Xvisor的ARMv8的启动代码分析
-foundation_v8_boot.S

CORY XIE

获得CurrentEL

```
#define PSR_MODE64_EL3                  0x0000000c
#define PSR_MODE64_EL2h                 0x00000009
#define PSR_MODE64_MASK                 0x0000001f
#define PSR_FIQ_DISABLED                (1 << 6)
#define PSR_IRQ_DISABLED                (1 << 7)
#define PSR_ASYNC_ABORT_DISABLED        (1 << 8)
#define PSR_MODE64_DEBUG_DISABLED       (1 << 9)


        /* Boot-wrapper entry point */
        .section .text, "ax", %progbits
        .globl _start
_start:
        /* Assume EL2 mode if not in EL3 mode */
        mrs     x0, CurrentEL
        cmp     x0, #PSR_MODE64_EL3
        b.ne    _start_el2
```

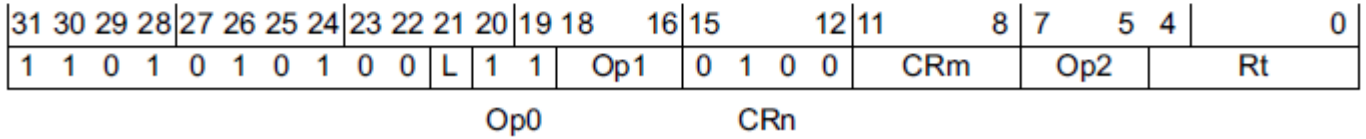
没有了CPSR!

Instructions for accessing special-purpose registers

The A64 instructions for accessing special-purpose registers are:

```
MSR <special-purpose register>, Xt    ; Write to special-purpose register
MRS Xt, <special-purpose register>    ; Read from special-purpose register
```

For these accesses, CRn has the value 4. The encoding for special-purpose register accesses is:



AArch64 The 64-bit Execution state. This Execution state:

- Provides 31 64-bit general-purpose registers, of which X30 is used as the procedure link register.
- Provides a 64-bit *program counter* (PC), *stack pointers* (SPs), and *exception link registers* (ELRs).
- Provides 32 128-bit registers for SIMD vector and scalar floating-point support.
- Provides a single instruction set, A64. For more information, see *The ARM instruction sets* on page A1-34.
- Defines the ARMv8 Exception model, with up to four Exception levels, EL0 - EL3, that provide an *execution privilege* hierarchy, see *Exception levels* on page D1-1408.
- Support for 64-bit *virtual addressing*. For more information, including the limits on address ranges, see Chapter D5 *The AArch64 Virtual Memory System Architecture*.
- Defines a number of PSTATE elements that hold PE state. The A64 instruction set includes instructions that operate directly on various PSTATE elements.
- Names each system register using a suffix that indicates the lowest Exception level at which the register can be accessed.

获得CurrentEL

```
#define PSR_MODE64_EL3                  0x0000000c
#define PSR_MODE64_EL2h                 0x00000009
#define PSR_MODE64_MASK                 0x0000001f
#define PSR_FIQ_DISABLED                (1 << 6)
#define PSR_IRQ_DISABLED                (1 << 7)
#define PSR_ASYNC_ABORT_DISABLED        (1 << 8)
#define PSR_MODE64_DEBUG_DISABLED       (1 << 9)


        /* Boot-wrapper entry point */
        .section .text, "ax", %progbits
        .globl _start
_start:
        /* Assume EL2 mode if not in EL3 mode */
        mrs     x0, CurrentEL
        cmp   x0, #PSR_MODE64_EL3
        b.ne   _start_el2
```

Table C5-7 lists the encodings for Op1, CRm, and Op2 fields for accesses to the special-purpose registers in AArch64.

Table C5-7 Special-purpose register accesses

Register	Access instruction encoding: Op1	CRm	Op2	Notes
SPSR_EL1	0	0	0	Accessible from EL1 or higher.
ELR_EL1			1	
SP_EL0		1	0	Accessible from EL1 or higher. If SP_EL0 is the current stack pointer then the access is UNDEFINED.
SPSel		2	0	Accessible from EL1 or higher.
CurrentEL			2	RO. Accessible from EL1 or higher.
DAIF	3	2	1	Configurable whether accesses at EL0 are permitted.
NZCV			0	Accessible from EL0 or higher.
FPCR		4	0	Accessible from EL0 or higher.
FPSR			1	
DSPSR_EL0		5	0	Accessible only in Debug state, from EL0 or higher.
DLR_EL0			1	
SPSR_EL2	4	0	0	Accessible from EL2 or higher.
ELR_EL2			1	
SP_EL1		1	0	
SPSR_irq		3	0	
SPSR_abt			1	
SPSR_und			2	
SPSR_fiq			3	
SPSR_EL3	6	0	0	Accessible from EL3 or higher.
ELR_EL3			1	
SP_EL2		1	0	

For the accesses to the special-purpose registers shown in Table C5-7:

- Any write to the FPCR must be synchronized, by a *Context synchronization operation*, before its effect on subsequent instructions can be relied upon.
- All other reads and writes to the registers appear to occur in program order relative to other instructions.

新定义了这么些特殊寄存器！

获得CurrentEL

```
#define PSR_MODE64_EL3                  0x0000000c
#define PSR_MODE64_EL2h                 0x00000009
#define PSR_MODE64_MASK                 0x0000001f
#define PSR_FIQ_DISABLED                (1 << 6)
#define PSR_IRQ_DISABLED                (1 << 7)
#define PSR_ASYNC_ABORT_DISABLED        (1 << 8)
#define PSR_MODE64_DEBUG_DISABLED       (1 << 9)


        /* Boot-wrapper entry point */
        .section .text, "ax", %progbits
        .globl _start
_start:
        /* Assume EL2 mode if not in EL3 mode */
        mrs     x0, CurrentEL
        cmp   x0, #PSR_MODE64_EL3
        b.ne   _start_el2
```

Accessing the CurrentEL

To access the CurrentEL:

```
MRS <Xt>, CurrentEL ; Read CurrentEL into Xt
```

Register access is encoded as follows:

op0	op1	CRn	CRm	op2
11	000	0100	0010	010

C5.3.1 CurrentEL, Current Exception Level

The CurrentEL characteristics are:

Purpose

Holds the current exception level.

This register is part of the Process state registers functional group.

Usage constraints

This register is accessible as shown below:

EL0	EL1 (NS)	EL1 (S)	EL2	EL3 (SCR.NS=1)	EL3 (SCR.NS=0)
-	RO	RO	RO	RO	RO

A write to the CurrentEL register is UNDEFINED.

Configurations

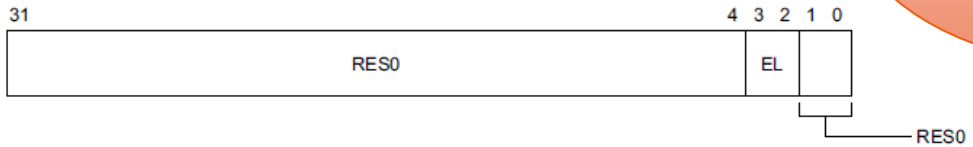
There are no configuration notes.

Attributes

CurrentEL is a 32-bit register.

Field descriptions

The CurrentEL bit assignments are:



Bits [31:4]

Reserved, RES0.

EL, bits [3:2]

Current exception level. Possible values of this field are:

00 EL0

01 EL1

10 EL2

11 EL3

Resets to an IMPLEMENTATION DEFINED value.

Bits [1:0]

Reserved, RES0.

Current Exception Level

获得CurrentEL

```
#define PSR_MODE64_EL3                  0x0000000c
#define PSR_MODE64_EL2h                 0x00000009
#define PSR_MODE64_MASK                 0x0000001f
#define PSR_FIQ_DISABLED                (1 << 6)
#define PSR_IRQ_DISABLED                (1 << 7)
#define PSR_ASYNC_ABORT_DISABLED        (1 << 8)
#define PSR_MODE64_DEBUG_DISABLED       (1 << 9)


        /* Boot-wrapper entry point */
        .section .text, "ax", %progbits
        .globl _start
_start:
        /* Assume EL2 mode if not in EL3 mode */
        mrs     x0, CurrentEL
        cmp   x0, #PSR_MODE64_EL3
        b.ne   _start_el2
```

D1.1 Exception levels

The ARMv8-A architecture defines a set of Exception levels, EL0 to EL3, where:

- If EL*n* is the Exception level, increased values of *n* indicate increased software execution privilege.
- Execution at EL0 is called *unprivileged execution*.
- EL2 provides support for virtualization of Non-secure operation.
- EL3 provides support for switching between two Security states, Secure state and Non-secure state.

An implementation might not include all of the Exception levels. All implementations must include EL0 and EL1. EL2 and EL3 are optional.

Note

A PE is not required to implement a contiguous set of Exception levels. For example, it is permissible for an implementation to include only EL0, EL1, and EL3.

Supported configurations on page D1-1524 shows some example implementations.

When executing in AArch64 state, execution can move between Exception levels only on taking an exception or on returning from an exception:

- On taking an exception, the Exception level can only increase or remain the same.
- On returning from an exception, the Exception level can only decrease or remain the same.

The Exception level that execution changes to or remains in on taking an exception is called the *target Exception level* of the exception.

Each exception type has a target Exception level that is either:

- Implicit in the nature of the exception.
- Defined by configuration bits in the system control registers.

An exception cannot target EL0.

Exception levels exist within a particular Security state. *The ARMv8-A security model* on page D1-1404 describes this. When executing at an Exception level, the PE can access both of the following:

- The resources that are available for the combination of the current Exception level and the current Security state.
- The resources that are available at all lower Exception levels, provided that those resources are available to the current Security state.

This means that if the implementation includes EL3, then when execution is at EL3, the PE can access all resources available at all Exception levels, for both Security states.

Each Exception level other than EL0 has its own translation regime and associated control registers. For information on the translation regimes, see Chapter D4 *The AArch64 Virtual Memory System Architecture*.

D1.1.1 Typical Exception level usage model

The architecture does not specify what software uses which Exception level. Such choices are outside the scope of the architecture. However, the following is a common usage model for the Exception levels:

EL0 Applications.
EL1 OS kernel and associated functions that are typically described as *privileged*.
EL2 Hypervisor.
EL3 Secure monitor.

Exception Level Definition

获得CurrentEL

```
#define PSR_MODE64_EL3                  0x0000000c
#define PSR_MODE64_EL2h                 0x00000009
#define PSR_MODE64_MASK                 0x0000001f
#define PSR_FIQ_DISABLED                (1 << 6)
#define PSR_IRQ_DISABLED                (1 << 7)
#define PSR_ASYNC_ABORT_DISABLED        (1 << 8)
#define PSR_MODE64_DEBUG_DISABLED       (1 << 9)


        /* Boot-wrapper entry point */
        .section .text, "ax", %progbits
        .globl _start
_start:
        /* Assume EL2 mode if not in EL3 mode */
        mrs     x0, CurrentEL
        cmp   x0, #PSR_MODE64_EL3
        b.ne   _start_el2
```

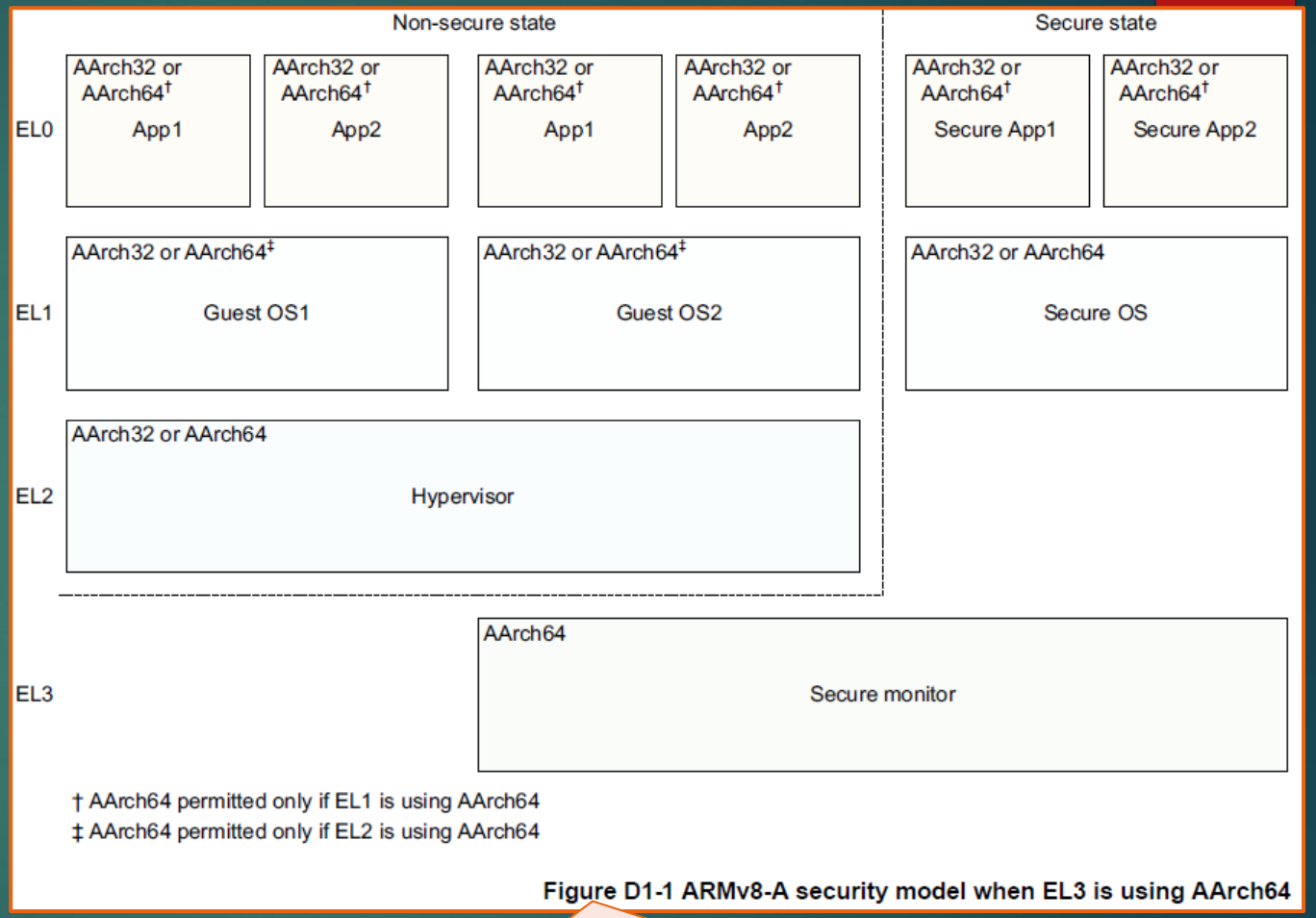


Figure D1-1 ARMv8-A security model when EL3 is using AArch64

Exception Level
&
Security Model

设置EL3 Security Control

```
/* Setup EL3 security control register */
mov   x0, #0x30                /* RES1 */
orr   x0, x0, #(1 << 0)        /* Non-secure EL1 */
orr   x0, x0, #(1 << 8)        /* HVC enable */
orr   x0, x0, #(1 << 10)       /* 64-bit EL2 */
msr   scr_el3, x0
```

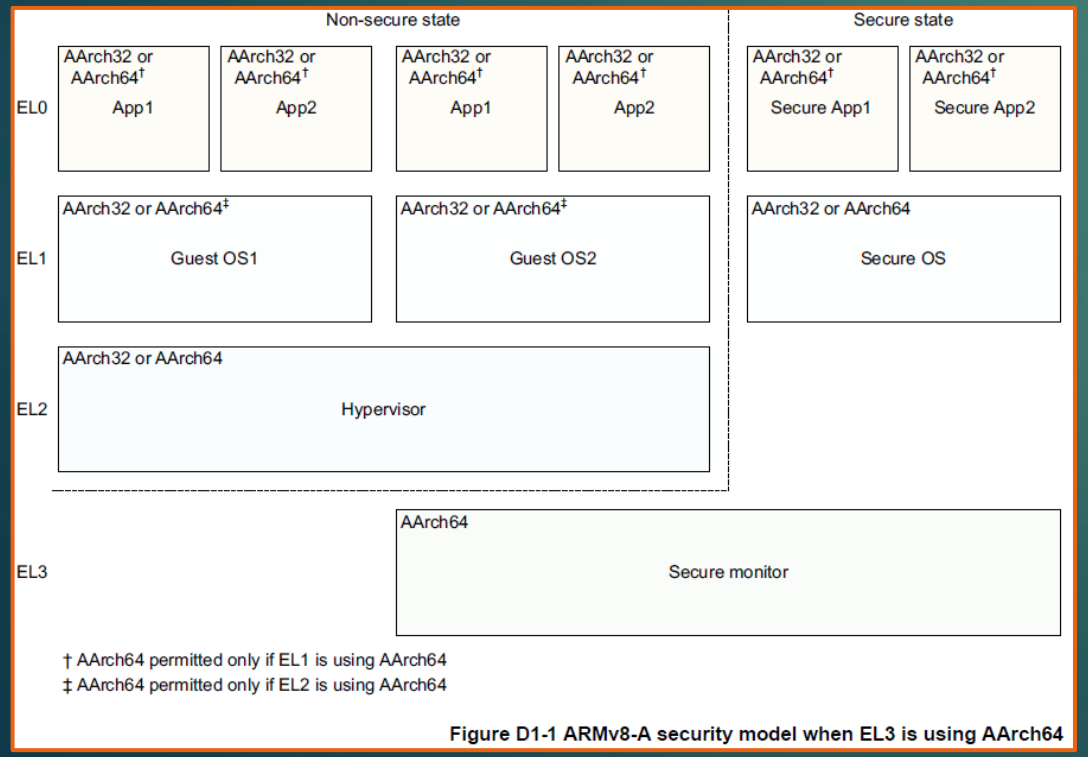


Figure D1-1 ARMv8-A security model when EL3 is using AArch64

D7.2.79 SCR_EL3, Secure Configuration Register

The SCR_EL3 characteristics are:

Purpose

Defines the configuration of the current Security state. It specifies:

- The Security state of EL0 and EL1, either Secure or Non-secure.
- The Execution state at lower exception levels.
- Whether IRQ, FIQ, and External Abort interrupts are taken to EL3.

This register is part of the Security registers functional group.

Usage constraints

This register is accessible as shown below:

EL0	EL1 (NS)	EL1 (S)	EL2	EL3 (SCR.NS=1)	EL3 (SCR.NS=0)
-	-	-	-	RW	RW

Configurations

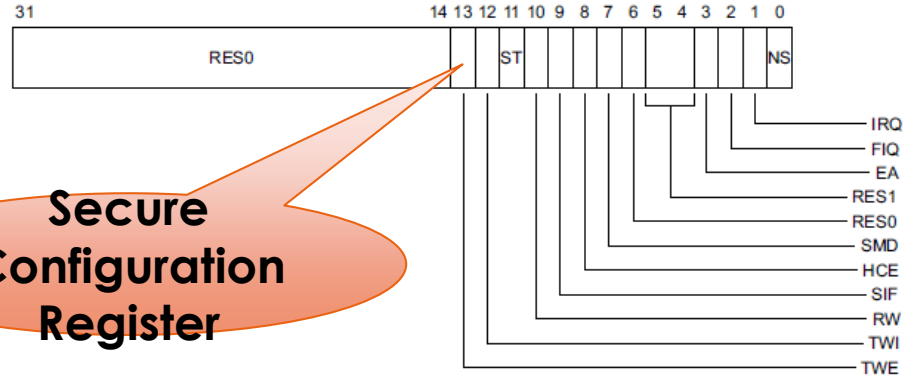
SCR_EL3 can be mapped to AArch32 register SCR, but this is not architecturally mandated.

Attributes

SCR_EL3 is a 32-bit register.

Field descriptions

The SCR_EL3 bit assignments are:



Secure Configuration Register

Bits [31:14]

Reserved, RES0.

TWE, bit [13]

Trap WFE. The possible values of this bit are:

0 WFE instructions not trapped.

1 WFE instructions executed in AArch32 or AArch64 at EL2, EL1, or EL0 are trapped to EL3 if the instruction would otherwise cause suspension of execution, i.e. if there is not a pending WFI wakeup event and the instruction does not cause another exception.

设置EL3 Security Control

```
/* Setup EL3 security control register */
mov   x0, #0x30                /* RES1 */
orr   x0, x0, #(1 << 0)        /* Non-secure EL1 */
orr   x0, x0, #(1 << 8)        /* HVC enable */
orr   x0, x0, #(1 << 10)       /* 64-bit EL2 */
msr   scr_el3, x0
```

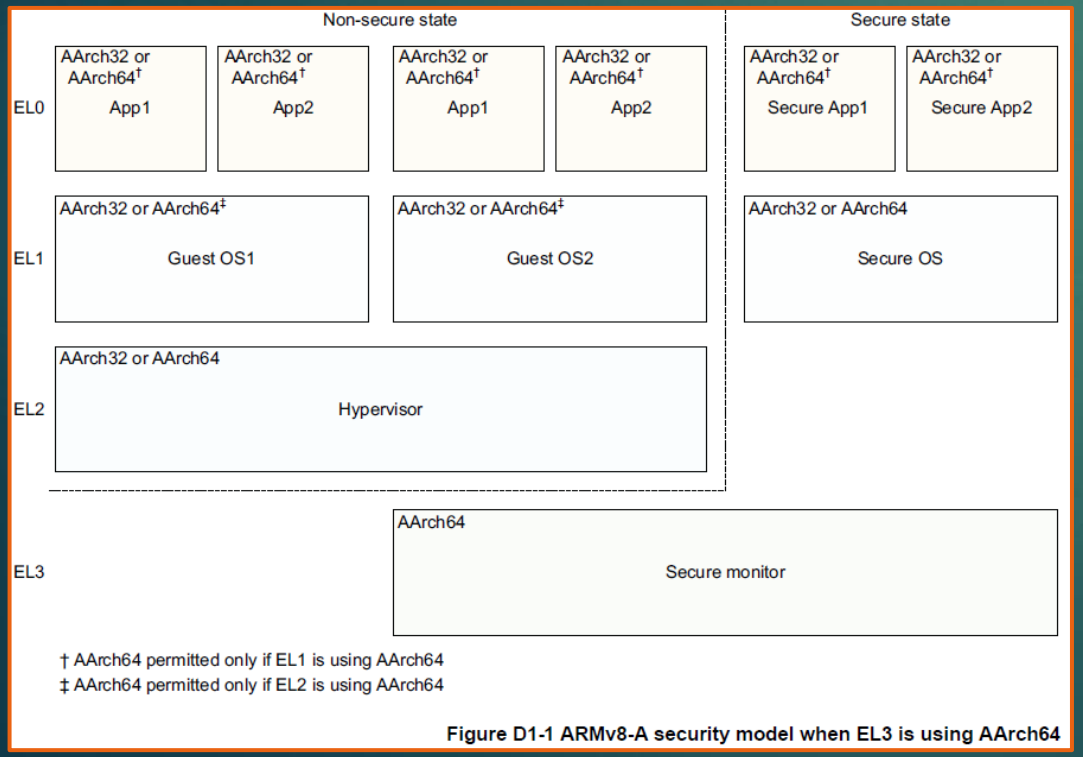


Figure D1-1 ARMv8-A security model when EL3 is using AArch64

TWI, bit [12]

Trap WFI. The possible values of this bit are:

0 WFI instructions not trapped.

1 WFI instructions executed in AArch32 or AArch64 at EL2, EL1, or EL0 are trapped to EL3 if the instruction would otherwise cause suspension of execution.

ST, bit [11]

Enables Secure EL1 access to the CNTPS_TVAL_EL1, CNTPS_CTL_EL1, and CNTPS_CVAL_EL1 registers. The possible values of this bit are:

0 These registers are only accessible in EL3.

1 These registers are accessible in EL3 and also in EL1 when SCR_EL3.NS==0.

If this bit is 0 and there is a Secure EL1 access to one of the CNTPS registers:

- An exception is taken to EL3.
- The exception class for this exception, as returned in ESR_EL3.EC, is 0x18.

RW, bit [10]

Execution state control for lower exception levels.

0 Lower levels are all AArch32.

1 The next lower level is AArch64.

If EL2 is present:

- EL2 is AArch64.
- EL2 controls EL1 and EL0 behaviors.

If EL2 is not present:

- EL1 is AArch64.
- EL0 is determined by the Execution state described in the current process state when executing at EL0.

This bit is permitted to be cached in a TLB.

SIF, bit [9]

Secure instruction fetch. When the processor is in Secure state, this bit disables instruction fetch from Non-secure memory. The possible values for this bit are:

0 Secure state instruction fetches from Non-secure memory are permitted.

1 Secure state instruction fetches from Non-secure memory are not permitted.

This bit is permitted to be cached in a TLB.

HCE, bit [8]

Hypervisor Call enable. This bit enables use of the HVC instruction from Non-secure EL1 modes. The possible values of this bit are:

0 HVC instruction is UNDEFINED in Non-secure EL1 modes, and either UNDEFINED or a NOP in Hyp mode, depending on the implementation.

1 HVC instruction is enabled in Non-secure EL1 modes, and performs a Hypervisor Call.

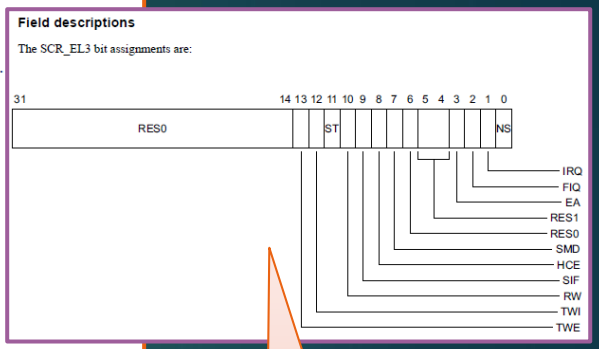
If EL3 is implemented but EL2 is not implemented, this bit is RES0.

SMD, bit [7]

SMC Disable.

0 SMC is enabled at EL1, EL2, or EL3.

1 SMC is UNDEFINED at all exception levels. At EL1 in the Non-secure state, the HCR_EL2.TSC bit has priority over this control.



Secure Configuration Register

设置EL3 Security Control

```
/* Setup EL3 security control register */
mov   x0, #0x30                /* RES1 */
orr   x0, x0, #(1 << 0)        /* Non-secure EL1 */
orr   x0, x0, #(1 << 8)        /* HVC enable */
orr   x0, x0, #(1 << 10)       /* 64-bit EL2 */
msr   scr_el3, x0
```

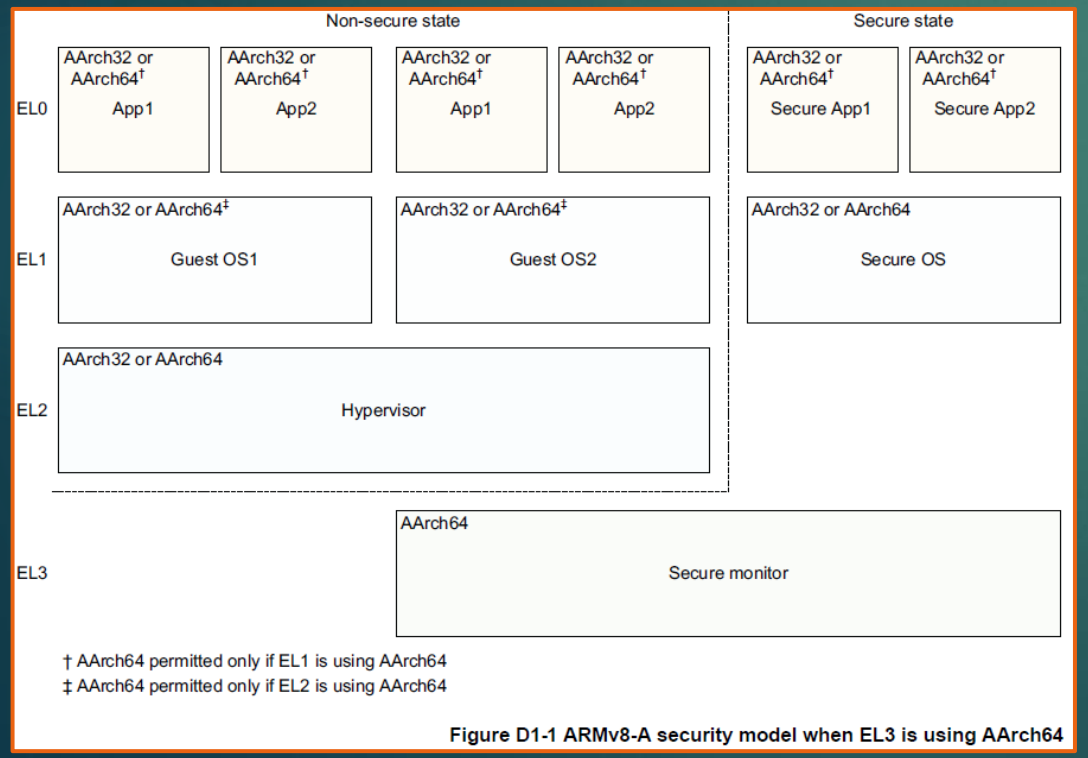
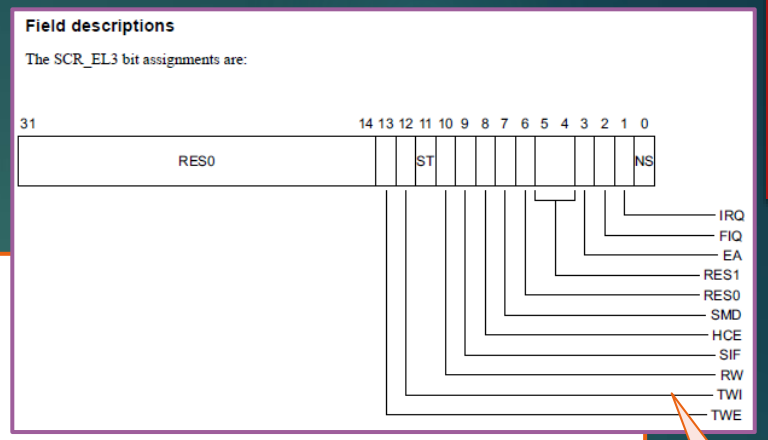


Figure D1-1 ARMv8-A security model when EL3 is using AArch64



Bit [6]

Reserved, RES0.

Bits [5:4]

Reserved, RES1.

EA, bit [3]

External Abort and SError Interrupt Routing.

0 External Aborts and SError Interrupts while executing at exception levels other than EL3 are not taken in EL3.

1 External Aborts and SError Interrupts while executing at all exception levels are taken in EL3.

FIQ, bit [2]

Physical FIQ Routing.

0 Physical FIQ while executing at exception levels other than EL3 are not taken in EL3.

1 Physical FIQ while executing at all exception levels are taken in EL3.

IRQ, bit [1]

Physical IRQ Routing.

0 Physical IRQ while executing at exception levels other than EL3 are not taken in EL3.

1 Physical IRQ while executing at all exception levels are taken in EL3.

NS, bit [0]

Non-secure bit.

0 Indicates that EL0 and EL1 are in Secure state, and so memory accesses from those exception levels can access Secure memory.

1 Indicates that EL0 and EL1 are in Non-secure state, and so memory accesses from those exception levels cannot access Secure memory.

Accessing the SCR_EL3

To access the SCR_EL3:

```
MRS <Xt>, SCR_EL3 ; Read SCR_EL3 into Xt
MSR SCR_EL3, <Xt> ; Write Xt to SCR_EL3
```

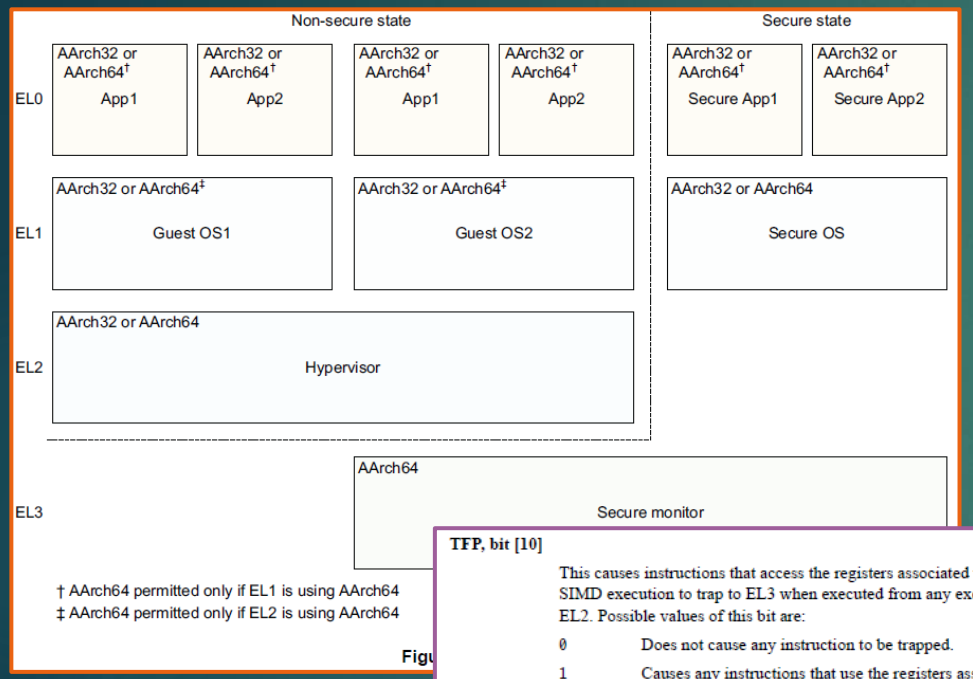
Register access is encoded as follows:

op0	op1	CRn	CRm	op2
11	110	0001	0001	000

Secure Configuration Register

禁止协处理器访问到EL3陷阱

```
/* Disable copro. traps to EL3 */
msr     cptr_el3, xzr
```



TFP, bit [10]

This causes instructions that access the registers associated with Floating Point and Advanced SIMD execution to trap to EL3 when executed from any exception level, unless trapped to EL1 or EL2. Possible values of this bit are:

0 Does not cause any instruction to be trapped.

1 Causes any instructions that use the registers associated with Floating Point and Advanced SIMD execution to be trapped.

Bits [9:0]

Reserved, RES0.

Accessing the CPTR_EL3

To access the CPTR_EL3:

```
MRS <Xt>, CPTR_EL3 ; Read CPTR_EL3 into Xt
MSR CPTR_EL3, <Xt> ; Write Xt to CPTR_EL3
```

Register access is encoded as follows:

op0	op1	CRn	CRm	op2
11	110	0001	0001	010

D7.2.19 CPTR_EL3, Architectural Feature Trap Register (EL3)

The CPTR_EL3 characteristics are:

Purpose

Controls trapping to EL3 of access to CPACR_EL1, Trace functionality and registers associated with Floating Point and Advanced SIMD execution. Also controls EL3 access to this functionality.

This register is part of the Security registers functional group.

Usage constraints

This register is accessible as shown below:

EL0	EL1 (NS)	EL1 (S)	EL2	EL3 (SCR.NS=1)	EL3 (SCR.NS=0)
-	-	-	-	RW	RW

Configurations

There are no configuration notes.

Attributes

CPTR_EL3 is a 32-bit register.

Architectural Feature Trap Register (EL3)

Field descriptions

The CPTR_EL3 bit assignments are:

TCPAC, bit [31]

This causes a direct access to the CPACR_EL1 from EL1 or the CPTR_EL2 from EL2 to trap to EL3 unless it is trapped at EL2. Possible values of this bit are:

0 Does not cause access to the CPACR_EL1 or CPTR_EL2 to be trapped.

1 Causes access to the CPACR_EL1 or CPTR_EL2 to be trapped.

Bits [30:21]

Reserved, RES0.

TTA, bit [20]

This causes access to the Trace functionality to trap to EL3 when executed from EL0, EL1, EL2, or EL3, unless already trapped to EL1 or EL2. Possible values of this bit are:

0 Does not cause System register access to the Trace Functionality to be trapped.

1 Causes System register access to the Trace Functionality to be trapped.

If system register access to trace functionality is not supported, this bit is RES0.

Bits [19:11]

Reserved, RES0.

禁止协处理器访问到EL3陷阱

```
/* Disable copro. traps to EL3 */
msr     cptr_el3, xzr
```

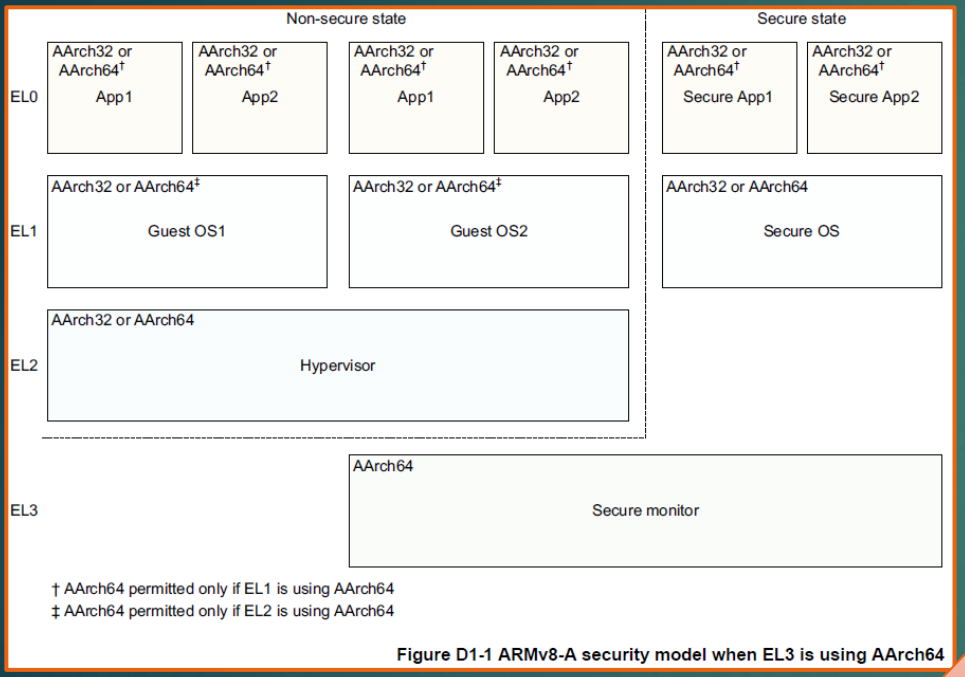


Figure D1-1 ARMv8-A security model when EL3 is using AArch64

专门的0值寄存器！

General-purpose register file and the stack pointer

The 31 general-purpose registers in the general-purpose register file are named R0-R30 and encoded in the instruction register fields with values 0-30. A general-purpose register field that encodes the value 31 represents represents either the current stack pointer or the zero register, depending on the instruction and the operand position.

When the registers are used in a specific instruction variant, they must be qualified to indicate the operand data size, 32 bits or 64 bits, and the data size of the instruction.

When the data size is 32 bits, the lower 32 bits of the register are used and the upper 32 bits are ignored on a read and cleared to zero on a write.

Table C1-2 shows the qualified names for registers, where *n* is a register number 0-30.

Table C1-2 General-purpose register names

Name	Size	Encoding	Description
Wn	32 bits	0-30	General-purpose register 0-30
Xn	64 bits	0-30	General-purpose register 0-30
WZR	32 bits	31	Zero register
XZR	64 bits	31	Zero register
WSP	32 bits	31	Current stack pointer
SP	64 bits	31	Current stack pointer

The following list provides further details relating to Table C1-2.

- The names Xn and Wn both refer to the same general-purpose register, Rn.
- There is no register named W31 or X31.
- The name SP represents the stack pointer for 64-bit operands where an encoding of the value 31 in the corresponding register field is interpreted as a read or write of the current stack pointer. When instructions do not interpret this operand encoding as the stack pointer, use of the name SP is an error.
- The name WSP represents the current stack pointer in a 32-bit context.
- The name XZR represents the zero register for 64-bit operands where an encoding of the value 31 in the corresponding register field is interpreted as returning zero when read or discarding the result when written. When instructions do not interpret this operand encoding as the zero register, use of the name XZR is an error.
- The name WZR represents the zero register in a 32-bit context.
- The architecture does not define a special name for general-purpose register R30 that reflects its special role as the link register on procedure calls. An A64 assembler must always use W30 and X30. Additional software names might be defined as part of the Procedure Call Standard, see *Procedure Call Standard for the ARM 64-bit Architecture*.

设置通用定时器

```
        /* Setup generic timer cntfrq */
        ldr     x0, __gentimer_freq
        msr     cntfrq_el0, x0
...
        .align 3
__gentimer_freq:
        .dword GENTIMER_FREQ
```

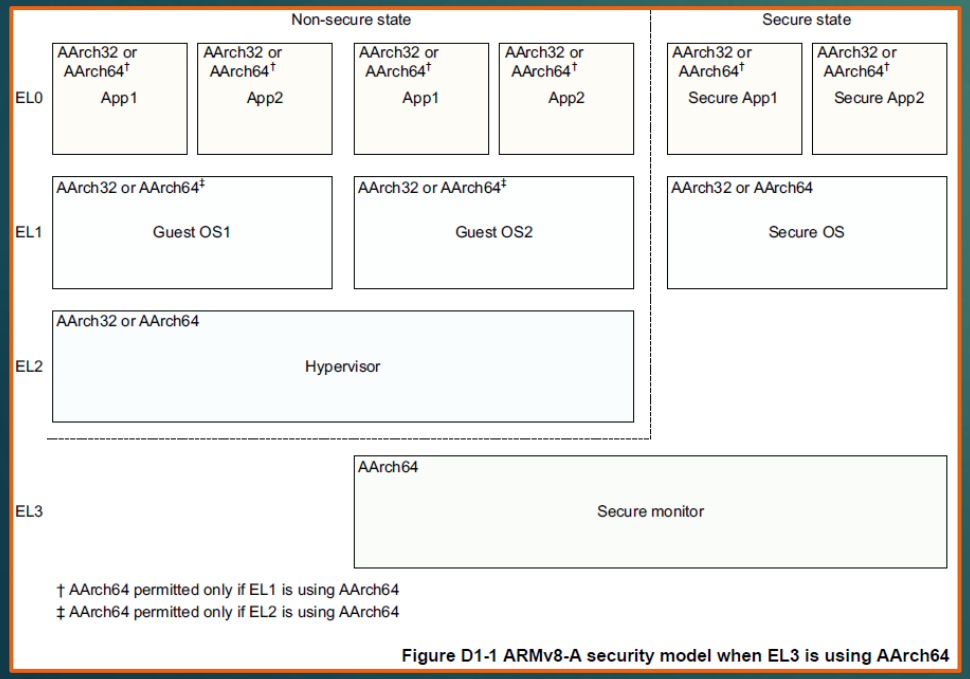


Figure D1-1 ARMv8-A security model when EL3 is using AArch64

D6.1 About the Generic Timer

Figure D6-1 shows an example system-on-chip that uses the Generic Timer as a system timer. In this figure:

- This manual defines the architecture of the individual PEs in the multiprocessor blocks.
- The *ARM Generic Interrupt Controller Architecture Specification* defines a possible architecture for the GICs.
- Generic Timer functionality is distributed across multiple components.

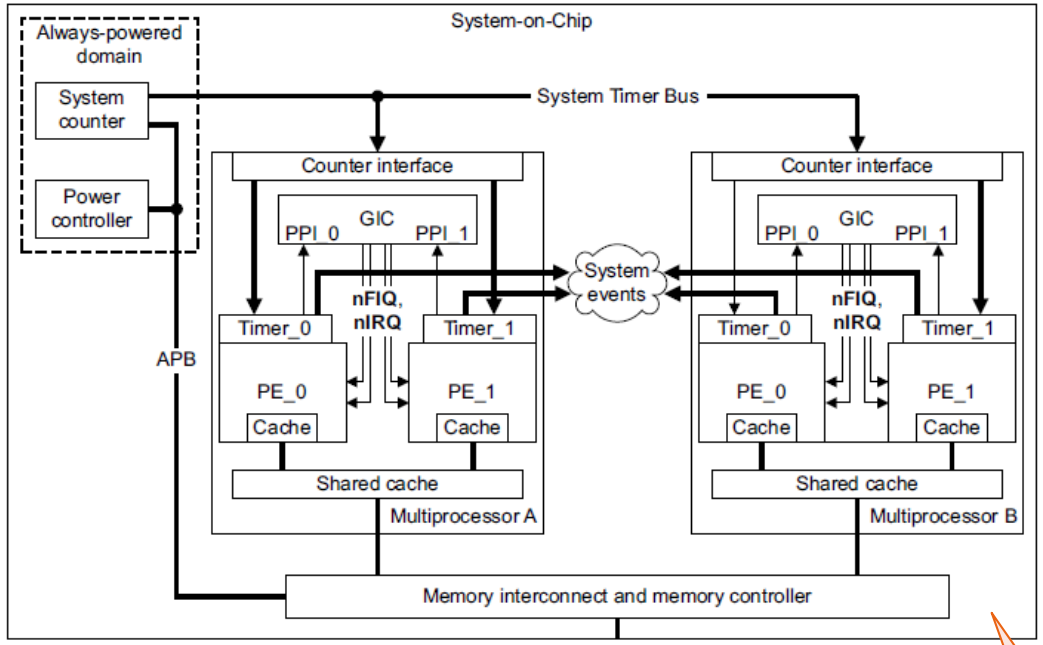


Figure D6-1 Generic Timer exam

This chapter:

- Gives a general description of the Generic Timer.
- Defines the system control register interface to the Generic Timer. Each PE shown in Figure D6-1 includes an implementation of this interface.

The Generic Timer:

- Provides a system counter, that measures the passing of time in real-time.
- Supports *virtual counters* that measure the passing of virtual-time. That is, a virtual counter can measure the passing of time on a particular virtual machine.
- Timers, that can trigger events after a period of time has passed. The timers:
  - Can be used as count-up or as count-down timers.
  - Can operate in real-time or in virtual-time.

系统定时器

GIC Distributor Interface Init

```
        /* GIC Distributor Interface Init */
        mrs     x4, mpidr_el1
        and     x4, x4, #0xffff         /* CPU number */
__gic_dist_init:
        ldr     x0, __gic_dist_base     /* Dist GIC base */
        mov     x1, #0                  /* non-0 cpus should at least */

        cmp     x4, xzr                 /* program IGROUP0 */
        bne     1f
        mov     x1, #3                  /* Enable group0 & group1 */
        str     w1, [x0, #0x00]         /* Ctrl Register */
        ldr     w1, [x0, #0x04]         /* Type Register */
1:      and     x1, x1, #0x1f           /* No. of IGROUPn registers */
        add     x2, x0, #0x080          /* IGROUP0 Register */
        movn    x3, #0                  /* All interrupts to group-1 */
2:      str     w3, [x2], #4
        subs    x1, x1, #1
        bge     2b
...
__gic_dist_base:
        .dword GIC_DIST_BASE
```

D7.2.66 MPIDR_EL1, Multiprocessor Affinity Register

The MPIDR_EL1 characteristics are:

Purpose

In a multiprocessor system, provides an additional processor identification mechanism for scheduling purposes.

This register is part of the Identification registers functional group.

Usage constraints

This register is accessible as shown below:

EL0	EL1 (NS)	EL1 (S)	EL2	EL3 (SCR.NS=1)	EL3 (SCR.NS=0)
-	RO	RO	RO	RO	RO

If EL2 is implemented, reads of MPIDR_EL1 from EL1(NS) return the value from VMPIDR_EL2.

Configurations

MPIDR_EL1 is architecturally mapped to AArch32 register MPIDR.

In a uniprocessor system, ARM recommends that this register returns a value of 0.

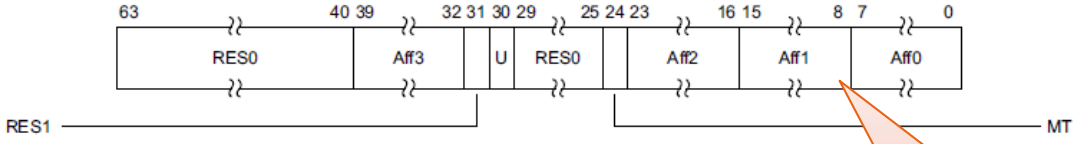
In the system as a whole, the assigned value of all affinity fields in the MPIDR_EL1 for each PE must be unique.

Attributes

MPIDR_EL1 is a 64-bit register.

Field descriptions

The MPIDR_EL1 bit assignments are:



Bits [63:40]

Reserved, RES0.

Aff3, bits [39:32]

Affinity level 3. Highest level affinity field.

Bit [31]

Reserved, RES1.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system. The possible values of this bit are:

0 PE is part of a multiprocessor system.

1 PE is part of a uniprocessor system.

Multiprocessor Affinity Register

GIC Distributor Interface Init

```
        /* GIC Distributor Interface Init */
        mrs     x4, mpidr_el1
        and     x4, x4, #0xffff         /* CPU number */
__gic_dist_init:
        ldr     x0, __gic_dist_base     /* Dist GIC base */
        mov     x1, #0                  /* non-0 cpus should at least */

        cmp     x4, xzr                 /* program IGROUP0 */
        bne     1f
        mov     x1, #3                  /* Enable group0 & group1 */
        str     w1, [x0, #0x00]         /* Ctrl Register */
        ldr     w1, [x0, #0x04]         /* Type Register */
1:      and     x1, x1, #0x1f           /* No. of IGROUPn registers */
        add     x2, x0, #0x080          /* IGROUP0 Register */
        movn    x3, #0                  /* All interrupts to group-1 */
2:      str     w3, [x2], #4
        subs    x1, x1, #1
        bge     2b
...
__gic_dist_base:
        .dword GIC_DIST_BASE
```

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using a multi-threading type approach. The possible values of this bit are:

0 Performance of the PEs at the lowest affinity level is largely independent.

1 Performance of the PEs at the lowest affinity level is very interdependent.

Aff2, bits [23:16]

Affinity level 2. Second highest level affinity field.

Aff1, bits [15:8]

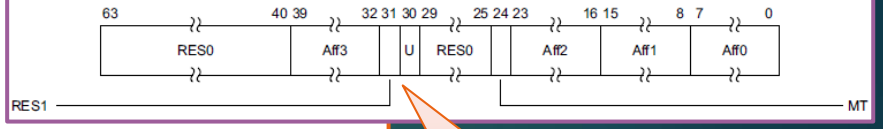
Affinity level 1. Third highest level affinity field.

Aff0, bits [7:0]

Affinity level 0. Lowest level affinity field.

Field descriptions

The MPIDR_EL1 bit assignments are:



Note

- The interpretation of these fields is IMPLEMENTATION DEFINED, and must be documented as part of the documentation of the multiprocessor system.
- The software mechanism to discover the total number of affinity numbers used at each level is IMPLEMENTATION DEFINED, and is part of the general system identification task.

Multiprocessor Affinity Register

Multi-threading approach to lowest affinity levels

If MIDR_EL1.MT is set to 1, this indicates that the PEs at affinity level 0 are logical PEs, implemented using a multi-threading type approach. In such an approach, there can be a significant performance impact if a new thread is assigned to the PE with:

- A different affinity level 0 value to some other thread, referred to as the original thread.
- A pair of values for affinity levels 1 and 2 that are the same as the pair of values of the original thread.

In this situation, the performance of the original thread might be significantly reduced.

Note

In this description a thread always refers to a thread or a PE.

Accessing the MPIDR_EL1:

To access the MPIDR_EL1:

```
MRS <Xt>, MPIDR_EL1 ; Read MPIDR_EL1 into Xt
```

Register access is encoded as follows:

op0	op1	CRn	CRm	op2
11	000	0000	0000	101

GIC Distributor Interface Init

```
        /* GIC Distributor Interface Init */
        mrs     x4, mpidr_el1
        and     x4, x4, #0xffff         /* CPU number */
__gic_dist_init:
        ldr     x0, __gic_dist_base     /* Dist GIC base */
        mov     x1, #0                  /* non-0 cpus should at least */

        cmp     x4, xzr                 /* program IGROUP0 */
        bne     1f
        mov     x1, #3                  /* Enable group0 & group1 */
        str     w1, [x0, #0x00]         /* Ctrl Register */
        ldr     w1, [x0, #0x04]         /* Type Register */
1:      and     x1, x1, #0x1f           /* No. of IGROUPn registers */
        add     x2, x0, #0x080          /* IGROUP0 Register */
        movn    x3, #0                  /* All interrupts to group-1 */
2:      str     w3, [x2], #4
        subs    x1, x1, #1
        bge     2b
...
__gic_dist_base:
        .dword GIC_DIST_BASE
```

0x00_2C00_1000	0x00_2C00_1FFF	GIC Distributor	GIC Distributor[a]	4KB	-
0x00_2C00_2000	0x00_2C00_2FFF	GIC Processor Interface	GIC Processor Interface[a]	4KB	-
0x00_2C00_4000	0x00_2C00_4FFF	GIC Processor Hyp Interface	GIC Processor Hyp Interface[a]	4KB	-
0x00_2C00_5000	0x00_2C00_5FFF	GIC Hyp Interface	GIC Hyp Interface[a]	4KB	-
0x00_2C00_6000	0x00_2C00_7FFF	GIC Virtual CPU Interface	GIC Virtual CPU Interface[a]	8KB	-
0x00_2C01_0000	0x00_2C01_0FFF	-	GIC Virtual Interface Control, GICH[b]	4KB	S/NS

Table 3-2 ARMv8-A Foundation Model memory map (continued)

Start address	End address	Foundation v1 peripheral	Foundation v2 peripheral	Size	Security (v2 only)
0x00_2C02_F000	0x00_2C03_0FFF	-	GIC Virtual CPU Interface, GICV[b]	8KB	S/NS
0x00_2C09_0000	0x00_2C09_FFFF	-	Warning + RAZ/W		
0x00_2E00_0000	0x00_2E00_FFFF	-	Non-trusted SRAM	64KB	S/NS
0x00_2F00_0000	0x00_2F00_FFFF	-	GICv3 Distributor GICD[b]	64KB	S/NS
0x00_2F02_0000	0x00_2F03_FFFF	-	GICv3 Distributor ITS[b]	128KB	S/NS
0x00_2F10_0000	0x00_2F1F_FFFF	-	GICv3 Distributor GICR[b]	1MB	S/NS

ARMv8-A Foundation Model memory map (GIC section)

GIC Distributor Interface Init

```
        /* GIC Distributor Interface Init */
        mrs     x4, mpidr_el1
        and     x4, x4, #0xffff         /* CPU number */
__gic_dist_init:
        ldr     x0, __gic_dist_base     /* Dist GIC base */
        mov     x1, #0                  /* non-0 cpus should at least */

        cmp     x4, xzr                 /* program IGROUP0 */
        bne     1f
        mov     x1, #3                  /* Enable group0 & group1 */
        str     w1, [x0, #0x00]         /* Ctrl Register */
        ldr     w1, [x0, #0x04]         /* Type Register */
1:      and     x1, x1, #0x1f           /* No. of IGROUPn registers */
        add     x2, x0, #0x080          /* IGROUP0 Register */
        movn    x3, #0                  /* All interrupts to group-1 */
2:      str     w3, [x2], #4
        subs    x1, x1, #1
        bge     2b
...
__gic_dist_base:
        .dword GIC_DIST_BASE
```

GIC partitioning

2.1 About GIC partitioning

The GIC architecture splits logically into a Distributor block and one or more CPU interface blocks. The GIC Virtualization Extensions add one or more virtual CPU interfaces to the GIC. Therefore, as Figure 2-1 on page 2-23 shows, the logical partitioning of the GIC is as follows:

Distributor The Distributor block performs interrupt prioritization and distribution to the CPU interface blocks that connect to the processors in the system.

The Distributor block registers are identified by the GICD_ prefix.

CPU interfaces Each CPU interface block performs priority masking and preemption handling for a connected processor in the system.

CPU interface block registers are identified by the GICC_ prefix.

When describing a GIC that includes the GIC Virtualization Extensions, a CPU interface is sometimes called a *physical CPU interface*, to avoid possible confusion with a virtual CPU interface.

Virtual CPU interfaces

The GIC Virtualization Extensions add a virtual CPU interface for each processor in the system. Each virtual CPU interface is partitioned into the following blocks:

Virtual interface control

The main component of the virtual interface control block is the GIC virtual interface control registers, that include a list of active and pending virtual interrupts for the current virtual machine on the connected processor. Typically, these registers are managed by the hypervisor that is running on that processor.

Virtual interface control block registers are identified by the GICH_ prefix.

Virtual CPU interface

Each virtual CPU interface block provides physical signaling of virtual interrupts to the connected processor. The ARM processor Virtualization Extensions signal these interrupts to the current virtual machine on that processor. The GIC virtual CPU interface registers, accessed by the virtual machine, provide interrupt control and status information for the virtual interrupts. The format of these registers is similar to the format of the physical CPU interface registers.

Virtual CPU interface block registers are identified by the GICV_ prefix.

Note

The virtual CPU interface does not support the power management functionality described in *Power management, GIC v2* on page 2-31.

A GIC can implement up to eight CPU interfaces, numbered from 0-7. In a GIC that implements the GIC Virtualization Extensions, virtual CPU interface numbering corresponds to the CPU interface numbering, so that CPU interface 0 and virtual CPU interface 0 connect to the same processor.

This model supports implementation of the GIC in uniprocessing or multiprocessing environments, and the GIC Virtualization Extensions extend that support to processors that support virtualization, in which, in Non-secure state:

- A Guest OS runs on a virtual machine
- A hypervisor is responsible for switching between virtual machines. This switching includes switching the state held in the GIC virtual interface control registers.

Each block provides part of the GIC programmers' model, and:

- the programmers' model is generally the same for each implemented CPU interface.
- the programmers' model for a virtual CPU interface is generally the same as the programmers' model for a physical CPU interface.

GIC Distributor Interface Init

```
        /* GIC Distributor Interface Init */
        mrs     x4, mpidr_el1
        and     x4, x4, #0xffff         /* CPU number */
__gic_dist_init:
        ldr     x0, __gic_dist_base     /* Dist GIC base */
        mov     x1, #0                  /* non-0 cpus should at least */

        cmp     x4, xzr                 /* program IGROUP0 */
        bne     1f
        mov     x1, #3                  /* Enable group0 & group1 */
        str     w1, [x0, #0x00]         /* Ctrl Register */
        ldr     w1, [x0, #0x04]         /* Type Register */
1:      and     x1, x1, #0x1f           /* No. of IGROUPn registers */
        add     x2, x0, #0x080          /* IGROUP0 Register */
        movn    x3, #0                  /* All interrupts to group-1 */
2:      str     w3, [x2], #4
        subs    x1, x1, #1
        bge     2b
...
__gic_dist_base:
        .dword GIC_DIST_BASE
```

Note

- The partitioning of the GIC described in this section is an architectural abstraction. Whether these blocks are implemented separately or combined is IMPLEMENTATION SPECIFIC.
- In a GIC that implements the GIC Security Extensions in a multiprocessor system, a CPU interface can be implemented so that it receives:
  - both Secure and Non-secure accesses
  - only Secure accesses
  - only Non-secure accesses.

GIC partitioning

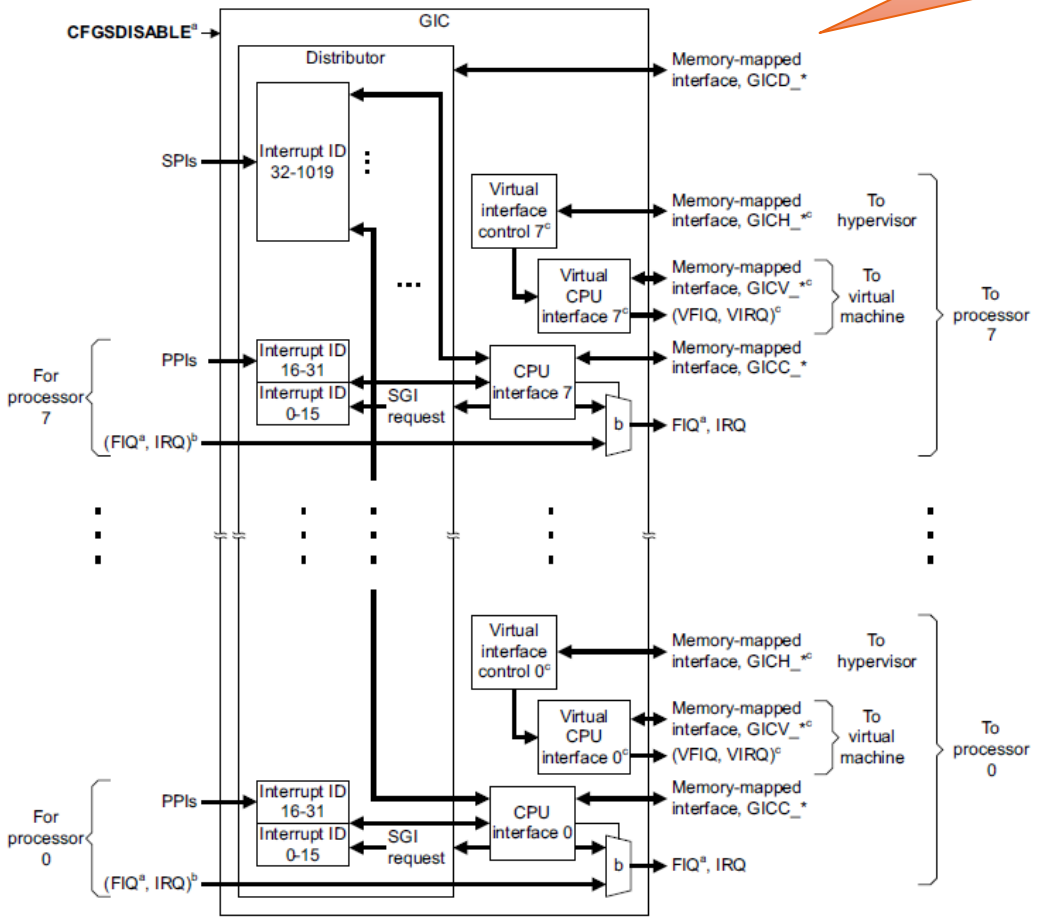


Figure 2-1 GIC logical partitioning

The remainder of this chapter, and Chapter 3 *Interrupt Handling and Prioritization* and Chapter 4 *Programmers' Model*, describe the GIC without the GIC Virtualization Extensions. Chapter 5 *GIC Support for Virtualization* describes the features added by the GIC Virtualization Extensions.

GIC Distributor Interface Init

```
        /* GIC Distributor Interface Init */
        mrs     x4, mpidr_el1
        and     x4, x4, #0xffff         /* CPU number */
__gic_dist_init:
        ldr     x0, __gic_dist_base     /* Dist GIC base */
        mov     x1, #0                  /* non-0 cpus should at least */

        cmp     x4, xzr                 /* program IGROUP0 */
        bne     1f
        mov     x1, #3                  /* Enable group0 & group1 */
        str     w1, [x0, #0x00]         /* Ctrl Register */
        ldr     w1, [x0, #0x04]         /* Type Register */
1:      and     x1, x1, #0x1f           /* No. of IGROUPn registers */
        add     x2, x0, #0x080          /* IGROUP0 Register */
        movn    x3, #0                  /* All interrupts to group-1 */
2:      str     w3, [x2], #4
        subs    x1, x1, #1
        bge     2b
...
__gic_dist_base:
        .dword GIC_DIST_BASE
```

- Interrupt numbers ID1020-ID1023 are reserved for special purposes, see *Special interrupt numbers* on page 3-43.

System software sets the priority of each interrupt. This priority is independent of the interrupt ID number.

In any system that implements the ARM Security Extensions, to support a consistent model for message passing between processors, ARM strongly recommends that all processors reserve:

- ID0-ID7 for Non-secure interrupts
- ID8-ID15 for Secure interrupts.

2.2 The Distributor

The Distributor centralizes all interrupt sources, determines the priority of each interrupt, and for each CPU interface forwards the interrupt with the highest priority to the interface, for priority masking and preemption handling.

The Distributor provides a programming interface for:

- Globally enabling the forwarding of interrupts to the CPU interfaces.
- Enabling or disabling each interrupt.
- Setting the priority level of each interrupt.
- Setting the target processor list of each interrupt.
- Setting each peripheral interrupt to be level-sensitive or edge-triggered.
- Setting each interrupt as either Group 0 or Group 1.

Note

For GICv1, setting interrupts as Group 0 or Group 1 is possible only when the implementation includes the GIC Security Extensions.

- Forwarding an SGI to one or more target processors.

In addition, the Distributor provides:

- visibility of the state of each interrupt
- a mechanism for software to set or clear the pending state of a peripheral interrupt.

2.2.1 Interrupt IDs

Interrupts from sources are identified using *ID numbers*. Each CPU interface can see up to 1020 interrupts. The banking of SPIs and PPIs increases the total number of interrupts supported by the Distributor.

The GIC assigns interrupt ID numbers ID0-ID1019 as follows:

- Interrupt numbers ID32-ID1019 are used for SPIs.
- Interrupt numbers ID0-ID31 are used for interrupts that are private to a CPU interface. These interrupts are banked in the Distributor.

  A banked interrupt is one where the Distributor can have multiple interrupts with the same ID. A banked interrupt is identified uniquely by its ID number and its associated CPU interface number. Of the banked interrupt IDs:

  — ID0-ID15 are used for SGIs

  — ID16-ID31 are used for PPIs

  In a multiprocessor system:

  — A PPI is forwarded to a particular CPU interface, and is private to that interface. In prioritizing interrupts for a CPU interface the Distributor does not consider PPIs that relate to other interfaces.

  — Each connected processor issues an SGI by writing to the GICD_SGIR in the Distributor. Each write can generate SGIs with the same ID that target multiple processors.

  In the Distributor, an SGI is identified uniquely by the combination of its interrupt number, ID0-ID15, the target processor ID, CPUID0-CPUID7, and the *processor source ID*, CPUID0-CPUID7, of the processor that issued the SGI. When the CPU interface communicates the interrupt ID to a targeted processor, it also provides the processor source ID, so that the targeted processor can uniquely identify the SGI.

  SGI banking means the GIC can handle multiple SGIs simultaneously, without resource conflicts.

  The Distributor ignores any write to the GICD_SGIR that is not from a processor that is connected to one of the CPU interfaces. How the Distributor determines the processor source ID of a processor writing to the GICD_SGIR is IMPLEMENTATION SPECIFIC.

In a uniprocessor system, there is no distinction between shared and private interrupts, because all interrupts are visible to the processor. In this case the processor source ID value is 0.

GIC Distributor

GIC Distributor Interface Init

```
        /* GIC Distributor Interface Init */
        mrs     x4, mpidr_el1
        and     x4, x4, #0xffff         /* CPU number */
__gic_dist_init:
        ldr     x0, __gic_dist_base     /* Dist GIC base */
        mov     x1, #0                  /* non-0 cpus should at least */

        cmp     x4, xzr                 /* program IGROUP0 */
        bne     1f
        mov     x1, #3                  /* Enable group0 & group1 */
        str     w1, [x0, #0x00]         /* Ctrl Register */
        ldr     w1, [x0, #0x04]         /* Type Register */
1:      and     x1, x1, #0x1f           /* No. of IGROUPn registers */
        add     x2, x0, #0x080          /* IGROUP0 Register */
        movn    x3, #0                  /* All interrupts to group-1 */
2:      str     w3, [x2], #4
        subs    x1, x1, #1
        bge     2b
...
__gic_dist_base:
        .dword GIC_DIST_BASE
```

Table 4-1 Distributor register map

Offset	Name	Type	Reset[a]	Description
0x000	GICD_CTLR	RW	0x00000000	Distributor Control Register
0x004	GICD_TYPER	RO	IMPLEMENTATION DEFINED	Interrupt Controller Type Register
0x008	GICD_IIDR	RO	IMPLEMENTATION DEFINED	Distributor Implementer Identification Register
0x00C-0x01C	-	-	-	Reserved
0x020-0x03C	-	-	-	IMPLEMENTATION DEFINED registers
0x040-0x07C	-	-	-	Reserved
0x080	GICD_IGROUPRn[b]	RW	IMPLEMENTATION DEFINED[c]	Interrupt Group Registers
0x084-0x0FC			0x00000000	
0x100-0x17C	GICD_ISENABLERn	RW	IMPLEMENTATION DEFINED	Interrupt Set-Enable Registers
0x180-0x1FC	GICD_ICENABLERn	RW	IMPLEMENTATION DEFINED	Interrupt Clear-Enable Registers
0x200-0x27C	GICD_ISPENDRn	RW	0x00000000	Interrupt Set-Pending Registers
0x280-0x2FC	GICD_ICPENDRn	RW	0x00000000	Interrupt Clear-Pending Registers
0x300-0x37C	GICD_ISACTIVERn[d]	RW	0x00000000	GICv2 Interrupt Set-Active Registers
0x380-0x3FC	GICD_ICACTIVERn[e]	RW	0x00000000	Interrupt Clear-Active Registers
0x400-0x7F8	GICD_IPRIORITYRn	RW	0x00000000	Interrupt Priority Registers
0x7FC	-	-	-	Reserved
0x800-0x81C	GICD_ITARGETSRn	RO[f]	IMPLEMENTATION DEFINED	Interrupt Processor Targets Registers
0x820-0xBF8		RW[f]	0x00000000	
0xBFC	-	-	-	Reserved
0xC00-0xCFC	GICD_ICFGRn	RW	IMPLEMENTATION DEFINED	Interrupt Configuration Registers
0xD00-0xDFC	-	-	-	IMPLEMENTATION DEFINED registers
0xE00-0xEFC	GICD_NSACRn[e]	RW	0x00000000	Non-secure Access Control Registers, optional
0xF00	GICD_SGIR	WO	-	Software Generated Interrupt Register
0xF04-0xF0C	-	-	-	Reserved
0xF10-0xF1C	GICD_CPENDSGIRn[e]	RW	0x00000000	SGI Clear-Pending Registers
0xF20-0xF2C	GICD_SPENDSGIRn[e]	RW	0x00000000	SGI Set-Pending Registers
0xF30-0xFCC	-	-	-	Reserved
0xFD0-0xFFC	-	RO	IMPLEMENTATION DEFINED	*Identification registers* on page 4-119

a. For details of any restrictions that apply to the reset values of IMPLEMENTATION DEFINED cases see the appropriate register description.
b. In a GICv1 implementation, present only if the GIC implements the GIC Security Extensions, otherwise RAZ/WI.
c. For more information see *GICD_IGROUPR0 reset value* on page 4-92.
d. In GICv1, these are the Active Bit Registers, ICDABRn. These registers are RO.

GIC Distributor register map

GIC Distributor Interface Init

```
        /* GIC Distributor Interface Init */
        mrs     x4, mpidr_el1
        and     x4, x4, #0xffff         /* CPU number */
__gic_dist_init:
        ldr     x0, __gic_dist_base     /* Dist GIC base */
        mov     x1, #0                  /* non-0 cpus should at least */

        cmp     x4, xzr                 /* program IGROUP0 */
        bne     1f
        mov     x1, #3                  /* Enable group0 & group1 */
        str     w1, [x0, #0x00]         /* Ctrl Register */
        ldr     w1, [x0, #0x04]         /* Type Register */
1:      and     x1, x1, #0x1f           /* No. of IGROUPn registers */
        add     x2, x0, #0x080          /* IGROUP0 Register */
        movn    x3, #0                  /* All interrupts to group-1 */
2:      str     w3, [x2], #4
        subs    x1, x1, #1
        bge     2b
...
__gic_dist_base:
        .dword GIC_DIST_BASE
```

The GIC includes *interrupt grouping* functionality that supports:

- configuring each interrupt as either Group 0 or Group 1
- signaling Group 0 interrupts to the target processor using either the IRQ or the FIQ exception request
- signaling Group 1 interrupts to the target processor using the IRQ exception request only
- a unified scheme for handling the priority of Group 0 and Group 1 interrupts
- optional lockdown of the configuration of some Group 0 interrupts.

Note

- Interrupt grouping is present in all GICv2 implementations and in GICv1 implementations that include the GIC Security Extensions, see *Changes in version 2.0 of the Specification* on page 1-15.

4.3.1 Distributor Control Register, GICD_CTLR

The GICD_CTLR characteristics are:

Purpose Enables the forwarding of pending interrupts from the Distributor to the CPU interfaces.

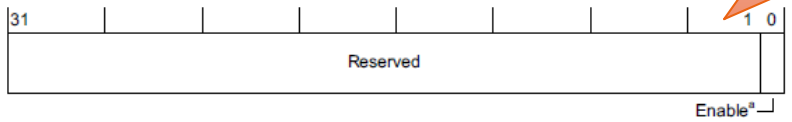
Usage constraints If the GIC implements the Security Extensions with configuration lockdown, the system can lock down the Secure GICD_CTLR, see *Configuration lockdown* on page 4-82.

Configurations This register is available in all configurations of the GIC. If the GIC implements the Security Extensions, this register is banked, see *Register banking* on page 4-77.

Attributes See the register summary in Table 4-1 on page 4-75.

Figure 4-1 and Table 4-4 show the GICD_CTLR bit assignments for:

- a GICv1 implementation that does not include the GIC Security Extensions
- the Non-secure copy of the register in an implementation that includes the GIC Security Extensions.



[a] Bit name is IMPLEMENTATION DEFINED in an implementation that includes the Security Extensions

Figure 4-1 GICD_CTLR bit assignments, GICv1 without Security Extensions or Non-secure

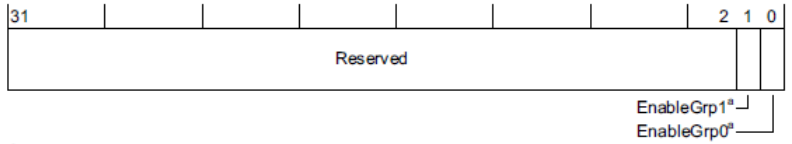
Table 4-4 GICD_CTLR bit assignments, GICv1 without Security Extensions or Non-secure

Bits	Name	Function
[31:1]	-	Reserved.
[0]	Enable[a]	Global enable for forwarding pending interrupts from the Distributor to the CPU interfaces. In the Non-secure copy of this register in an implementation that includes the Security Extensions, this bit controls only the forwarding of Group 1 interrupts: **0** interrupts not forwarded. **1** interrupts forwarded, subject to the priority rules. See *Enabling and disabling the Distributor and CPU interfaces* on page 4-77 for more information about this bit.

a. Bit name is IMPLEMENTATION DEFINED in a GICv1 implementation that includes the Security Extensions.

Figure 4-2 and Table 4-5 on page 4-86 shows the GICD_CTLR bit assignments for:

- Any GICv2 implementation. If the implementation includes the Security Extensions then these assignments apply only to the Secure copy of the register.
- The Secure copy of the register in a GICv1 implementation that includes the Security Extensions.



[a] In a GICv1 implementation that includes the Security Extensions:
- Bit[0] is named Enable
- Bit[1] is IMPLEMENTATION DEFINED.

Figure 4-2 GICD_CTLR bit assignments, GICv2, and GICv1 Secure copy

GIC Distributor Control Register

GIC Distributor Interface Init

```
        /* GIC Distributor Interface Init */
        mrs     x4, mpidr_el1
        and     x4, x4, #0xffff         /* CPU number */
__gic_dist_init:
        ldr     x0, __gic_dist_base     /* Dist GIC base */
        mov     x1, #0                  /* non-0 cpus should at least */

        cmp     x4, xzr                 /* program IGROUP0 */
        bne     1f
        mov     x1, #3                  /* Enable group0 & group1 */
        str     w1, [x0, #0x00]         /* Ctrl Register */
        ldr     w1, [x0, #0x04]         /* Type Register */
1:      and     x1, x1, #0x1f           /* No. of IGROUPn registers */
        add     x2, x0, #0x080          /* IGROUP0 Register */
        movn    x3, #0                  /* All interrupts to group-1 */
2:      str     w3, [x2], #4
        subs    x1, x1, #1
        bge     2b
...
__gic_dist_base:
        .dword GIC_DIST_BASE
```

Peripheral interrupt

An interrupt generated by the assertion of an interrupt request signal input to the GIC. The GIC architecture defines the following types of peripheral interrupt:

Private Peripheral Interrupt (PPI)

A peripheral interrupt that is specific to a single processor.

Shared Peripheral Interrupt (SPI)

A peripheral interrupt that the Distributor can route to a combination of processors, as specified by the corresponding GICD_ITARGETSRn register.

Table 4-5 GICD_CTLR bit assignments, GICv2, and GICv1 Secure copy

Bits	Name	Function
[31:2]	-	Reserved.
[1]	EnableGrp1	Global enable for forwarding pending Group 1 interrupts from the Distributor to the CPU interfaces: **0** Group 1 interrupts not forwarded. **1** Group 1 interrupts forwarded, subject to the priority rules. **Note** In a GICv1 implementation that includes the Security Extensions: • Whether this bit is implemented, and the bit name if implemented, is IMPLEMENTATION DEFINED. If not implemented the bit is reserved. • When the bit is implemented, it is an alias of bit[0] of the Non-secure copy of the register.
[0]	EnableGrp0	Global enable for forwarding pending Group 0 interrupts from the Distributor to the CPU interfaces: **0** Group 0 interrupts not forwarded. **1** Group 0 interrupts forwarded, subject to the priority rules.

When any of the Distributor global enable bits are set to 0, disabling the Distributor functions, other GIC register read and writes still operate normally. This means software can change the state of PPIs and SPIs before re-enabling the Distributor. For example, software can:

- Make an interrupt pending by writing to the corresponding GICD_ISPENDRn.
- Remove the active state from an interrupt by writing to the corresponding GICC_EOIR or GICC_AEOIR.

Note

Setting a Distributor global enable bit to 0 disables forwarding of interrupts to the CPU interfaces. In addition:

- When forwarding of pending interrupts is disabled for Group 0 or Group 1 interrupts, it is IMPLEMENTATION DEFINED whether an edge-triggered interrupt signal sets an edge-triggered interrupt in a disabled group to the pending state.
- In GICv2, software can manage SGI pending state using the Interrupt Set-Pending Register, GICD_ISPENDRn and Interrupt Clear-Pending Register, GICD_ICPENDRn. However, in GICv1, the GIC clears the pending state of an SGI only when the SGI becomes active, and therefore software cannot clear the pending state of an SGI.
- In GICv2, software can manage the active state using the Interrupt Set-Active Registers, GICD_ISACTIVERn and the Interrupt Clear-Active Registers, GICD_ICACTIVERn.

If the forwarding of only one group of interrupts is disabled, and the highest priority pending interrupt is in the disabled group:

- In GICv1, it is IMPLEMENTATION DEFINED whether the Distributor forwards any pending interrupts of Sufficient priority from the other group, to the CPU interfaces.
- In GICv2, the Distributor does not forward any interrupts, from either group, to the CPU interfaces.

When the GICD_CTLR.{EnableGrp1, EnableGrp0} settings mean the Distributor does not forward any pending interrupts to the CPU interfaces, a read of a GICC_IAR or GICC_AIAR register returns a spurious interrupt ID.

GIC Distributor Control Register

GIC Distributor Interface Init

```
        /* GIC Distributor Interface Init */
        mrs     x4, mpidr_el1
        and     x4, x4, #0xffff         /* CPU number */
__gic_dist_init:
        ldr     x0, __gic_dist_base     /* Dist GIC base */
        mov     x1, #0                  /* non-0 cpus should at least */

        cmp     x4, xzr                 /* program IGROUP0 */
        bne     1f
        mov     x1, #3                  /* Enable group0 & group1 */
        str     w1, [x0, #0x00]         /* Ctrl Register */
        ldr     w1, [x0, #0x04]         /* Type Register */
1:      and     x1, x1, #0x1f           /* No. of IGROUPn registers */
        add     x2, x0, #0x080          /* IGROUP0 Register */
        movn    x3, #0                  /* All interrupts to group-1 */
2:      str     w3, [x2], #4
        subs    x1, x1, #1
        bge     2b
…
__gic_dist_base:
        .dword GIC_DIST_BASE
```

Interrupt Controller Type Register

4.3.2 Interrupt Controller Type Register, GICD_TYPER

The GICD_TYPER characteristics are:

Purpose Provides information about the configuration of the GIC. It indicates:

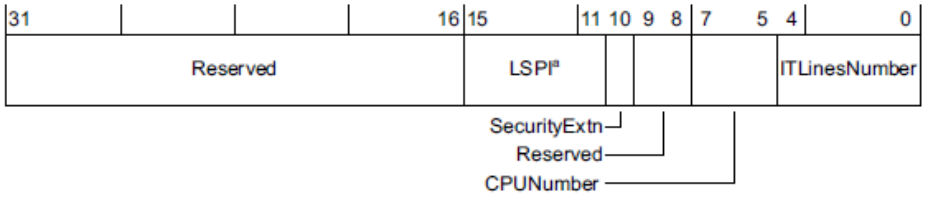
- whether the GIC implements the Security Extensions
- the maximum number of interrupt IDs that the GIC supports
- the number of CPU interfaces implemented
- if the GIC implements the Security Extensions, the maximum number of implemented *Lockable Shared Peripheral Interrupts* (LSPIs).

Usage constraints No usage constraints.

Configurations This register is available in all configurations of the GIC. If the GIC implements the Security Extensions this register is Common.

Attributes See the register summary in Table 4-1 on page 4-75.

Figure 4-3 shows the GICD_TYPER bit assignments.



a Implemented only if the GIC implements the Security Extensions, Reserved otherwise

Figure 4-3 GICD_TYPER bit assignments

Table 4-6 shows the GICD_TYPER bit assignments.

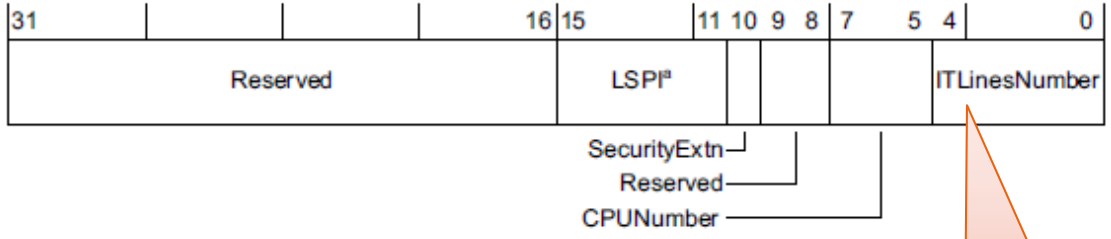
Table 4-6 GICD_TYPER bit assignments

Bits	Name	Function
[31:16]	-	Reserved.
[15:11]	LSPI	If the GIC implements the Security Extensions, the value of this field is the maximum number of implemented lockable SPIs, from 0 (0b00000) to 31 (0b11111), see *Configuration lockdown* on page 4-82. If this field is 0b00000 then the GIC does not implement configuration lockdown. If the GIC does not implement the Security Extensions, this field is reserved.
[10]	SecurityExtn	Indicates whether the GIC implements the Security Extensions. 0 Security Extensions not implemented. 1 Security Extensions implemented.

GIC Distributor Interface Init

```
        /* GIC Distributor Interface Init */
        mrs     x4, mpidr_el1
        and     x4, x4, #0xffff         /* CPU number */
__gic_dist_init:
        ldr     x0, __gic_dist_base     /* Dist GIC base */
        mov     x1, #0                  /* non-0 cpus should at least */

        cmp     x4, xzr                 /* program IGROUP0 */
        bne     1f
        mov     x1, #3                  /* Enable group0 & group1 */
        str     w1, [x0, #0x00]         /* Ctrl Register */
        ldr     w1, [x0, #0x04]         /* Type Register */
1:      and     x1, x1, #0x1f           /* No. of IGROUPn registers */
        add     x2, x0, #0x080          /* IGROUP0 Register */
        movn    x3, #0                  /* All interrupts to group-1 */
2:      str     w3, [x2], #4
        subs    x1, x1, #1
        bge     2b
...
__gic_dist_base:
        .dword GIC_DIST_BASE
```

Figure 4-3 shows the GICD_TYPER bit assignments.



a Implemented only if the GIC implements the Security Extensions, Reserved otherwise

Figure 4-3 GICD_TYPER bit as nts

Table 4-6 GICD_TYPER bit assignmen ed)

Bits	Name	Function
[9:8]	-	Reserved.
[7:5]	CPUNumber	Indicates the number of implemented CPU interfaces. The number of implemented CPU interfaces is one more than the value of this field, for example if this field is 0b011, there are four CPU interfaces. If the GIC implements the Virtualization Extensions, this is also the number of virtual CPU interfaces.
[4:0]	ITLinesNumber	Indicates the maximum number of interrupts that the GIC supports. If ITLinesNumber=N, the maximum number of interrupts is 32(N+1). The interrupt ID range is from 0 to (number of IDs – 1). For example: 0b00011 Up to 128 interrupt lines, interrupt IDs 0-127. The maximum number of interrupts is 1020 (0b11111). See the text in this section for more information. Regardless of the range of interrupt IDs defined by this field, interrupt IDs 1020-1023 are reserved for special purposes, see *Special interrupt numbers* on page 3-43 and *Interrupt IDs* on page 2-24.

The ITLinesNumber field only indicates the maximum number of SPIs that the GIC might support. This value determines the number of implemented interrupt registers, that is, the number of instances of the following registers:

- GICD_IGROUPRn
- GICD_ISENABLERn
- GICD_ICENABLERn
- GICD_ISPENDRn
- GICD_ICPENDRn
- GICD_ISACTIVERn
- GICD_IPRIORITYRn
- GICD_ITARGETSRn
- GICD_ICFGRn.

The GIC architecture does not require a GIC to support a continuous range of SPI interrupt IDs, and the supported SPI interrupt ID range is likely to be non-continuous. Software must check which SPI interrupt IDs are supported, up to the maximum value indicated by the ITLinesNumber field, see *Identifying the supported interrupts* on page 3-35.

Interrupt Controller Type Register

GIC Distributor Interface Init

```
        /* GIC Distributor Interface Init */
        mrs     x4, mpidr_el1
        and     x4, x4, #0xffff         /* CPU number */
__gic_dist_init:
        ldr     x0, __gic_dist_base     /* Dist GIC base */
        mov     x1, #0                  /* non-0 cpus should at least */

        cmp     x4, xzr                 /* program IGROUP0 */
        bne     1f
        mov     x1, #3                  /* Enable group0 & group1 */
        str     w1, [x0, #0x00]         /* Ctrl Register */
        ldr     w1, [x0, #0x04]         /* Type Register */
1:      and     x1, x1, #0x1f           /* No. of IGROUPn registers */
        add     x2, x0, #0x080          /* IGROUP0 Register */
        movn    x3, #0                  /* All interrupts to group-1 */
2:      str     w3, [x2], #4
        subs    x1, x1, #1
        bge     2b
...
__gic_dist_base:
        .dword GIC_DIST_BASE
```

Interrupt Group Register

4.3.4 Interrupt Group Registers, GICD_IGROUPRn

The GICD_IGROUPR characteristics are:

Purpose

The GICD_IGROUPR registers provide a status bit for each interrupt supported by the GIC. Each bit controls whether the corresponding interrupt is in Group 0 or Group 1.

Usage constraints

In implementations that include the GIC Security Extensions, accessible by Secure accesses only. The register addresses are RAZ/WI to Non-secure accesses.

A register bit corresponding to an unimplemented interrupt is RAZ/WI.

If the GIC implements configuration lockdown, the system can lockdown the group status bits for lockable SPIs that are configured as Group 0, see *Configuration lockdown* on page 4-82.

Configurations

In GICv1, only implemented if the GIC implements the Security Extensions. If a GICv1 implementation does not include the Security Extensions the GICD_IGROUPR addresses are RAZ/WI.

Note

Typically, when used with a processor that implements the ARM Security Extensions, Group 0 interrupts are Secure interrupts, and Group 1 interrupts are Non-secure interrupts, see *Security Extensions support* on page 1-16 for more information.

In GICv2, these registers are always implemented.

The number of implemented GICD_IGROUPR registers is (GICD_TYPER.ITLinesNumber + 1). The implemented GICD_IGROUPR registers number upwards from GICD_IGROUPR0. If the GIC implements the Security Extensions, these are Secure registers.

In a multiprocessor implementation, GICD_IGROUPR0 is banked for each connected processor. This register holds the group status bits for interrupts 0-31.

Attributes

See the register summary in Table 4-1 on page 4-75, and *GICD_IGROUPR0 reset value* on page 4-92.

Figure 4-5 shows the GICD_IGROUPR bit assignments.

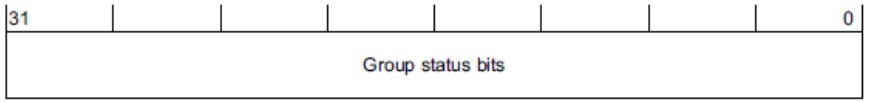


Figure 4-5 GICD_IGROUPR bit assignments

Table 4-8 shows the GICD_IGROUPR bit assignments.

Table 4-8 GICD_IGROUPR bit assignments

Bits	Name	Function	
[31:0]	Group status bits	For each bit:	
		0	The corresponding interrupt is Group 0.
		1	The corresponding interrupt is Group 1.

GIC Distributor Interface Init

```
        /* GIC Distributor Interface Init */
        mrs     x4, mpidr_el1
        and     x4, x4, #0xffff         /* CPU number */
__gic_dist_init:
        ldr     x0, __gic_dist_base     /* Dist GIC base */
        mov     x1, #0                  /* non-0 cpus should at least */

        cmp     x4, xzr                 /* program IGROUP0 */
        bne     1f
        mov     x1, #3                  /* Enable group0 & group1 */
        str     w1, [x0, #0x00]         /* Ctrl Register */
        ldr     w1, [x0, #0x04]         /* Type Register */
1:      and     x1, x1, #0x1f           /* No. of IGROUPn registers */
        add     x2, x0, #0x080          /* IGROUP0 Register */
        movn    x3, #0                  /* All interrupts to group-1 */
2:      str     w3, [x2], #4
        subs    x1, x1, #1
        bge     2b
...
__gic_dist_base:
        .dword GIC_DIST_BASE
```

Note

On start-up or reset, each interrupt with ID32 or higher resets as Group 0 and therefore all SPIs are Group 0 unless the system reprograms the appropriate GICD_IGROUPR bit. See *GICD_IGROUPR0 reset value* for information about the reset configuration of interrupts with IDs 0-31.

For interrupt ID *m*, when DIV and MOD are the integer division and modulo operations:

- the corresponding GICD_IGROUPRn number, *n*, is given by *n* = *m* DIV 32
- the offset of the required GICD_IGROUPR is (0x080 + (4*n))
- the bit number of the required group status bit in this register is *m* MOD 32

Interrupt Group Register

GICD_IGROUPR0 reset value

Typically, the reset value of all GICD_IGROUPR registers is zero, so that all interrupts are Group 0 unless reprogrammed as Group 1 by Secure accesses to the appropriate GICD_IGROUPR registers.

For GICv2 implementations that do not include the Security Extensions, the GICD_IGROUPRn reset values are IMPLEMENTATION DEFINED.

A multiprocessor implementation that supports the Security Extensions might include one or more *Non-secure processors*, meaning processors that cannot make Secure accesses to the GIC. In this situation only, a GIC can implement a Secure IMPLEMENTATION DEFINED mechanism that resets to 1 the GICD_IGROUPR0 bits for the peripheral interrupts and SGIs of any Non-secure processor. This mechanism must apply only to:

- a banked GICD_IGROUPR0 that corresponds to a Non-secure processor
- bits in that banked GICD_IGROUPR0 that correspond to implemented interrupts.

GIC Secured CPU Interface Init

```
__gic_cpu_init:
        /* GIC Secured CPU Interface Init */
        ldr     x0, __gic_cpu_base  /* GIC CPU base */
        mov     x1, #0x80
        str     w1, [x0, #0x4]      /* GIC CPU Priority Mask */
        mov     x1, #0x3            /* Enable group0 & group1 */
        str     w1, [x0]            /* GIC CPU Control */
        ...
__gic_cpu_base:
        .dword GIC_CPU_BASE
```

GIC CPU interface register map

4.1.3 CPU interface register map

Table 4-2 shows the CPU interface register map. Address offsets are relative to the *CPU interface base address* defined by the system memory map. All GIC registers are 32-bits wide. Reserved register addresses are RAZ/WI.

For a multiprocessor implementation, the GIC implements a set of CPU interface registers for each CPU interface. ARM strongly recommends that each processor has the same CPU interface base address for the CPU interface that connects it to the GIC. This is the private CPU interface base address for that processor. It is IMPLEMENTATION DEFINED whether a processor can access the CPU interface registers of other processors in the system.

Note

For more information about:

- the registers added by the GIC Virtualization Extensions, see Chapter 5 *GIC Support for Virtualization*
- legacy register names, see Appendix B *Register Names*.

Table 4-2 CPU interface register map

Offset	Name	Type	Reset	Description
0x0000	GICC_CTLR	RW	0x00000000	CPU Interface Control Register
0x0004	GICC_PMR	RW	0x00000000	Interrupt Priority Mask Register
0x0008	GICC_BPR	RW	0x0000000x[a]	Binary Point Register
0x000C	GICC_IAR	RO	0x000003FF	Interrupt Acknowledge Register
0x0010	GICC_EOIR	WO	-	End of Interrupt Register
0x0014	GICC_RPR	RO	0x000000FF	Running Priority Register
0x0018	GICC_HPPIR	RO	0x000003FF	Highest Priority Pending Interrupt Register
0x001C	GICC_ABPR[b]	RW	0x0000000x[a]	Aliased Binary Point Register
0x0020	GICC_AIAR[c]	RO	0x000003FF	Aliased Interrupt Acknowledge Register
0x0024	GICC_AEOIR[c]	WO	-	Aliased End of Interrupt Register
0x0028	GICC_AHPPIR[c]	RO	0x000003FF	Aliased Highest Priority Pending Interrupt Register
0x002C-0x003C	-	-	-	Reserved
0x0040-0x00CF	-	-	-	IMPLEMENTATION DEFINED registers
0x00D0-0x00DC	GICC_APRn[c]	RW	0x00000000	Active Priorities Registers
0x00E0-0x00EC	GICC_NSAPRn[c]	RW	0x00000000	Non-secure Active Priorities Registers
0x00ED-0x00F8	-	-	-	Reserved
0x00FC	GICC_IIDR	RO	IMPLEMENTATION DEFINED	CPU Interface Identification Register
0x1000	GICC_DIR[c]	WO	-	Deactivate Interrupt Register

a. See the register description for more information.
b. Present in GICv1 if the GIC implements the GIC Security Extensions. Always present in GICv2.
c. GICv2 only.

GIC Secured CPU Interface Init

```
__gic_cpu_init:
        /* GIC Secured CPU Interface Init */
        ldr     x0, __gic_cpu_base /* GIC CPU base */
        mov     x1, #0x80
        str     w1, [x0, #0x4]          /* GIC CPU Priority Mask */
        mov     x1, #0x3                /* Enable group0 & group1 */
        str     w1, [x0]                /* GIC CPU Control */
        ...
__gic_cpu_base:
        .dword GIC_CPU_BASE
```

GIC CPU Interface Control Register

4.4.1 CPU Interface Control Register, GICC_CTLR

The GICC_CTLR characteristics are:

Purpose Enables the signaling of interrupts by the CPU interface to the connected processor, and provides additional top-level control of the CPU interface. In a GICv2 implementation, this includes control of the *end of interrupt* (EOI) behavior.

Note

In a GICv2 implementation that includes the GIC Security Extensions, independent EOI controls are provided for:

- Accesses from Secure state. This control applies to the handling of both Group 0 and Group 1 interrupts.
- Accesses from Non-secure state. This control only applies to the handling of Group 1 interrupts.

The EOI controls affect the behavior of accesses to GICC_EOIR and GICC_DIR. See the register descriptions for more information.

Usage constraints If the GIC implements the Security Extensions with support for configuration lockdown, the system can prevent write access to certain register fields in the Secure GICC_CTLR, see *Configuration lockdown* on page 4-82.

Configurations If the implementation supports interrupt grouping, this register provides independent control of Group 0 and Group 1 interrupts.

If the GIC implements the Security Extensions:

- this register is banked to provide Secure and Non-secure copies, see *Register banking* on page 4-77
- the register bit assignments are different in the Secure and Non-secure copies of the register, and:
  - the Secure copy of the register can control both Group 0 and Group 1 interrupts
  - the Non-secure copy of the register can control only Group 1 interrupts.

Attributes See the register summary in Table 4-2 on page 4-76.

Figure 4-22 on page 4-126 and Table 4-29 on page 4-126 shows the GICC_CTLR bit assignments for a GICv1 implementation, for

- an implementation that does not include the Security Extensions
- the Non-secure copy of the register, in an implementation that includes the Security Extensions.

GIC Secured CPU Interface Init

```
__gic_cpu_init:
        /* GIC Secured CPU Interface Init */
        ldr     x0, __gic_cpu_base /* GIC CPU base */
        mov     x1, #0x80
        str     w1, [x0, #0x4]          /* GIC CPU Priority Mask */
        mov     x1, #0x3                /* Enable group0 & group1 */
        str     w1, [x0]                /* GIC CPU Control */
        ...
__gic_cpu_base:
        .dword GIC_CPU_BASE
```

GIC CPU Interface Control Register

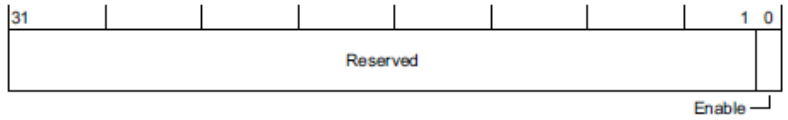


Figure 4-22 GICC_CTLR bit assignments, GICv1 without Security Extensions or Non-secure

Table 4-29 GICC_CTLR bit assignments, GIC1 without Security Extensions or Non-secure

Bits	Name	Function
[31:1]	-	Reserved
[0]	Enable	Enable for the signaling of Group 1 interrupts by the CPU interface to the connected processor. 0 Disable signaling of interrupts 1 Enable signaling of interrupts. **Note** • When this bit is cleared to 0, the CPU interface ignores any pending interrupt forwarded to it. When this bit is set to 1, the CPU interface starts to process pending interrupts that are forwarded to it. There is a small but finite time required for a change to take effect. • On a GICv1 implementation that does not include the Security Extensions, this bit controls the signaling of all interrupts by the CPU interface to the connected processor. See *Enabling and disabling the Distributor and CPU interfaces* on page 4-77 for more information about this bit.

Figure 4-23 and Table 4-30 show the GICC_CTLR bit assignments for the Non-secure copy of the register in a GIC v2 implementation that includes the Security Extensions,

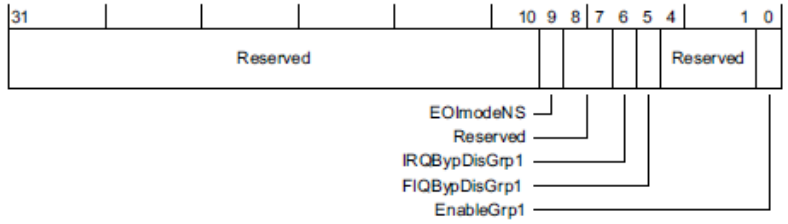


Figure 4-23 GICC_CTLR bit assignments, GICv2 with Security Extensions, Non-secure copy

Table 4-30 GICC_CTLR bit assignments, GIC2 with Security Extensions, Non-secure copy

Bits	Name	Function
[31:10]	-	Reserved
[9]	EOImodeNS	Controls the behavior of Non-secure accesses to the GICC_EOIR and GICC_DIR registers: 0 GICC_EOIR has both priority drop and deactivate interrupt functionality. Accesses to the GICC_DIR are UNPREDICTABLE. 1 GICC_EOIR has priority drop functionality only. The GICC_DIR register has deactivate interrupt functionality. See *Behavior of writes to GICC_EOIR, GICv2* on page 4-140 for more information.
[8:7]	-	Reserved.

GIC Secured CPU Interface Init

```
__gic_cpu_init:
        /* GIC Secured CPU Interface Init */
        ldr     x0, __gic_cpu_base  /* GIC CPU base */
        mov     x1, #0x80
        str     w1, [x0, #0x4]      /* GIC CPU Priority Mask */
        mov     x1, #0x3            /* Enable group0 & group1 */
        str     w1, [x0]            /* GIC CPU Control */
        ...
__gic_cpu_base:
        .dword GIC_CPU_BASE
```

GIC CPU Interface Control Register

Table 4-30 GICC_CTLR bit assignments, GIC2 with Security Extensions, Non-secure copy (continued)

Bits	Name	Function
[6]	IRQBypDisGrp1	When the signaling of IRQs by the CPU interface is disabled, this bit partly controls whether the bypass IRQ signal is signaled to the processor: 0 Bypass IRQ signal is signaled to the processor 1 Bypass IRQ signal is not signaled to the processor. See *Interrupt signal bypass, and GICv2 bypass disable* on page 2-27 for more information.
[5]	FIQBypDisGrp1	When the signaling of FIQs by the CPU interface is disabled, this bit partly controls whether the bypass FIQ signal is signaled to the processor: 0 Bypass FIQ signal is signaled to the processor 1 Bypass FIQ signal is not signaled to the processor. See *Interrupt signal bypass, and GICv2 bypass disable* on page 2-27 for more information.
[4:1]	-	Reserved
[0]	EnableGrp1	Enable for the signaling of Group 1 interrupts by the CPU interface to the connected processor. 0 Disable signaling of interrupts 1 Enable signaling of interrupts. **Note** When this bit is set to 0, the CPU interface ignores any pending Group 1 interrupt forwarded to it. When this bit is set to 1, the CPU interface starts to process pending Group 1 interrupts that are forwarded to it. There is a small but finite time required for a change to take effect. See *Enabling and disabling the Distributor and CPU interfaces* on page 4-77 for more information about this bit.

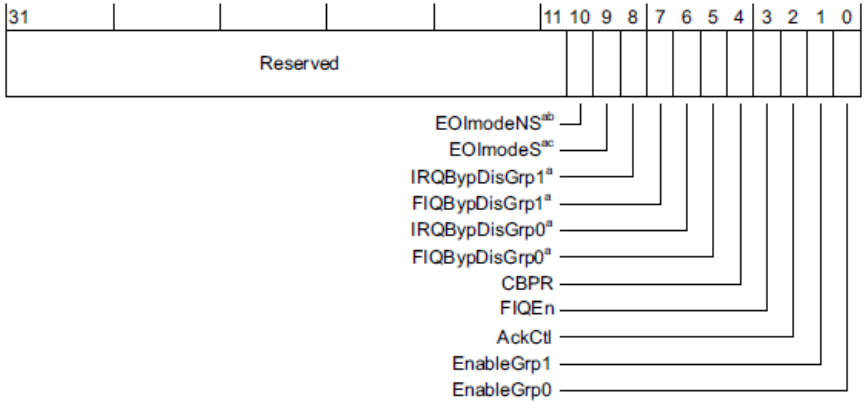
Figure 4-24 on page 4-128 and Table 4-31 on page 4-128 show the GICC_CTLR bit assignments for:

- a GICv2 implementation, for:
  - an implementation that does not include the Security Extensions
  - the Secure copy of the register, in an implementation that includes the Security Extensions
- a GICv1 implementation that includes the Security Extensions, for the Secure copy of the register.

GIC Secured CPU Interface Init

```
__gic_cpu_init:
        /* GIC Secured CPU Interface Init */
        ldr     x0, __gic_cpu_base  /* GIC CPU base */
        mov     x1, #0x80
        str     w1, [x0, #0x4]      /* GIC CPU Priority Mask */
        mov     x1, #0x3            /* Enable group0 & group1 */
        str     w1, [x0]            /* GIC CPU Control */
        ...
__gic_cpu_base:
        .dword GIC_CPU_BASE
```

GIC CPU Interface Control Register



a GICv2 only
b When the GIC implementation includes the Security Extensions
c EOImode in a GIC implementation that does not include the Security Extensions

Figure 4-24 GICC_CTLR bit assignments, GICv2 without Security Extensions or Secure

Table 4-31 GICC_CTLR bit assignments, GICv2 without Security Extensions or Secure

Bit	Name	Function
[31:11]	-	Reserved.
[10]	EOImodeNS[ab]	Alias of EOImodeNS from the Non-secure copy of this register, see Table 4-30 on page 4-126. In a GICv2 implementation that does not include the Security Extensions, and in a GICv1 implementation, this bit is reserved.
[9]	EOImodeS[a]	Controls the behavior of accesses to GICC_EOIR and GICC_DIR registers. In a GIC implementation that includes the Security Extensions, this control applies only to Secure accesses, and the EOImodeNS bit controls the behavior of Non-secure accesses to these registers: **0** GICC_EOIR has both priority drop and deactivate interrupt functionality. Accesses to the GICC_DIR are UNPREDICTABLE. **1** GICC_EOIR has priority drop functionality only. GICC_DIR has deactivate interrupt functionality. See *Behavior of writes to GICC_EOIR, GICv2* on page 4-140 for more information. **Note** This bit is called EOImode in a GIC implementation that does not include the Security Extensions. In a GICv1 implementation, this bit is reserved.
[8]	IRQBypDisGrp1[a]	Alias of IRQBypDisGrp1 from the Non-secure copy of this register, see Table 4-30 on page 4-126. In a GICv1 implementation, this bit is reserved.
[7]	FIQBypDisGrp1[a]	Alias of FIQBypDisGrp1 from the Non-secure copy of this register, see Table 4-30 on page 4-126. In a GICv1 implementation, this bit is reserved.

GIC Secured CPU Interface Init

```
__gic_cpu_init:
        /* GIC Secured CPU Interface Init */
        ldr     x0, __gic_cpu_base  /* GIC CPU base */
        mov     x1, #0x80
        str     w1, [x0, #0x4]      /* GIC CPU Priority Mask */
        mov     x1, #0x3            /* Enable group0 & group1 */
        str     w1, [x0]            /* GIC CPU Control */
        ...
__gic_cpu_base:
        .dword GIC_CPU_BASE
```

GIC CPU Interface Control Register

Table 4-31 GICC_CTLR bit assignments, GICv2 without Security Extensions or Secure (continued)

Bit	Name	Function
[6]	IRQBypDisGrp0[a]	When the signaling of IRQs by the CPU interface is disabled, this bit partly controls whether the bypass IRQ signal is signaled to the processor: **0** Bypass IRQ signal is signaled to the processor **1** Bypass IRQ signal is not signaled to the processor. See *Interrupt signal bypass, and GICv2 bypass disable* on page 2-27 and *Power management, GIC v2* on page 2-31 for more information. In a GICv1 implementation, this bit is reserved.
[5]	FIQBypDisGrp0[a]	When the signaling of FIQs by the CPU interface is disabled, this bit partly controls whether the bypass FIQ signal is signaled to the processor: **0** Bypass FIQ signal is signaled to the processor **1** Bypass FIQ signal is not signaled to the processor. See *Interrupt signal bypass, and GICv2 bypass disable* on page 2-27 and *Power management, GIC v2* on page 2-31 for more information. In a GICv1 implementation, this bit is reserved.
[4]	CBPR[c]	Controls whether the GICC_BPR provides common control to Group 0 and Group 1 interrupts. **0** To determine any preemption, use: • the GICC_BPR for Group 0 interrupts • the GICC_ABPR for Group 1 interrupts. **1** To determine any preemption use the GICC_BPR for both Group 0 and Group 1 interrupts. See *The effect of interrupt grouping on priority grouping* on page 3-57 for more information about how GICC_CTLR.CBPR affects accesses to GICC_BPR and GICC_ABPR.
[3]	FIQEn	Controls whether the CPU interface signals Group 0 interrupts to a target processor using the FIQ or the IRQ signal. **0** Signal Group 0 interrupts using the IRQ signal. **1** Signal Group 0 interrupts using the FIQ signal. The GIC always signals Group 1 interrupts using the IRQ signal. **Note** If using software written for a system that includes an implementation of GICv1 without the Security Extensions, all interrupts are signaled by the IRQ signal. In such systems, ensure that this bit is 0. This is the default value. See *Example GIC usage models* on page 3-68 for more information.

GIC Secured CPU Interface Init

```
__gic_cpu_init:
        /* GIC Secured CPU Interface Init */
        ldr     x0, __gic_cpu_base  /* GIC CPU base */
        mov     x1, #0x80
        str     w1, [x0, #0x4]      /* GIC CPU Priority Mask */
        mov     x1, #0x3            /* Enable group0 & group1 */
        str     w1, [x0]            /* GIC CPU Control */
        ...
__gic_cpu_base:
        .dword GIC_CPU_BASE
```

GIC CPU Interface Control Register

Table 4-31 GICC_CTLR bit assignments, GICv2 without Security Extensions or Secure (continued)

Bit	Name	Function
[2]	AckCtl	When the highest priority pending interrupt is a Group 1 interrupt, determines both: • whether a read of GICC_IAR acknowledges the interrupt, or returns a spurious interrupt ID • whether a read of GICC_HPPIR returns the ID of the highest priority pending interrupt, or returns a spurious interrupt ID. **0** If the highest priority pending interrupt is a Group 1 interrupt, a read of the GICC_IAR or the GICC_HPPIR returns an Interrupt ID of 1022. A read of the GICC_IAR does not acknowledge the interrupt, and has no effect on the pending status of the interrupt. **1** If the highest priority pending interrupt is a Group 1 interrupt, a read of the GICC_IAR or the GICC_HPPIR returns the Interrupt ID of the Group 1 interrupt. A read of GICC_IAR acknowledges and Activates the interrupt. In a GIC implementation that includes the Security Extensions, this control affects only the behavior of Secure register accesses. For more information, see: • *The effect of interrupt grouping on interrupt acknowledgement* on page 3-50 • *Interrupt grouping and interrupt prioritization* on page 3-53 • *Behavior of writes to GICC_EOIR, GICv1 with Security Extensions* on page 4-139 • *Effect of interrupt grouping and the Security Extensions on reads of the GICC_HPPIR* on page 4-143. **Note** ARM deprecates use of GICC_CTLR.AckCtl, and strongly recommends using a software model where GICC_CTLR.AckCtl is set to 0. See *Enabling and disabling the Distributor and CPU interfaces* on page 4-77 for more information about the effects of setting this bit.
[1]	EnableGrp1[d, e]	Enable for the signaling of Group 1 interrupts by the CPU interface to the connected processor: **0** Disable signaling of Group 1 interrupts. **1** Enable signaling of Group 1 interrupts.
[0]	EnableGrp0[e, f]	Enable for the signaling of Group 0 interrupts by the CPU interface to the connected processor: **0** Disable signaling of Group 0 interrupts. **1** Enable signaling of Group 0 interrupts.

a. GICv2 only, see *Power management, GIC v2* on page 2-31 for more information.
b. When the GIC implements the Security Extensions.
c. SBPR in GICv1.
d. EnableNS in GICv1.
e. There is a small but finite time required for a change in the value of this register to take effect.
f. EnableS in GICv1.

For more information about the optional support for interrupt signal bypass, including the GICv2 disable controls of this functionality, see *Interrupt signal bypass, and GICv2 bypass disable* on page 2-27.

GIC Secured CPU Interface Init

```
__gic_cpu_init:
        /* GIC Secured CPU Interface Init */
        ldr     x0, __gic_cpu_base  /* GIC CPU base */
        mov     x1, #0x80
        str     w1, [x0, #0x4]      /* GIC CPU Priority Mask */
        mov     x1, #0x3            /* Enable group0 & group1 */
        str     w1, [x0]            /* GIC CPU Control */
        ...
__gic_cpu_base:
        .dword GIC_CPU_BASE
```

GIC Interrupt Priority Mask Register

4.4.2 Interrupt Priority Mask Register, GICC_PMR

The GICC_PMR characteristics are:

Purpose Provides an interrupt priority filter. Only interrupts with higher priority than the value in this register are signaled to the processor.

Note

Higher priority corresponds to a lower Priority field value.

Usage constraints If the GIC implements the Security Extensions then:

- a Non-secure access to this register can only read or write a value that corresponds to the lower half of the priority range, see *Interrupt grouping and interrupt prioritization* on page 3-53.
- if a Secure write has programmed the GICC_PMR with a value that corresponds to a value in the upper half of the priority range then:
  - any Non-secure read of the GICC_PMR returns 0x00, regardless of the value held in the register
  - any Non-secure write to the GICC_PMR is ignored.

For more information see *Non-secure access to register fields for Group 0 interrupt priorities* on page 4-81.

When determining interrupt preemption, the priority value can be split into two parts, using the Binary Point register, GICC_BPR.

Configurations This register is available in all configurations of the GIC. If the GIC implements the Security Extensions, this register is Common.

Attributes See the register summary in Table 4-2 on page 4-76.

Figure 4-25 shows the GICC_PMR bit assignments.

Figure 4-25 GICC_PMR bit assignments

Table 4-32 shows the GICC_PMR Register bit assignments.

Table 4-32 GICC_PMR Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved.
[7:0]	Priority	The priority mask level for the CPU interface. If the priority of an interrupt is higher than the value indicated by this field, the interface signals the interrupt to the processor. If the GIC supports fewer than 256 priority levels then some bits are RAZ/WI, as follows: **128 supported levels** Bit [0] = 0. **64 supported levels** Bit [1:0] = 0b00. **32 supported levels** Bit [2:0] = 0b000. **16 supported levels** Bit [3:0] = 0b0000. For more information see *Interrupt prioritization* on page 3-44.

GIC Secured CPU Interface Init

```
__gic_cpu_init:
        /* GIC Secured CPU Interface Init */
        ldr     x0, __gic_cpu_base /* GIC CPU base */
        mov     x1, #0x80
        str     w1, [x0, #0x4]          /* GIC CPU Priority Mask */
        mov     x1, #0x3                /* Enable group0 & group1 */
        str     w1, [x0]                /* GIC CPU Control */
        ...
__gic_cpu_base:
        .dword GIC_CPU_BASE
```

GIC Interrupt Priority Mask Register

The following pseudocode shows the effects of the GIC Security Extensions on accesses to this register:

```
// MaskRegRead()
// -------------

bits(8) MaskRegRead()

    read_value = GICC_PMR<7:0>;
    if NS_access then                                   // A non-secure GIC access.
        if read_value<7> == '0' then
            read_value = '00000000';                    // A secure priority value, RAZ
        else
            read_value = LSL((read_value AND P_MASK), 1);
    return(read_value);


// MaskRegWrite()
// --------------

MaskRegWrite(bits(8) value)

    if NS_access then                                   // A non-secure GIC access.
        mod_write_val = ('10000000' OR LSR(value,1)) AND P_MASK;
        if GICC_PMR<7> == '1' then                      // Non-secure execution can only update the
            GICC_PMR[cpu_id]<7:0> = mod_write_val;      // Priority Mask Register if the current
                                                        // value is in the range 0x80 to 0xFF
        else
            IgnoreWriteRequest();
    else                                                // A secure GIC access
        GICC_PMR<7:0> = value AND P_MASK;
```

Clear EL2 control register

```
/* Clear EL2 control register */
msr     sctlr_el2, xzr
```

D7.2.81 SCTLR_EL2, System Control Register (EL2)

The SCTLR_EL2 characteristics are:

Purpose

Provides top level control of the system, including its memory system, at EL2.

This register is part of:

- the Virtualization registers functional group
- the Other system control registers functional group.

Usage constraints

This register is accessible as shown below:

EL0	EL1 (NS)	EL1 (S)	EL2	EL3 (SCR.NS=1)	EL3 (SCR.NS=0)
-	-	-	RW	RW	RW

Configurations

SCTLR_EL2 is architecturally mapped to AArch32 register HSCTLR.

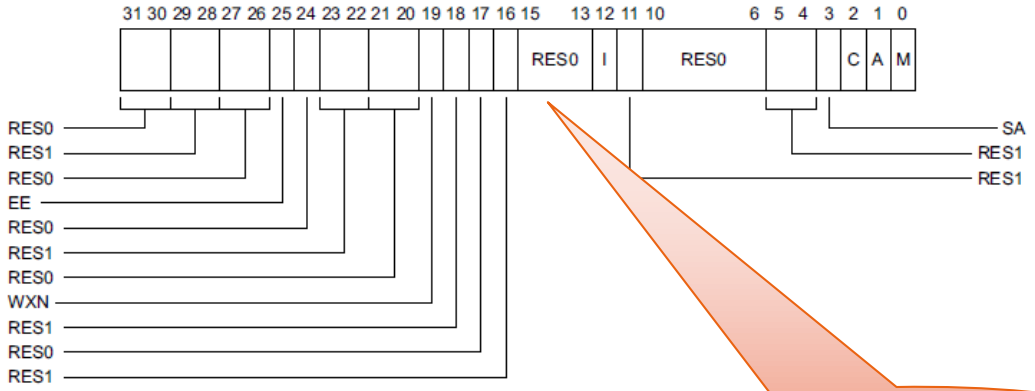
If EL2 is not implemented, this register is RES0 from EL3.

Attributes

SCTLR_EL2 is a 32-bit register.

Field descriptions

The SCTLR_EL2 bit assignments are:



Bits [31:30]

Reserved, RES0.

Bits [29:28]

Reserved, RES1.

Bits [27:26]

Reserved, RES0.

System Control Register

Clear EL2 control register

```
/* Clear EL2 control register */
msr     sctlr_el2, xzr
```

EE, bit [25]

Exception Endianness. This bit controls the endianness for:

- Explicit data accesses at EL2.
- Stage 1 translation table walks at EL2.
- Stage 2 translation table walks at EL1 and EL0.

The possible values of this bit are:

0 Little-endian.

1 Big-endian.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

The EE bit is permitted to be cached in a TLB.

If this register is at the highest exception level implemented, field resets to an IMPLEMENTATION DEFINED value. Otherwise, its reset value is UNKNOWN.

Bit [24]

Reserved, RES0.

Bits [23:22]

Reserved, RES1.

Bits [21:20]

Reserved, RES0.

WXN, bit [19]

Write permission implies XN (Execute Never). This bit can be used to require all memory regions with write permission to be treated as XN. The possible values of this bit are:

0 Regions with write permission are not forced to XN.

1 Regions with write permission are forced to XN.

The WXN bit is permitted to be cached in a TLB.

Reset value is architecturally UNKNOWN.

Bit [18]

Reserved, RES1.

Bit [17]

Reserved, RES0.

Bit [16]

Reserved, RES1.

Bits [15:13]

Reserved, RES0.

I, bit [12]

Instruction cache enable. This is an enable bit for instruction caches at EL2:

0 Instruction caches disabled at EL2. If SCTLR_EL2.M is set to 0, instruction accesses from stage 1 of the EL2 translation regime are to Normal memory, Outer Shareable, Inner Non-cacheable, Outer Non-cacheable.

1 Instruction caches enabled at EL2. If SCTLR_EL2.M is set to 0, instruction accesses from stage 1 of the EL2 translation regime are to Normal memory, Outer Shareable, Inner Write-Through, Outer Write-Through.

System Control Register

Clear EL2 control register

```
/* Clear EL2 control register */

msr     sctlr_el2, xzr
```

Accessing the SCTLR_EL2

To access the SCTLR_EL2:

```
MRS <Xt>, SCTLR_EL2 ; Read SCTLR_EL2 into Xt
MSR SCTLR_EL2, <Xt> ; Write Xt to SCTLR_EL2
```

Register access is encoded as follows:

op0	op1	CRn	CRm	op2
11	100	0001	0000	000

System Control Register

When this bit is 0, all EL2 Normal memory instruction accesses are Non-cacheable. This bit has no effect on the EL1&0 or EL3 translation regimes.

If this register is at the highest exception level implemented, field resets to 0. Otherwise, its reset value is UNKNOWN.

Bit [11]

Reserved, RES1.

Bits [10:6]

Reserved, RES0.

Bits [5:4]

Reserved, RES1.

SA, bit [3]

Stack Alignment Check Enable. When set, use of the stack pointer as the base address in a load/store instruction at this register's exception level must be aligned to a 16-byte boundary, or a Stack Alignment Fault exception will be raised.

Reset value is architecturally UNKNOWN.

C, bit [2]

Cache enable. This is an enable bit for data and unified caches at EL2:

0 Data and unified caches disabled at EL2.

1 Data and unified caches enabled at EL2.

When this bit is 0, all EL2 Normal memory data accesses and all accesses to the EL2 translation tables are Non-cacheable. This bit has no effect on the EL1&0 or EL3 translation regimes.

If this register is at the highest exception level implemented, field resets to 0. Otherwise, its reset value is UNKNOWN.

A, bit [1]

Alignment check enable. This is the enable bit for Alignment fault checking:

0 Alignment fault checking disabled.

Instructions that load or store one or more registers, other than load/store exclusive and load-acquire/store-release, do not check that the address being accessed is aligned to the size of the data element(s) being accessed.

1 Alignment fault checking enabled.

All instructions that load or store one or more registers have an alignment check that the address being accessed is aligned to the size of the data element(s) being accessed. If this check fails it causes an Alignment fault, which is taken as a Data Abort exception.

Load/store exclusive and load-acquire/store-release instructions have an alignment check regardless of the value of the A bit.

Reset value is architecturally UNKNOWN.

M, bit [0]

MMU enable for EL2 stage 1 address translation. Possible values of this bit are:

0 EL2 stage 1 address translation disabled.

1 EL2 stage 1 address translation enabled.

If this register is at the highest exception level implemented, field resets to 0. Otherwise, its reset value is UNKNOWN.

Switch to EL2 from EL3

```
        /* Prepare the switch to EL2 mode from EL3 mode */
        ldr     x0, __start_el2         /* Return after mode switch */
        mov     x1, #0x3c9              /* EL2_SP1 | D | A | I | F */
        msr     elr_el3, x0
        msr     spsr_el3, x1
        eret
...
__start_el2:
        .dword _start_el2
```

Exception Link Register

C5.3.7 ELR_EL3, Exception Link Register (EL3)

The ELR_EL3 characteristics are:

Purpose

When taking an exception to EL3, holds the address to return to.

This register is part of the Special purpose registers functional group.

Usage constraints

This register is accessible as shown below:

EL0	EL1 (NS)	EL1 (S)	EL2	EL3 (SCR.NS=1)	EL3 (SCR.NS=0)
-	-	-	-	RW	RW

Configurations

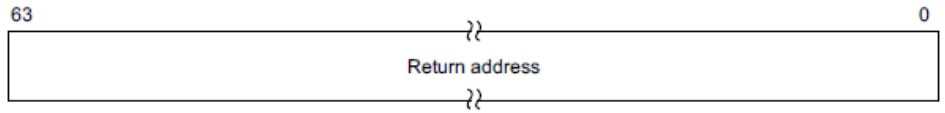
There are no configuration notes.

Attributes

ELR_EL3 is a 64-bit register.

Field descriptions

The ELR_EL3 bit assignments are:



Bits [63:0]

Return address.

Accessing the ELR_EL3

To access the ELR_EL3:

```
MRS <Xt>, ELR_EL3 ; Read ELR_EL3 into Xt
MSR ELR_EL3, <Xt> ; Write Xt to ELR_EL3
```

Register access is encoded as follows:

op0	op1	CRn	CRm	op2
11	110	0100	0000	001

Switch to EL2 from EL3

```
        /* Prepare the switch to EL2 mode from EL3 mode */
        ldr     x0, __start_el2         /* Return after mode switch */
        mov     x1, #0x3c9              /* EL2_SP1 | D | A | I | F */
        msr     elr_el3, x0
        msr     spsr_el3, x1
        eret
...
__start_el2:
        .dword _start_el2
```

Saved Program Status Register

C5.3.19 SPSR_EL3, Saved Program Status Register (EL3)

The SPSR_EL3 characteristics are:

Purpose

Holds the saved processor state when an exception is taken to EL3.

This register is part of the Special purpose registers functional group.

Usage constraints

This register is accessible as shown below:

EL0	EL1 (NS)	EL1 (S)	EL2	EL3 (SCR.NS=1)	EL3 (SCR.NS=0)
-	-	-	-	RW	RW

Configurations

SPSR_EL3 can be mapped to AArch32 register SPSR_mon, but this is not architecturally mandated.

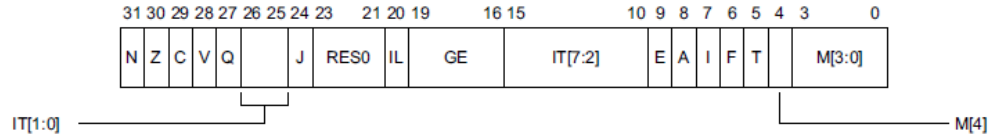
Attributes

SPSR_EL3 is a 32-bit register.

Field descriptions

The SPSR_EL3 bit assignments are:

When exception taken from AArch32:



N, bit [31]

Set to the value of CPSR.N on taking an exception to Monitor mode, and copied to CPSR.N on executing an exception return operation in Monitor mode.

Z, bit [30]

Set to the value of CPSR.Z on taking an exception to Monitor mode, and copied to CPSR.Z on executing an exception return operation in Monitor mode.

C, bit [29]

Set to the value of CPSR.C on taking an exception to Monitor mode, and copied to CPSR.C on executing an exception return operation in Monitor mode.

V, bit [28]

Set to the value of CPSR.V on taking an exception to Monitor mode, and copied to CPSR.V on executing an exception return operation in Monitor mode.

Q, bit [27]

Cumulative saturation bit. Set to 1 to indicate that overflow or saturation occurred in some instructions.

Switch to EL2 from EL3

```
        /* Prepare the switch to EL2 mode from EL3 mode */
        ldr     x0, __start_el2         /* Return after mode switch */
        mov     x1, #0x3c9              /* EL2_SP1 | D | A | I | F */
        msr     elr_el3, x0
        msr     spsr_el3, x1
        eret
...
__start_el2:
        .dword _start_el2
```

Saved Program Status Register

IT[1:0], bits [26:25]

If-Then Execution State bits for the T32 IT (If-Then) instruction. See IT[7:2] for explanation of this field.

J, bit [24]

RES0.

In previous versions of the architecture, the {J, T} bits determined the AArch32 Instruction set state. ARMv8 does not support either Jazelle state or T32EE state, and the T bit determines the Instruction set state.

Bits [23:21]

Reserved, RES0.

IL, bit [20]

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

GE, bits [19:16]

Greater than or Equal flags, for parallel addition and subtraction.

IT[7:2], bits [15:10]

If-Then Execution State bits for the T32 IT (If-Then) instruction. This field must be interpreted in two parts.

- IT[7:5] holds the base condition for the IT block. The base condition is the top 3 bits of the condition code specified by the first condition field of the IT instruction.
- IT[4:0] encodes the size of the IT block, which is the number of instructions that are to be conditionally executed, by the position of the least significant 1 in this field. It also encodes the value of the least significant bit of the condition code for each instruction in the block.

The IT field is `0b00000000` when no IT block is active.

E, bit [9]

Endianness Execution State bit. Controls the load and store endianness for data accesses:

`0` Little-endian operation

`1` Big-endian operation.

Instruction fetches ignore this bit.

When the reset value of the SCTLR.EE bit is defined by a configuration input signal, that value also applies to the CPSR.E bit on reset, and therefore applies to software execution from reset.

If an implementation does not provide Big-endian support, this bit is RES0. If it does not provide Little-endian support, this bit is RES1.

If an implementation provides Big-endian support but only at EL0, this bit is RES0 for an exception return to any exception level other than EL0.

Likewise, if it provides Little-endian support only at EL0, this bit is RES1 for an exception return to any exception level other than EL0.

A, bit [8]

Asynchronous data abort mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

`0` Exception not masked.

`1` Exception masked.

Switch to EL2 from EL3

```
        /* Prepare the switch to EL2 mode from EL3 mode */
        ldr     x0, __start_el2         /* Return after mode switch */
        mov     x1, #0x3c9              /* EL2_SP1 | D | A | I | F */
        msr     elr_el3, x0
        msr     spsr_el3, x1
        eret
...
__start_el2:
        .dword _start_el2
```

Saved Program Status Register

F, bit [6]

FIQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

T, bit [5]

T32 Instruction set state bit. Determines the AArch32 instruction set state that the exception was taken from. Possible values of this bit are:

0 Taken from A32 state.

1 Taken from T32 state.

M[4], bit [4]

Execution state that the exception was taken from. Possible values of this bit are:

1 Exception taken from AArch32.

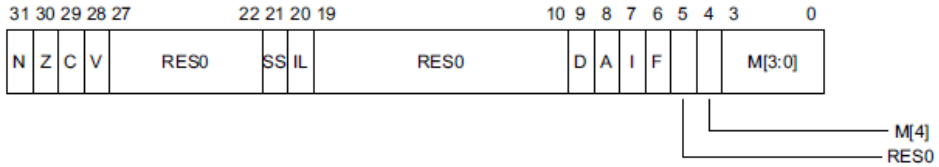
M[3:0], bits [3:0]

Mode that an exception was taken from. For exceptions taken from AArch32, the possible values are:

M[3:0]	Mode
0b0000	User
0b0001	FIQ
0b0010	IRQ
0b0011	Supervisor
0b0110	Monitor
0b0111	Abort
0b1010	Hyp
0b1011	Undefined
0b1111	System

Other values are reserved.

When exception taken from AArch64:



N, bit [31]

Set to the value of the N condition flag on taking an exception to EL3, and copied to the N condition flag on executing an exception return operation in EL3.

Switch to EL2 from EL3

```
        /* Prepare the switch to EL2 mode from EL3 mode */
        ldr     x0, __start_el2         /* Return after mode switch */
        mov     x1, #0x3c9              /* EL2_SP1 | D | A | I | F */
        msr     elr_el3, x0
        msr     spsr_el3, x1
        eret
...
__start_el2:
        .dword _start_el2
```

Saved Program Status Register

Z, bit [30]

Set to the value of the Z condition flag on taking an exception to EL3, and copied to the Z condition flag on executing an exception return operation in EL3.

C, bit [29]

Set to the value of the C condition flag on taking an exception to EL3, and copied to the C condition flag on executing an exception return operation in EL3.

V, bit [28]

Set to the value of the V condition flag on taking an exception to EL3, and copied to the V condition flag on executing an exception return operation in EL3.

Bits [27:22]

Reserved, RES0.

SS, bit [21]

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

IL, bit [20]

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

Bits [19:10]

Reserved, RES0.

D, bit [9]

Process state D mask. The possible values of this bit are:

0 Debug exceptions from Watchpoint, Breakpoint, and Software step debug events targeted at the current exception level are not masked.

1 Debug exceptions from Watchpoint, Breakpoint, and Software step debug events targeted at the current exception level are masked.

When the target exception level of the debug exception is not than the current exception level, the exception is not masked by this bit.

A, bit [8]

SError (System Error) mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

I, bit [7]

IRQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

F, bit [6]

FIQ mask bit. The possible values of this bit are:

0 Exception not masked.

1 Exception masked.

Bit [5]

Reserved, RES0.

Switch to EL2 from EL3

```
        /* Prepare the switch to EL2 mode from EL3 mode */
        ldr     x0, __start_el2         /* Return after mode switch */
        mov     x1, #0x3c9              /* EL2_SP1 | D | A | I | F */
        msr     elr_el3, x0
        msr     spsr_el3, x1
        eret
...
__start_el2:
        .dword _start_el2
```

Saved Program Status Register

M[4], bit [4]

Execution state that the exception was taken from. Possible values of this bit are:

0 Exception taken from AArch64.

M[3:0], bits [3:0]

Mode that an exception was taken from. For exceptions taken from AArch64, the possible values are:

M[3:0]	Mode
0b0000	EL0t
0b0100	EL1t
0b0101	EL1h
0b1000	EL2t
0b1001	EL2h
0b1100	EL3t
0b1101	EL3h

Other values are reserved.

For exceptions from AArch64:

- M[3:2] holds the Exception Level.
- M[1] is unused, and returning to an exception level that is using AArch64 with this bit set is treated as an illegal exception return.
- M[0] is used to select the SP:
  - 0 means the SP is always SP0.
  - 1 means the exception SP is determined by the EL.

Accessing the SPSR_EL3

To access the SPSR_EL3:

```
MRS <Xt>, SPSR_EL3 ; Read SPSR_EL3 into Xt
MSR SPSR_EL3, <Xt> ; Write Xt to SPSR_EL3
```

Register access is encoded as follows:

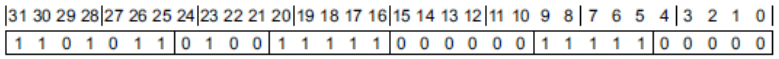
op0	op1	CRn	CRm	op2
11	110	0100	0000	000

Switch to EL2 from EL3

```
        /* Prepare the switch to EL2 mode from EL3 mode */
        ldr     x0, __start_el2         /* Return after mode switch */
        mov     x1, #0x3c9              /* EL2_SP1 | D | A | I | F */
        msr     elr_el3, x0
        msr     spsr_el3, x1
        eret
...
__start_el2:
        .dword _start_el2
```

C6.6.66 ERET

Exception return using current ELR and SPSR



System variant

ERET

```
if PSTATE.EL == EL0 then UnallocatedEncoding();
```

Operation

```
AArch64.ExceptionReturn(ELR[], SPSR[]);
```

C3.1.4 Exception generation and return

This section describes the following exceptions:

- *Exception generating*.
- *Exception return* on page C3-128.
- *Debug state* on page C3-128.

Exception generating

Table C3-4 shows the Exception generating instructions.

Table C3-4 Exception generating instructions

Mnemonic	Instruction	See
BRK	Software breakpoint instruction	*BRK* on page C6-428
HLT	Halting software breakpoint instruction	*HLT* on page C6-479
HVC	Generate exception targeting Exception level 2	*HVC* on page C6-480
SMC	Generate exception targeting Exception level 3	*SMC* on page C6-658
SVC	Generate exception targeting Exception level 1	*SVC* on page C6-743

Exception return

Table C3-5 shows the Exception return instructions.

Table C3-5 Exception return instructions

Mnemonic	Instruction	See
ERET	Exception return using current ELR and SPSR	*ERET* on page C6-474

Debug state

Table C3-6 shows the Debug state instructions.

Table C3-6 Debug state instructions

Mnemonic	Instruction	See
DCPS1	Debug switch to Exception level 1	*DCPS1* on page C6-461
DCPS2	Debug switch to Exception level 2	*DCPS2* on page C6-462
DCPS3	Debug switch to Exception level 3	*DCPS3* on page C6-463
DRPS	Debug restore PE state	*DRPS* on page C6-466

Exception generation and return

Start running in EL2

```
_start_el2:
        /* Skip secondary loop for Primary core */
        mrs     x4, mpidr_el1
        ands    x4, x4, #0xffff
        beq     __secondary_spin_skip

        /* Copy the secondary_spin(start, end) to SPIN_LOOP_ADDR */
        adr     x0, __secondary_spin_code_start
        adr     x1, __secondary_spin_code_end
        ldr     x2, __secondary_spin_loc

1:      ldr     x4, [x0], #8
        str     x4, [x2], #8
        cmp     x1, x0
        bge     1b
        adr     x3, __secondary_spin
        sub     x3, x3, x0
        add     x3, x3, x2
        br      x3

        .align 3
__secondary_spin_code_start:
        .dword  0x0
__secondary_spin_loc:
        .dword  SPIN_LOOP_ADDR
__leds_base:
        .dword  0x1c010008
#ifdef DTB
__dtb_addr:
        .dword  dtb
#endif

__secondary_spin:
        ldr     x0, __secondary_spin_loc
        sevl
1:      wfe
        ldr     x1, [x0]
        cmp     x1, xzr
        beq     1b

        ldr     x15, __leds_base
        mov     w14, #1
        str     w14, [x15]

        br      x1
__secondary_spin_code_end:
```

ANDS (immediate)

C6.6.13 ANDS (immediate)

Bitwise AND (immediate), setting the condition flags: `Rd = Rn AND imm`

This instruction is used by the alias TST (immediate). See the *Alias conditions* table for details of when each alias is preferred.

31	30 29	28	27 26 25 24	23 22	21	16	15	10	9	5	4	0
sf	1 1	1	0 0 1 0	0	N	immr		imms		Rn		Rd
	opc											

32-bit variant (sf = 0, N = 0)

```
ANDS <Wd>, <Wn>, #<imm>
```

64-bit variant (sf = 1)

```
ANDS <Xd>, <Xn>, #<imm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
boolean setflags;
LogicalOp op;
case opc of
    when '00' op = LogicalOp_AND; setflags = FALSE;
    when '01' op = LogicalOp_ORR; setflags = FALSE;
    when '10' op = LogicalOp_EOR; setflags = FALSE;
    when '11' op = LogicalOp_AND; setflags = TRUE;

bits(datasize) imm;
if sf == '0' && N != '0' then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

Alias conditions

Alias	is preferred when
TST (immediate)	Rd == '11111'

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the Rd field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the Rn field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the Rd field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the Rn field.
<imm>	Is the bitmask immediate, encoded in N:imms:immr.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = imm;

case op of
    when LogicalOp_AND result = operand1 AND operand2;
```

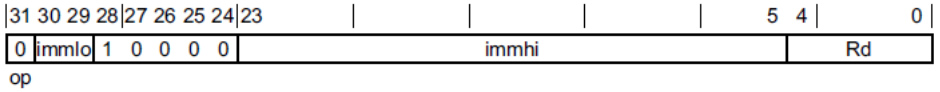
Start running in EL2

```
_start_el2:
        /* Skip secondary loop for Primary core */
        mrs     x4, mpidr_el1
        ands    x4, x4, #0xffff
        beq     __secondary_spin_skip

        /* Copy the secondary_spin(start, end) to SPIN_LOOP_ADDR */
        adr     x0, __secondary_spin_code_start
        adr     x1, __secondary_spin_code_end
        ldr     x2, __secondary_spin_loc

1:      ldr     x4, [x0], #8
        str     x4, [x2], #8
        cmp     x1, x0
        bge     1b
        adr     x3, __secondary_spin
        sub     x3, x3, x0
        add     x3, x3, x2
        br      x3

        .align 3
__secondary_spin_code_start:
        .dword  0x0
__secondary_spin_loc:
        .dword  SPIN_LOOP_ADDR
__leds_base:
        .dword  0x1c010008
#ifdef DTB
__dtb_addr:
        .dword  dtb
#endif

__secondary_spin:
        ldr     x0, __secondary_spin_loc
        sevl
1:      wfe
        ldr     x1, [x0]
        cmp     x1, xzr
        beq     1b

        ldr     x15, __leds_base
        mov     w14, #1
        str     w14, [x15]

        br      x1
__secondary_spin_code_end:
```

ADR

C6.6.9 ADR

Address of label at a PC-relative offset



Literal variant

```
ADR <Xd>, <label>

integer d = UInt(Rd);
boolean page = (op == '1');
bits(64) imm;

if page then
    imm = SignExtend(immhi:immlo:Zeros(12), 64);
else
    imm = SignExtend(immhi:immlo, 64);
```

Assembler symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the Rd field.
<label>	Is the program label whose address is to be calculated. Its offset from the address of this instruction, in the range +/-1MB, is encoded in immhi:immlo.

Operation

```
bits(64) base = PC[];

if page then
    base<11:0> = Zeros(12);

X[d] = base + imm;
```

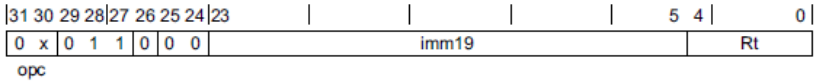
Start running in EL2

```
_start_el2:
        /* Skip secondary loop for Primary core */
        mrs     x4, mpidr_el1
        ands    x4, x4, #0xffff
        beq     __secondary_spin_skip

        /* Copy the secondary_spin(start, end) to SPIN_LOOP_ADDR */
        adr     x0, __secondary_spin_code_start
        adr     x1, __secondary_spin_code_end
        ldr     x2, __secondary_spin_loc

1:      ldr     x4, [x0], #8
        str     x4, [x2], #8
        cmp     x1, x0
        bge     1b
        adr     x3, __secondary_spin
        sub     x3, x3, x0
        add     x3, x3, x2
        br      x3

        .align 3
__secondary_spin_code_start:
        .dword  0x0
__secondary_spin_loc:
        .dword  SPIN_LOOP_ADDR
__leds_base:
        .dword  0x1c010008
#ifdef DTB
__dtb_addr:
        .dword  dtb
#endif

__secondary_spin:
        ldr     x0, __secondary_spin_loc
        sevl
1:      wfe
        ldr     x1, [x0]
        cmp     x1, xzr
        beq     1b

        ldr     x15, __leds_base
        mov     w14, #1
        str     w14, [x15]

        br      x1
__secondary_spin_code_end:
```

LDR (literal)

C6.6.84 LDR (literal)

Load register (PC-relative literal)



32-bit variant (opc = 00)

```
LDR <Wt>, <label>
```

64-bit variant (opc = 01)

```
LDR <Xt>, <label>

integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
    when '00'
        size = 4;
    when '01'
        size = 8;
    when '10'
        size = 4;
        signed = TRUE;
    when '11'
        memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the Rt field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the Rt field.
<label>	Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as imm19 times 4.

Operation

```
bits(64) address = PC[] + offset;
bits(size*8) data;

case memop of
    when MemOp_LOAD
        data = Mem[address, size, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, 64);
        else
            X[t] = data;

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Start running in EL2

```
_start_el2:
        /* Skip secondary loop for Primary core */
        mrs     x4, mpidr_el1
        ands    x4, x4, #0xffff
        beq     __secondary_spin_skip

        /* Copy the secondary_spin(start, end) to SPIN_LOOP_ADDR */
        adr     x0, __secondary_spin_code_start
        adr     x1, __secondary_spin_code_end
        ldr     x2, __secondary_spin_loc

1:      ldr     x4, [x0], #8
        str     x4, [x2], #8
        cmp     x1, x0
        bge     1b
        adr     x3, __secondary_spin
        sub     x3, x3, x0
        add     x3, x3, x2
        br      x3

        .align 3
__secondary_spin_code_start:
        .dword  0x0
__secondary_spin_loc:
        .dword  SPIN_LOOP_ADDR
__leds_base:
        .dword  0x1c010008
#ifdef DTB
__dtb_addr:
        .dword  dtb
#endif

__secondary_spin:
        ldr     x0, __secondary_spin_loc
        sevl
1:      wfe
        ldr     x1, [x0]
        cmp     x1, xzr
        beq     1b

        ldr     x15, __leds_base
        mov     w14, #1
        str     w14, [x15]

        br      x1
__secondary_spin_code_end:
```

LDR (immediate)

C6.6.83 LDR (immediate)

Load register (immediate offset)

It has encodings from 3 classes: *Post-index*, *Pre-index* and *Unsigned offset*

Post-index

31	30	29	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1	x	1	1	1	0	0	0	0	1	0		imm9	0	1		Rn		Rt
size								opc										

32-bit variant (size = 10)

```
LDR <Wt>, [<Xn|SP>], #<simm>
```

64-bit variant (size = 11)

```
LDR <Xt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20	12	11	10	9	5	4	0
1	x	1	1	1	0	0	0	0	1	0		imm9	1	1		Rn		Rt
size								opc										

32-bit variant (size = 10)

```
LDR <Wt>, [<Xn|SP>, #<simm>]!
```

64-bit variant (size = 11)

```
LDR <Xt>, [<Xn|SP>, #<simm>]!

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21	10	9	5	4	0
1	x	1	1	1	0	0	1	0	1		imm12		Rn		Rt
size								opc							

32-bit variant (size = 10)

```
LDR <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit variant (size = 11)

```
LDR <Xt>, [<Xn|SP>{, #<pimm>}]

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Start running in EL2

```
_start_el2:
        /* Skip secondary loop for Primary core */
        mrs     x4, mpidr_el1
        ands    x4, x4, #0xffff
        beq     __secondary_spin_skip

        /* Copy the secondary_spin(start, end) to SPIN_LOOP_ADDR */
        adr     x0, __secondary_spin_code_start
        adr     x1, __secondary_spin_code_end
        ldr     x2, __secondary_spin_loc

1:      ldr     x4, [x0], #8
        str     x4, [x2], #8
        cmp     x1, x0
        bge     1b
        adr     x3, __secondary_spin
        sub     x3, x3, x0
        add     x3, x3, x2
        br      x3

        .align 3
__secondary_spin_code_start:
        .dword  0x0
__secondary_spin_loc:
        .dword  SPIN_LOOP_ADDR
__leds_base:
        .dword  0x1c010008
#ifdef DTB
__dtb_addr:
        .dword  dtb
#endif

__secondary_spin:
        ldr     x0, __secondary_spin_loc
        sevl
1:      wfe
        ldr     x1, [x0]
        cmp     x1, xzr
        beq     1b

        ldr     x15, __leds_base
        mov     w14, #1
        str     w14, [x15]

        br      x1
__secondary_spin_code_end:
```

LDR (immediate)

Operation for all classes

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
```


tion Descriptions
structions

```
        when Constraint_NOP        EndOfInstruction();

 n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Start running in EL2

```
_start_el2:
        /* Skip secondary loop for Primary core */
        mrs     x4, mpidr_el1
        ands    x4, x4, #0xffff
        beq     __secondary_spin_skip

        /* Copy the secondary_spin(start, end) to SPIN_LOOP_ADDR */
        adr     x0, __secondary_spin_code_start
        adr     x1, __secondary_spin_code_end
        ldr     x2, __secondary_spin_loc

1:      ldr     x4, [x0], #8
        str     x4, [x2], #8
        cmp     x1, x0
        bge     1b
        adr     x3, __secondary_spin
        sub     x3, x3, x0
        add     x3, x3, x2
        br      x3

        .align 3
__secondary_spin_code_start:
        .dword  0x0
__secondary_spin_loc:
        .dword  SPIN_LOOP_ADDR
__leds_base:
        .dword  0x1c010008
#ifdef DTB
__dtb_addr:
        .dword  dtb
#endif

__secondary_spin:
        ldr     x0, __secondary_spin_loc
        sevl
1:      wfe
        ldr     x1, [x0]
        cmp     x1, xzr
        beq     1b

        ldr     x15, __leds_base
        mov     w14, #1
        str     w14, [x15]

        br      x1
__secondary_spin_code_end:
```

BR

C6.6.28 BR

Branch to register, branches unconditionally to an address in a register, with a hint that this is not a subroutine return

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	5	4	3	2	1	0
1	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	Rn			0	0	0	0
op																												

Integer variant

BR <Xn>

```
integer n = UInt(Rn);
BranchType branch_type;

case op of
    when '00' branch_type = BranchType_JMP;
    when '01' branch_type = BranchType_CALL;
    when '10' branch_type = BranchType_RET;
    otherwise UnallocatedEncoding();
```

Assembler symbols

<Xn>	Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the Rn field.

Operation

```
bits(64) target = X[n];

if branch_type == BranchType_CALL then X[30] = PC[] + 4;
BranchTo(target, branch_type);
```

x2’

x3’

x2

x1,x0’

x3

x0

Start running in EL2

```
_start_el2:
        /* Skip secondary loop for Primary core */
        mrs     x4, mpidr_el1
        ands    x4, x4, #0xffff
        beq     __secondary_spin_skip

        /* Copy the secondary_spin(start, end) to SPIN_LOOP_ADDR */
        adr     x0, __secondary_spin_code_start
        adr     x1, __secondary_spin_code_end
        ldr     x2, __secondary_spin_loc

1:      ldr     x4, [x0], #8
        str     x4, [x2], #8
        cmp     x1, x0
        bge     1b
        adr     x3, __secondary_spin
        sub     x3, x3, x0
        add     x3, x3, x2
        br      x3

        .align 3
__secondary_spin_code_start:
        .dword  0x0
__secondary_spin_loc:
        .dword  SPIN_LOOP_ADDR
__leds_base:
        .dword  0x1c010008
#ifdef DTB
__dtb_addr:
        .dword  dtb
#endif

__secondary_spin:
        ldr     x0, __secondary_spin_loc
        sevl
1:      wfe
        ldr     x1, [x0]
        cmp     x1, xzr
        beq     1b

        ldr     x15, __leds_base
        mov     w14, #1
        str     w14, [x15]

        br      x1
__secondary_spin_code_end:
```

C6.6.68 HINT

Hint instruction

This instruction is used by the aliases NOP, SEVL, SEV, WFE, WFI, and YIELD. See the *Alias conditions* table for details of when each alias is preferred.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11 – 8	7 – 5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	CRm	op2	1	1	1	1	1

System variant

```
HINT #<imm>

SystemHintOp op;

case CRm:op2 of
    when '0000 000' op = SystemHintOp_NOP;
    when '0000 001' op = SystemHintOp_YIELD;
    when '0000 010' op = SystemHintOp_WFE;
    when '0000 011' op = SystemHintOp_WFI;
    when '0000 100' op = SystemHintOp_SEV;
    when '0000 101' op = SystemHintOp_SEVL;
    otherwise  op = SystemHintOp_NOP;
```

Alias conditions

Alias	is preferred when
NOP	UInt(CRm:op2) == 0
SEVL	UInt(CRm:op2) == 5
SEV	UInt(CRm:op2) == 4
WFE	UInt(CRm:op2) == 2
WFI	UInt(CRm:op2) == 3
YIELD	UInt(CRm:op2) == 1

Assembler symbols

<imm> Is a 7-bit unsigned immediate, in the range 0 to 127, encoded in CRm:op2.

Operation

```
case op of
    when SystemHintOp_YIELD
        Hint_Yield();

    when SystemHintOp_WFE
        if EventRegistered() then
            ClearEventRegister();
        else
            if PSTATE.EL == EL0 then
                AArch64.CheckForWFxTrap(EL1, TRUE);
            if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
                AArch64.CheckForWFxTrap(EL2, TRUE);
```


C6.6.16[illegible] SEVL

Send event local.

This instruction is an alias of the HINT instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11 – 8	7 – 5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	CRm	op2	1	1	1	1	1

System variant

SEVL
is equivalent to
HINT #5
and is the preferred disassembly when UInt(CRm:op2) == 5.

SEVL

ion Descriptions
of instructions

```
            if HaveEL(EL3) && PSTATE.EL != EL3 then
                AArch64.CheckForWFxTrap(EL3, TRUE);
            WaitForEvent();

    when SystemHintOp_WFI
        if !InterruptPending() then
            if PSTATE.EL == EL0 then
                AArch64.CheckForWFxTrap(EL1, FALSE);
            if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0,EL1} then
                AArch64.CheckForWFxTrap(EL2, FALSE);
            if HaveEL(EL3) && PSTATE.EL != EL3 then
                AArch64.CheckForWFxTrap(EL3, FALSE);
            WaitForInterrupt();

    when SystemHintOp_SEV
        SendEvent();

    when SystemHintOp_SEVL
        EventRegisterSet();

    otherwise // do nothing
```

Start running in EL2

```
_start_el2:
        /* Skip secondary loop for Primary core */
        mrs     x4, mpidr_el1
        ands    x4, x4, #0xffff
        beq     __secondary_spin_skip

        /* Copy the secondary_spin(start, end) to SPIN_LOOP_ADDR */
        adr     x0, __secondary_spin_code_start
        adr     x1, __secondary_spin_code_end
        ldr     x2, __secondary_spin_loc

1:      ldr     x4, [x0], #8
        str     x4, [x2], #8
        cmp     x1, x0
        bge     1b
        adr     x3, __secondary_spin
        sub     x3, x3, x0
        add     x3, x3, x2
        br      x3

        .align 3
__secondary_spin_code_start:
        .dword  0x0
__secondary_spin_loc:
        .dword  SPIN_LOOP_ADDR
__leds_base:
        .dword  0x1c010008
#ifdef DTB
__dtb_addr:
        .dword  dtb
#endif

__secondary_spin:
        ldr     x0, __secondary_spin_loc
        sevl
1:      wfe
        ldr     x1, [x0]
        cmp     x1, xzr
        beq     1b

        ldr     x15, __leds_base
        mov     w14, #1
        str     w14, [x15]

        br      x1
__secondary_spin_code_end:
```

LEDs

Table 3-2 ARMv8-A Foundation Model memory map

Start address	End address	Foundation v1 peripheral	Foundation v2 peripheral	Size	Security (v2 only)
0x00_0000_0000	0x00_03FF_FFFF	RAM	Trusted Boot ROM, secureflash	64MB	S
0x00_0400_0000	0x00_0403_FFFF		Trusted SRAM	256KB	S
0x00_0600_0000	0x00_07FF_FFFF		Trusted DRAM	32MB	S
0x00_0800_0000	0x00_0BFF_FFFF	-	NOR flash, flash0	64MB	S/NS
0x00_0C00_0000	0x00_0FFF_FFFF	-	NOR flash, flash1	64MB	S/NS
0x00_1800_0000	0x00_19FF_FFFF	-	Warning + RAZ/WI		
0x00_1A00_0000	0x00_1AFF_FFFF	Ethernet, SMSC 91C111	Ethernet, SMSC 91C111	16MB	S/NS
0x00_1C01_0000	0x00_1C01_FFFF	System Registers	System Registers	64KB	S/NS

3.3 System register block

The system register block that Table 3-5 shows provides a minimal set of registers.

Table 3-5 System register block

Offset	Type	Bits	Register
0x0000	R/O	[31:0]	System ID Register
0x0004	R/W	[7:0]	User Programmable Switches
0x0008	R/W	[7:0]	LEDs
0x00A0	R/W	[31:0]	System configuration data
0x00A4	R/W	[31:0]	System configuration control
0x00A8	R/W	[31:0]	System configuration status

The LEDs store eight bits of state that software can read or write on the model and can be viewed at runtime from the web interface.

等待有人写入非零值
并跳转!

Start running in EL2

```
__secondary_spin_skip:
#ifdef DTB
        /* if dtb provided load the address where we placed it */
        ldr     x0, __dtb_addr
#else
        /* deliberately put a non-8B aligned value to x0 skip dtb checking */
        mov     x0, #1
#endif
        /* Jump to input binary */
        b       input_bin

#define str(s)          #s
#define stringify(s)    str(s)

        .section .text, "ax", %progbits
#ifdef DTB
        /* DTB binary */
        .globl  dtb
        .balign 0x1000
dtb:
        .incbin stringify(DTB)
#endif
        /* Input binary containing OS images */
        .globl  input_bin
        .balign 0x10000
input_bin:
        .incbin stringify(IMAGE)
        .globl  input_bin_end
input_bin_end:
```

跳转到包含进来的代码入口，
即**cpu_entry.S!**