Note

- If a system call instruction is trapped, disabled, or is UNDEFINED because the Exception level has insufficient privilege to execute the instruction, the preferred exception return address is the address of the system call instruction.
- A system call is generated by the execution of an SVC, HVC, or SMC instruction.

When an exception is taken from an Exception level using AArch32 to an Exception level using AArch64, the top 32 bits of the modified ELR_ELx are 0.

D1.10.2 Exception vectors

When the PE takes an exception to an Exception level that is using AArch64, execution is forced to an address that is the *exception vector* for the exception. The exception vector exists in a *vector table* at the Exception level the exception is taken to.

A vector table occupies a number of consecutive word-aligned addresses in memory, starting at the *vector base address*.

Each Exception level has an associated *Vector Base Address Register* (VBAR), that defines the exception base address for the table at that Exception level.

For exceptions taken to AArch64 state, the vector table provides the following information:

- Whether the exception is one of the following:
  - Synchronous exception.
  - SError.
  - IRQ.
  - FIQ.
- Information about the Exception level that the exception came from, combined with information about the stack pointer in use, and the state of the register file.

Table D1-6 shows this:

Table D1-6 Vector offsets from vector table base address

Exception taken from	Offset for exception type			
	Synchronous	IRQ or vIRQ	FIQ or vFIQ	SError or vSError
Current Exception level with SP_EL0.	0x000	0x080	0x100	0x180
Current Exception level with SP_ELx, x>0.	0x200	0x280	0x300	0x380
Lower Exception level, where the implemented level immediately lower than the target level is using AArch64.[a]	0x400	0x480	0x500	0x580
Lower Exception level, where the implemented level immediately lower than the target level is using AArch32.[a]	0x600	0x680	0x700	0x780

a. For exceptions taken to EL3, if EL2 is implemented, the level immediately lower than the target level is EL2 if the exception was taken from Non-secure state, but EL1 if the exception was taken from Secure EL1 or EL0.

Reset is treated as a special vector for the highest implemented Exception level. This special vector uses an IMPLEMENTATION DEFINED address that is typically set either by a hardwired configuration of the PE or by configuration input signals. The RVBAR_ELx register contains this reset vector address, where *x* is the number of the highest implemented Exception level.

D1.10.3 Pseudocode description of exception entry to AArch64 state

The following pseudocode shows behavior when the PE takes an exception to an Exception level that is using AArch64.

```
// AArch64.TakeException()

// ======================

// Take an exception to an Exception Level using AArch64.


AArch64.TakeException(bits(2) target_el, ExceptionRecord exception,

                      bits(64) preferred_exception_return, integer vect_offset)

    assert !ELUsingAArch32(target_el);

    assert UInt(target_el) >= UInt(PSTATE.EL);


    // If being routed to from AArch32 state, the top parts of the X[] registers may

    // be set to zero

    if UsingAArch32() then MaybeZeroRegisterUppers(target_el);


    if UInt(target_el) > UInt(PSTATE.EL) then

        boolean lower_32;

        if target_el == EL3 then

            if !IsSecure() && HaveEL(EL2) then

                lower_32 = ELUsingAArch32(EL2);

            else

                lower_32 = ELUsingAArch32(EL1);

        else

            lower_32 = ELUsingAArch32(target_el - 1);

        vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);


    elsif PSTATE.SP == '1' then

        vect_offset = vect_offset + 0x200;


    spsr = GetSPSRFromPSTATE();


    if !(exception.type IN {Exception_IRQ, Exception_FIQ}) then

        AArch64.ReportException(exception, target_el);


    PSTATE.EL = target_el;

    PSTATE.nRW = '0';

    PSTATE.SP = '1';
```

```
    SPSR[] = spsr;
    ELR[] = preferred_exception_return;

    PSTATE.<D,A,I,F> = '1111';
    if spsr<4> == '1' then                  // Coming from AArch32
        PSTATE.IT = '00000000';
        PSTATE.J = '0';
        PSTATE.T = '0';

    BranchTo(VBAR[] + vect_offset, BranchType_EXCEPTION);
    EndOfInstruction();

// AArch64.ReportException()
// =========================
// Report syndrome information for exception taken to AArch64 state.

AArch64.ReportException(ExceptionRecord exception, bits(2) target_el)

    Exception type = exception.type;

    (ec,il) = AArch64.ExceptionClass(type, target_el);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    ESR[target_el] = ec<5:0>:il:iss;

    if type IN {Exception_InstructionAbort, Exception_PCAlignment, Exception_DataAbort,
                Exception_Watchpoint} then
        FAR[target_el] = exception.vaddress;
    else
        FAR[target_el] = bits(64) UNKNOWN;

    if target_el == EL2 && exception.ipavalid then
```

```
        HPFAR_EL2<39:4> = exception.ipaddress<47:12>;

    return;

// AArch64.ExceptionClass()
// ========================
// Return the Exception Class and Instruction Length fields for reported in ESR

(integer,bit) AArch64.ExceptionClass(Exception type, bits(2) target_el)

    il = if ThisInstrLength() == 32 then '1' else '0';
    from_32 = UsingAArch32();
    assert from_32 || il == '1';            // AArch64 instructions always 32-bit

    case type of
        when Exception_Uncategorized        ec = 0x00; il = '1';
        when Exception_WFxTrap              ec = 0x01;
        when Exception_CP15RTTrap           ec = 0x03;          assert from_32;
        when Exception_CP15RRTTrap          ec = 0x04;          assert from_32;
        when Exception_CP14RTTrap           ec = 0x05;          assert from_32;
        when Exception_CP14DTTrap           ec = 0x06;          assert from_32;
        when Exception_AdvSIMDFPAccessTrap  ec = 0x07;
        when Exception_FPIDTrap             ec = 0x08;
        when Exception_CP14RRTTrap          ec = 0x0C;          assert from_32;
        when Exception_IllegalState         ec = 0x0E; il = '1';
        when Exception_SupervisorCall       ec = 0x11;
        when Exception_HypervisorCall       ec = 0x12;
        when Exception_MonitorCall          ec = 0x13;
        when Exception_SystemRegisterTrap   ec = 0x18;          assert !from_32;
        when Exception_InstructionAbort     ec = 0x20; il = '1';
        when Exception_PCAlignment          ec = 0x22; il = '1';
        when Exception_DataAbort            ec = 0x24;
        when Exception_SPAlignment          ec = 0x26; il = '1'; assert !from_32;
        when Exception_FPTrappedException   ec = 0x28;
        when Exception_SError               ec = 0x2F; il = '1';
        when Exception_Breakpoint           ec = 0x30; il = '1';
        when Exception_SoftwareStep         ec = 0x32; il = '1';
```

```
        when Exception_Watchpoint           ec = 0x34; il = '1';
        when Exception_SoftwareBreakpoint   ec = 0x38;
        when Exception_VectorCatch          ec = 0x3A; il = '1'; assert from_32;
        otherwise                           Unreachable();

    if ec IN {0x20,0x24,0x30,0x32,0x34} && target_el == PSTATE.EL then
        ec = ec + 1;

    if ec IN {0x11,0x12,0x13,0x28,0x38} && !from_32 then
        ec = ec + 4;

    return (ec,il);

// MaybeZeroRegisterUppers()
// =========================
// On taking an exception to "handle_el" using AArch64 from AArch32, it is CONSTRAINED
// UNPREDICTABLE whether the top 32 bits of registers visible at any lower Exception level
// using AArch32 are set to zero.

MaybeZeroRegisterUppers(bits(2) handle_el)
    assert UsingAArch32() && !ELUsingAArch32(handle_el);

    SCR_RW = if HaveEL(EL3) then SCR_EL3.RW else '1';
    HCR_RW = if CurrentStateHasEL2() && SCR_RW == '1' then HCR_EL2.RW else SCR_RW;

    case SCR_RW:HCR_RW:handle_el of
        when '0011'  first = 0; last = 30; include_R15 = TRUE;
        when '101x'  first = 0; last = 30; include_R15 = FALSE;
        when '11xx'  first = 0; last = 14; include_R15 = FALSE;
        otherwise    Unreachable();

    for n = first to last
        if (n != 15 || include_R15) && ConstrainUnpredictableBool() then
            _R[n]<63:32> = Zeros();

    return;
```

D1.10.4 Exception classes

If the exception is a synchronous exception or an SError interrupt, information characterizing the reason for the exception is saved in the ESR_ELx at the Exception level the exception is taken to. The information saved is determined at the time the exception is taken, and is not changed as a result of the explicit synchronization that takes place at the start of taking the exception. See *Synchronization requirements for system registers* on page D8-1866.

Table D1-7 shows the possible encodings of the ESR_ELx.EC Exception Class field, for exceptions taken to AArch64 state. For each EC encoding:

- Table D1-7 includes a reference to the corresponding Instruction Specific Syndrome field, ESR_ELx.ISS.
- Table D1-75 on page D1-1513 indicates:
  - Which of the ESR_ELx the fault can be reported in.
  - Which Execution states the fault can be taken from.

Table D1-7 ESR_ELx.EC field encodings

EC	Meaning, and information on the corresponding *Instruction Specific Syndrome*, ESR_ELx.ISS. encoding.
0x00	Unknown reason. Generally used for exceptions that are a result of erroneous execution, see *Exceptions with an unknown reason* on page D1-1517. ESR_ELx.ISS is RES0.
0x01	Caused by a WFI or WFE instruction. Occurred because of a configurable trap, or a configurable enable or disable. For the encoding of the ISS field, see *Exception from a WFI or WFE instruction, from AArch32 or AArch64 state* on page D1-1518.
0x03	Caused by an MCR or MRC access to CP15, that is not reported using ESR_ELx.EC == 0x00. Occurred because of a configurable trap, or a configurable enable or disable. This code is only valid for exceptions taken from AArch32 state. For the encoding of the ISS field, see *Exception from an MCR or MRC access from AArch32 state* on page D1-1518.
0x04	Caused by an MCRR or MRRC access to CP15, that is not reported using ESR_ELx.EC == 0x00. Occurred because of a configurable trap, or a configurable enable or disable. This code is only valid for exceptions taken from AArch32 state. For the encoding of the ISS field, see *Exception from an MCRR or MRRC access from AArch32 state* on page D1-1519.
0x05	Caused by an MCR or MRC access to CP14. Occurred because of a configurable trap, or a configurable enable or disable. This code is only valid for exceptions taken from AArch32 state. For the encoding of the ISS field, see *Exception from an MCR or MRC access from AArch32 state* on page D1-1518.
0x06	Caused by an LDC or STC access to CP14. Occurred because of a configurable trap, or a configurable enable or disable. This code is only valid for exceptions taken from AArch32 state. Only valid for exceptions taken from AArch32 state. For the encoding of the ISS field, see *Exception from an LDC or STC access to CP14 from AArch32 state* on page D1-1520.
0x07	Caused by an access to SIMD or floating-point register[a]. Occurred because of a configurable trap, or a configurable enable or disable. For the encoding of the ISS field, see *Exception from an access to SIMD or floating-point registers, from AArch32 or AArch64* on page D1-1521.
0x08	Caused by an MRC or VMRS access to CP10 for MVFR0, MVFR1, MVFR2, or FPSID, from an ID Group trap, that is not reported using EC 0x07. This code is only valid for exceptions taken from AArch32 state. For the encoding of the ISS field, see *Exception from an MCR or MRC access from AArch32 state* on page D1-1518.
0x0C	Caused by an MRRC access to CP14. Occurred because of a configurable trap, or a configurable enable or disable. This code is only valid for exceptions taken from AArch32 state. For the encoding of the ISS field, see *Exception from an MCRR or MRRC access from AArch32 state* on page D1-1519.
0x0E	Illegal Execution State exception. SPSR_ELx.IL is 1. ESR_ELx.ISS is RES0.

Table D1-7 ESR_ELx.EC field encodings (continued)

EC	Meaning, and information on the corresponding *Instruction Specific Syndrome*, ESR_ELx.ISS. encoding.
0x11	SVC instruction executed in AArch32 state. For the encoding of the ISS field, see *Exception from HVC or SVC instruction execution* on page D1-1522.
0x12	HVC instruction executed in AArch32 state, when HVC instructions are not disabled[b]. For the encoding of the ISS field, see *Exception from HVC or SVC instruction execution* on page D1-1522.
0x13	SMC instruction executed in AArch32 state, when SMC instructions are not disabled[b]. For the encoding of the ISS field, see *Exception from SMC instruction execution in AArch32 state* on page D1-1522.
0x15	SVC instruction executed in AArch64 state. For the encoding of the ISS field, see *Exception from HVC or SVC instruction execution* on page D1-1522.
0x16	HVC instruction executed in AArch64 state, when HVC instructions are not disabled[b]. For the encoding of the ISS field, see *Exception from HVC or SVC instruction execution* on page D1-1522.
0x17	SMC instruction executed in AArch64 state, when SMC instructions are not disabled[b]. For the encoding of the ISS field, see *Exception from SMC instruction execution in AArch64 state* on page D1-1523.
0x18	Caused by an MSR, MRS, or System instruction, that is not reported using EC == 0x00, 0x01, or 0x07. Occurred because of a configurable trap, or a configurable enable or disable. For the encoding of the ISS field, see *Exception from MSR, MRS, or System instruction execution in AArch64 state* on page D1-1523.
0x20	Instruction Abort taken from a lower Exception level. Used for MMU faults generated by instruction accesses, and for synchronous external aborts, including synchronous parity errors. Not used for debug exceptions. For the encoding of the ISS field, see *Exception from an Instruction abort* on page D1-1524.
0x21	Instruction Abort taken without a change of Exception level. Used for MMU faults generated by instruction accesses, and for synchronous external aborts, including synchronous parity errors. Not used for debug exceptions. For the encoding of the ISS field, see *Exception from an Instruction abort* on page D1-1524.
0x22	Misaligned PC exception. See *PC alignment checking* on page D1-1423. ESR_ELx.ISS is RES0.
0x24	Data Abort taken from a lower Exception level. Used for MMU faults generated by data accesses, alignment faults other than stack pointer alignment faults, and for synchronous external aborts, including synchronous parity errors. Not used for debug exceptions. For the encoding of the ISS field, see *Exception from a Data abort* on page D1-1525.
0x25	Data Abort taken without a change of Exception level. Used for MMU faults generated by data accesses, alignment faults other than stack pointer alignment faults, and for synchronous external aborts, including synchronous parity errors. Not used for debug exceptions. For the encoding of the ISS field, see *Exception from a Data abort* on page D1-1525.
0x26	Stack Pointer Alignment exception. See *Stack pointer alignment checking* on page D1-1424. ESR_ELx.ISS is RES0.
0x28	Floating-point exception, if supported. Exceptions as a result of Floating-point exception traps, taken from AArch32 state. For the encoding of the ISS field, see *Floating-point exceptions* on page D1-1529.
0x2C	Floating-point exception, if supported. Exceptions as a result of Floating-point exception traps, taken from AArch64 state. For the encoding of the ISS field, see *Floating-point exceptions* on page D1-1529.

Table D1-7 ESR_ELx.EC field encodings (continued)

EC	Meaning, and information on the corresponding *Instruction Specific Syndrome*, ESR_ELx.ISS. encoding.
0x2F	SError interrupt. For the encoding of the ISS field, see *SError interrupt* on page D1-1530.
0x30	Breakpoint exception taken from a lower Exception level. For the encoding of the ISS field, see *Breakpoint exception or Vector Catch exception* on page D1-1530.
0x31	Breakpoint exception taken without a change of Exception level. For the encoding of the ISS field, see *Breakpoint exception or Vector Catch exception* on page D1-1530.
0x32	Software Step exception taken from a lower Exception level. For the encoding of the ISS field, see *Software Step exception* on page D1-1532.
0x33	Software Step exception taken without a change of Exception level. For the encoding of the ISS field, see *Software Step exception* on page D1-1532.
0x34	Watchpoint exception taken from a lower Exception level. For the encoding of the ISS field, see *Watchpoint exception* on page D1-1531.
0x35	Watchpoint exception taken without a change of Exception level. For the encoding of the ISS field, see *Watchpoint exception* on page D1-1531.
0x38	BKPT instruction executed in AArch32 state. For the encoding of the ISS field, see *Software Breakpoint Instruction exception* on page D1-1532.
0x3A	Vector Catch exception taken from AArch32 state. For the encoding of the ISS field, see *Breakpoint exception or Vector Catch exception* on page D1-1530.
0x3C	BRK instruction executed in AArch64 state. For the encoding of the ISS field, see *Software Breakpoint Instruction exception* on page D1-1532.

a. Except if the access is trapped because HCR_EL2.TGE is 1, in which case, the exception is reported with EC code 0x00
b. When HVC or SMC instructions are disabled, the resulting exceptions are reported with EC code 0x00.

EC encodings when routing general exceptions to EL2

When an exception is taken to EL2 because the exception routing control HCR_EL2.TGE is enabled, the EC encoding that would have been used if the exception had been taken to EL1 is recorded in ESR_EL2.EC instead, unless that encoding is 0x07.

Exceptions that use 0x07 when the HCR_EL2.TGE routing control is disabled use 0x00 when the HCR_EL2.TGE routing control is enabled.

Exceptions taken for an unknown reason, EC encoding 0x00

These are:

- Instruction-related encodings that are UNALLOCATED at all or the current Exception level, as follows:
  - UNALLOCATED instruction encodings.
  - Instruction encodings for instructions not implemented.
  - UNALLOCATED System register encodings. For example, System register encodings that are allocated for reads, or for writes, or for both reads and writes.
  - Debug state execution of instruction bit patterns that are UNALLOCATED in Debug state.
  - Non-debug state execution of instruction bit patterns that are UNALLOCATED in Non-debug state.
- An HVC instruction, when HVC instructions are disabled by SCR_EL3.HCE or HCR_EL2.HCD.

- An SMC instruction, when SMC instructions are disabled by SCR_EL3.SMD.
- An MSR or MRS to SP_EL0 when SPSel == 0.
- An HLT instruction, when HLT instructions are disabled by EDSCR.HDE.
- Exceptions from Debug state, as a result of any of the following:
  - A DCPS1 executed in Non-secure EL0 when HCR_EL2.TGE is 1.
  - A DCPS2 executed in EL1 or EL0 when SCR_EL3.NS is 0, or when EL2 is not implemented.
  - A DCPS3 executed when EDSCR.SDD is 1, or when EL3 is not implemented.
  - An instruction that is trapped to EL3 in Non-debug state, that is executed in EL2, EL1, or EL0 when EDSCR.SDD is 1.
- Exceptions from AArch32 state, as follows:
  - Exceptions generated by any of the SCTLR_EL1.{ITD, SED, CP15BEN, THEE} control bits.
  - SRS using R13_mon from Secure EL1 when EL3 is using AArch64. See *Traps to EL3 of monitor functionality from Secure EL1 using AArch32* on page D1-1500.
  - MRS or MSR (Banked register) to SPSR_mon, R13_mon, or R14_mon.
  - Short vector VFP instructions.
  - A DCPS3 executed in Debug state when EL3 is using AArch32.
- Exceptions that would normally have an EC encoding of 0x07, but that instead are taken to EL2 because HCR_EL2.TGE is 1.

D1.11 Exception return

In the ARMv8-A architecture, an exception return is always to the same Exception level or a lower Exception level. An exception return is used for:

- A return to a previously executing thread.
- Entry to a new execution thread. For example:
  - The initialization of a hypervisor by a Secure monitor.
  - The initialization of an operating system by a hypervisor.
  - Application entry from an operating system or hypervisor.

An exception return requires the simultaneous restoration of the PC and PSTATE to values that are consistent with the desired state of execution on returning from the exception.

In AArch64 state, an ERET instruction causes an exception return. On an ERET instruction:

- The PC is restored with the value held in the ELR_ELx.
- PSTATE is restored by using the contents of the SPSR_ELx.

The ELR_ELx and SPSR_ELx are the ELR_ELx and SPSR_ELx at the Exception level the exception is returning from.

Note

When returning from an Exception level using AArch64 to an Exception level using AArch32, the top 32 bits of the ELR_ELx are ignored.

An ERET instruction also:

- Sets the Event Register for the PE executing the ERET instruction. See *Mechanisms for entering a low-power state* on page D1-1533.
- Resets the local exclusive monitor for the PE executing the ERET instruction. This removes the risk of errors that might be caused when a path to an exception return fails to include a CLREX instruction.

  Note

  This behavior prevents self-hosted debug from software stepping through an LDREX/STREX pair. However, when self-hosted debug is using software step, it is highly probable that the exclusive monitor state would be lost anyway, for other reasons. *Stepping code that uses exclusive monitors* on page D2-1645 describes this.

  It is IMPLEMENTATION DEFINED whether the resetting of the local exclusive monitor also resets the global exclusive monitor.

The ERET instruction is UNDEFINED in EL0.

When returning from an Exception level using AArch64 to an Exception level using AArch32, the AArch32 context is restored. The ARMv8-A architecture defines the relationship between AArch64 state and AArch32 state, for:

- General purpose registers.
- Special purpose registers.
- System registers.

In an implementation that includes EL3, the Security state can only change on returning from an exception if the return is from EL3 to a lower Exception level.

The following sections give more information:

- *Pseudocode description of exception return* on page D1-1440.
- *Exception return and PC alignment* on page D1-1440.
- *Illegal return events* on page D1-1441.