32-bit code to be compiled or assembled to take advantage of Thumb-2 technology.

In keeping with the naming convention, and reflecting the 64-bit operation, the instruction set for AArch64 is called A64

All A64 instructions are the same length, unlike T32, which is a variable-length instruction set. This makes management and tracking of generated code sequences easier, particularly affecting dynamic code generators.

13.1 Switching between instruction sets

It is not possible to use code from the two Execution states within a single application. There is no interworking between A64 and A32 or T32 instruction sets in ARMv8-A as there is between A32 and T32 instruction sets. Code that is written in A64 for the ARMv8-A processors cannot run on ARMv7-A Cortex®-A series processors. However, code that is written for ARMv7-A processors can run on ARMv8-A processors in the AArch32 Execution state. This is summarized in the following figure.



Copyright © 2017 ARM Limited or its affiliates. All rights reserved.

13.2 Addressing

When the processor can store 64-bit values in a single register, it becomes much simpler to access large amounts of memory within a program. A single thread executing on a 32-bit core is limited to accessing 4GB of address space. Large parts of that addressable space are reserved for use by the OS kernel, library code, peripherals, and more. As a result, lack of space means that the program might need to map some data in or out of memory while executing. Having a larger address space, with 64-bit pointers, avoids this problem. It also makes techniques such as memory-mapped files more attractive and convenient to use. The file contents are mapped into the memory map of a thread, even though the physical RAM might not be large enough to contain the whole file.

Copyright © 2017 ARM Limited or its affiliates. All rights reserved.