B1.3 System level operation and terminology overview

Several concepts are critical to the understanding of the system level architecture support.

B1.3.1 Modes, Privilege and Stacks

Mode, privilege and stack pointer are key concepts used in ARMv7-M.

Mode The microcontroller profile supports two modes (Thread and Handler modes). Handler mode is entered as a result of an exception. An exception return can only be issued in Handler mode.

Thread mode is entered on Reset, and can be entered as a result of an exception return.

Privilege Code can execute as privileged or unprivileged. Unprivileged execution limits or excludes access to some resources. Privileged execution has access to all resources. Handler mode is always privileged. Thread mode can be privileged or unprivileged.

Stack Pointer Two separate banked stack pointers exist, the Main Stack Pointer, and the Process Stack pointer. The Main Stack Pointer can be used in either Thread or Handler mode. The Process Stack Pointer can only be used in Thread mode. See *The SP registers* on page B1-8 for more details.

Table B1-1 shows the relationship between mode, privilege and stack pointer usage.

Table B1-1 Mode, privilege and stack relationship

Mode	Privilege	Stack Pointer	Example (typical) usage model
Handler	Privileged	Main	Exception handling
Handler	Unprivileged	Any	Reserved combination (Handler is always privileged)
Handler	Any	Process	Reserved combination (Handler always uses the Main stack)
Thread	Privileged	Main	Execution of a privileged process/thread using a common stack in a system that only supports privileged access

Table B1-1 Mode, privilege and stack relationship (continued)

Mode	Privilege	Stack Pointer	Example (typical) usage model
Thread	Privileged	Process	Execution of a privileged process/thread using a stack reserved for that process/thread in a system that only supports privileged access, or where a mix of privileged and unprivileged threads exist.
Thread	Unprivileged	Main	Execution of an unprivileged process/thread using a common stack in a system that supports privileged and unprivileged (User) access
Thread	Unprivileged	Process	Execution of an unprivileged process/thread using a stack reserved for that process/thread in a system that supports privileged and unprivileged (User) access

B1.3.2 Exceptions

An exception is a condition that changes the normal flow of control in a program. Exception behavior splits into two parts:

- Exception recognition when an exception event is generated and presented to the processor
- Exception processing (activation) when the processor is executing an exception entry, exception return, or exception handler code sequence. Migration from exception recognition to processing can be instantaneous.

Exceptions can be split into four categories

Reset Reset is a special form of exception which terminates current execution in a potentially unrecoverable way when reset is asserted. When reset is de-asserted execution is restarted from a fixed point.

Supervisor call (SVCall)
An exception which is explicitly caused by the `SVC` instruction. A supervisor call is used by application code to make a system (service) call to an underlying operating system. The `SVC` instruction enables the application to issue a system call that requires privileged access to the system and will execute in program order with respect to the application. ARMv7-M also supports an interrupt driven service calling mechanism PendSV (see *Interrupts* in *Overview of the exceptions supported* on page B1-14 for more details).

Fault A fault is an exception which results from an error condition due to instruction execution. Faults can be reported synchronously or asynchronously to the instruction which caused them. In general, faults are reported synchronously. The Imprecise BusFault is an asynchronous fault supported in the ARMv7-M profile.

A synchronous fault is always reported with the instruction which caused the fault. An asynchronous fault does not guarantee how it is reported with respect to the instruction which caused the fault.

Synchronous debug monitor exceptions are classified as faults. Watchpoints are asynchronous and treated as an interrupt.

Interrupt An interrupt is an exception, other than a reset, fault or a supervisor call. All interrupts are asynchronous to the instruction stream. Typically interrupts are used by other elements within the system which wish to communicate with the processor, including software running on other processors.

Each exception has:

- a priority level
- an exception number
- a vector in memory which defines the entry-point (address) for execution on taking the exception. The associated code is described as the exception handler or the interrupt service routine (ISR).

Each synchronous exception, other than reset, is in one of three possible states:

- an Inactive exception is one which is not Pending or Active
- a Pending exception is one where the exception event has been generated, and which has not yet started being processed on the processor
- an Active exception is one whose handler has been started on a processor, but processing is not complete. An Active exception can be either running or pre-empted by a higher priority exception.

Asynchronous exceptions can be in one of the three possible states or both Pending and Active at the same time, where one instance of the exception is Active, and a second instance of the exception is Pending.

Priority Levels and Execution Pre-emption

All exceptions are assigned a priority level, the exception priority. Three exceptions have fixed values, while all others can be altered by privileged software. In addition, the instruction stream executing on the processor has a priority level associated with it, the execution priority. An exception whose exception priority is sufficiently[1] higher than the execution priority will become active. In this case, the currently running instruction stream is pre-empted, and the exception that is taken is activated.

When an instruction stream is pre-empted by an exception other than reset, key context information is saved onto the stack automatically. Execution branches to the code pointed to by the exception vector that has been activated.

1. The concept of sufficiently higher relates to priority grouping within the exception prioritization model. Priority grouping is explained in *Priority grouping* on page B1-18

B1.4 Registers

The ARMv7-M profile has the following registers closely coupled to the core:

- general purpose registers R0-R12
- 2 Stack Pointer registers, SP_main and SP_process (banked versions of R13)
- the Link Register, LR (R14)
- the Program Counter, PC
- status registers for flags, exception/interrupt level, and execution state bits
- mask registers associated with managing the prioritization scheme for exceptions and interrupts
- a control register (CONTROL) to identify the current stack and thread mode privilege level.

All other registers described in this specification are memory mapped.

Note

Register access restrictions where stated apply to normal execution. Debug restrictions can differ, see *General rules applying to debug register access* on page C1-6, *Debug Core Register Selector Register (DCRSR)* on page C1-22 and *Debug Core Register Data Register (DCRDR)* on page C1-23.

B1.4.1 The SP registers

There are two stacks supported in ARMv7-M, each with its own (banked) stack pointer register.

- the Main stack – SP_main
- the Process stack – SP_process.

ARMv7-M implementations treat bits [1:0] as RAZ/WI. Software should treat bits [1:0] as SBZP for maximum portability across ARMv7 profiles.

The SP that is used by instructions which explicitly reference the SP is selected according to the function `LookUpSP()` described in *Pseudocode details for ARM core register access in the Thumb instruction set* on page B1-12.

The stack pointer that is used in exception entry and exit is described in the pseudocode sequences of the exception entry and exit, see *Exception entry behavior* on page B1-21 and *Exception return behavior* on page B1-25 for more details. SP_main is selected and initialized on reset, see *Reset behavior* on page B1-20.

B1.4.2 The special-purpose program status registers (xPSR)

Program status at the system level breaks down into three categories. They can be accessed as individual registers, a combination of any two from three, or a combination of all three using the `MRS` and `MSR` instructions.

- The Application Program Status Register APSR - User writeable flags. APSR handling of user writeable flags by the `MSR` and `MRS` instructions is consistent across all ARMv7 profiles.
- The Interrupt Program Status Register IPSR – Exception Number (for current execution)

- The Execution Program Status Register EPSR - Execution state bits

The APSR, IPSR and EPSR registers are allocated as mutually exclusive bitfields within a 32-bit register. The combination of the APSR, IPSR and EPSR registers is referred to as the xPSR register.

Table B1-2 The xPSR register layout

	31	30	29	28	27	26	25	24	23 ... 16	15 ... 10	9	8 ... 0
APSR	N	Z	C	V	Q							
IPSR												0 or Exception Number
EPSR						ICI/IT	T			ICI/IT	a	

a. While EPSR[9] is reserved, its associated bit location in memory for stacking xPSR context information is allocated to stack alignment support, see *Stack alignment on exception entry* on page B1-24

The APSR is modified by flag setting instructions and used to evaluate conditional execution in IT and conditional branch instructions. The flags (NZCVQ) are as described in *ARM core registers* on page A2-11. The flags are UNPREDICTABLE on reset.

The IPSR is written on exception entry and exit. It can be read using an `MRS` instruction. Writes to the IPSR by an `MSR` instruction are ignored. The IPSR Exception Number field is defined as follows:

- When in Thread mode, the value is 0.
- When in Handler mode, the value reflects the exception number as defined in *Exception number definition* on page B1-16.

The exception number is used to determine the currently executing exception and its entry vector (see *Exception number definition* on page B1-16 and *The vector table* on page B1-16).

On reset, the core is in Thread mode and the Exception Number field of the IPSR is cleared. As a result, the value 1 (the Reset Exception Number) is a transitory value, and not a valid IPSR Exception Number.

The EPSR contains the T-bit and overlaid IT/ICI execution state bits to support the `IT` instruction or interrupt-continue load/store instructions. All fields read as zero using an `MRS` instruction. `MSR` writes are ignored.

The EPSR T-bit supports the ARM architecture interworking model, however, as ARMv7-M only supports execution of Thumb instructions, it must always be maintained with the value T-bit == 1. Updates to the PC which comply with the Thumb instruction interworking rules must update the T-bit accordingly. The execution of an instruction with the EPSR T-bit clear will cause an invalid state (INVSTATE) UsageFault. The T-bit is set and the IT/ICI bits cleared on reset (see *Reset behavior* on page B1-20 for details).

The ICI/IT bits are used for saved IT state or saved exception-continuable instruction state.

- The IT bits provide context information for the conditional execution of a sequence of instructions such that it can be interrupted and restarted at the appropriate point. See the IT instruction definition in Chapter A6 *Thumb Instruction Details* for more information.
- The ICI bits provide information on the outstanding register list for exception-continuable multi-cycle load and store instructions.

The IT/ICI bits are assigned according to Table B1-3.

Table B1-3 ICI/IT bit allocation in the EPSR

EPSR[26:25]	EPSR[15:12]	EPSR[11:10]	Additional Information
IT[1:0]	IT[7:4]	IT[3:2]	See *ITSTATE* on page A6-10.
ICI[7:6] ('00')	ICI[5:2] (reg_num)	ICI[1:0] ('00')	See *Exceptions in LDM and STM operations* on page B1-30.

The IT feature takes precedence over the ICI feature if an exception-continuable instruction is used within an IT construct. In this situation, the multi-cycle load or store instruction is treated as restartable.

All unused bits in the individual or combined registers are reserved.

B1.4.3 The special-purpose mask registers

There are three special-purpose registers which are used for the purpose of priority boosting. Their function is explained in detail in *Execution priority and priority boosting within the core* on page B1-18:

- the exception mask register (PRIMASK) which has a 1-bit value
- the base priority mask (BASEPRI) which has an 8-bit value
- the fault mask (FAULTMASK) which has a 1-bit value.

All mask registers are cleared on reset. All unprivileged writes are ignored.

The formats of the mask registers are illustrated in Table B1-4.

Table B1-4 The special-purpose mask registers

	31 ... 8	7 ... 1	0
PRIMASK	RESERVED		PM
FAULTMASK	RESERVED		FM
BASEPRI	RESERVED	BASEPRI	

Implementations can support an IMPLEMENTATION DEFINED number of priorities in powers of 2. Where fewer than 256 priorities are implemented, the low-order bits of the BASEPRI field corresponding to the unimplemented priority bits are RAZ/WI.

These registers can be accessed using the `MSR/MRS` instructions. The `MSR` instruction includes an additional register mask value BASEPRI_MAX, which updates BASEPRI only where the new value increases the priority level (decreases BASEPRI to a non-zero value). See *MSR (register)* on page B4-8 for details.

In addition:

- FAULTMASK is set by the execution of the instruction: CPSID f
- FAULTMASK is cleared by the execution of the instruction: CPSIE f
- PRIMASK is set by the execution of the instruction: CPSID i
- PRIMASK is cleared by the execution of the instruction: CPSIE i.

B1.4.4 The special-purpose ~~control~~ register

The special-purpose CONTROL register is a 2-bit register defined as follows:

- bit [0] defines the Thread mode privilege (Handler mode is always privileged)
  - 0: Thread mode has privileged access
  - 1: Thread mode has unprivileged access.
- bit [1] defines the stack to be used
  - 0: SP_main is used as the current stack
  - 1: For Thread mode, SP_process is used for the current stack. For Handler mode, this value is reserved.
  - Software can update bit [1] in Thread mode. Explicit writes from Handler mode are ignored.
  - The bit is updated on exception entry and exception return. See the pseudocode in *Exception entry behavior* on page B1-21 and *Exception return behavior* on page B1-25 for more details.
- bits [31:2] reserved.

The CONTROL register is cleared on reset. The `MRS` instruction is used to read the register, and the `MSR` instruction is used to write the register. Unprivileged write accesses are ignored.

An ISB barrier instruction is required to ensure a CONTROL register write access takes effect before the next instruction is executed.

B1.4.5 Reserved special-purpose register bits

All unused bits in special-purpose registers are reserved. `MRS` and `MSR` instructions that access reserved bits treat them as RAZ/WI. For future software compatibility, the bits are UNK/SBZP. Software should write them to zero when initializing the register for a new process, otherwise software should restore reserved bits when updating or restoring a special-purpose register.

B1.5 Exception model

The exception model is central to the architecture and system correctness in the ARMv7-M profile. The ARMv7-M profile differs from the other ARMv7 profiles in using hardware saving and restoring of key context state on exception entry and exit, and using a table of vectors to determine the exception entry points. In addition, the exception categorization in the ARMv7-M profile is different from the other ARMv7 profiles.

B1.5.1 Overview of the exceptions supported

The following exceptions are supported by the ARMv7-M profile.

Reset Two levels of reset are supported by the ARMv7-M profile. The levels of reset control which register bit fields are forced to their reset values on the de-assertion of reset.

- Power-On Reset (POR) resets the core, System Control Space and debug logic.
- Local Reset resets the core and System Control Space except some fault and debug-related resources. For more details, see *Debug and reset* on page C1-13.

The Reset exception is permanently enabled, and has a fixed priority of -3.

NMI – Non Maskable Interrupt Non Maskable Interrupt is the highest priority exception other than reset. It is permanently enabled and has a fixed priority of -2.

NMI can be set to the Pending state by software (see *Interrupt Control State Register (ICSR)* on page B3-12) or hardware.

HardFault HardFault is the generic fault that exists for all classes of fault that cannot be handled by any of the other exception mechanisms. HardFault will typically be used for unrecoverable system failure situations, though this is not required, and some uses of HardFault might be recoverable. HardFault is permanently enabled and has a fixed priority of -1.

HardFault is used for fault escalation, see *Priority escalation* on page B1-19 for details.

MemManage The MemManage fault handles memory protection related faults which are determined by the Memory Protection Unit or by fixed memory protection constraints, for both instruction and data generated memory transactions. The fault can be disabled (in this case, a MemManage fault will escalate to HardFault). MemManage has a configurable priority.

BusFault The BusFault fault handles memory related faults other than those handled by the MemManage fault for both instruction and data generated memory transactions. Typically these faults will arise from errors detected on the system buses. Implementations are permitted to report synchronous or asynchronous BusFaults according to the circumstances that trigger the exceptions. The fault can be disabled (in this case, a BusFault will escalate to HardFault). BusFault has a configurable priority.

UsageFault The UsageFault fault handles non-memory related faults caused by the instruction execution. A number of different situations will cause usage faults, including:

- UNDEFINED Instructions
- invalid state on instruction execution

- errors on exception return
- disabled or unavailable coprocessor access.

The following can cause usage faults when the core is configured to report them:

- unaligned addresses on word and halfword memory accesses
- division by zero.

UsageFault can be disabled (in this case, a UsageFault will escalate to HardFault). UsageFault has a configurable priority.

Debug Monitor In general, a DebugMonitor exception is a synchronous exception and classified as a fault. Watchpoints are asynchronous and behave as an interrupt. Debug monitor exceptions occur when halting debug is disabled, and debug monitor support is enabled. DebugMonitor has a configurable priority. See *Priority escalation* on page B1-19 and *Debug event behavior* on page C1-14 for more details.

SVCall This supervisor call handles the exception caused by the `SVC` instruction. SVCall is permanently enabled and has a configurable priority.

Interrupts The ARMv7-M profile supports two system level interrupts – PendSV for software generation of asynchronous system calls, and SysTick for a Timer integral to the ARMv7-M profile – along with up to 496 external interrupts. All interrupts have a configurable priority.

PendSV[1] is a permanently enabled interrupt, controlled using ICSR.PENDSVSET and ICSR.PENDSVCLR (see *Interrupt Control State Register (ICSR)* on page B3-12). SysTick can not be disabled.

Note

While hardware generation of a SysTick event can be suppressed, ICSR.PENDSTSET and ICSR.PENDSTCLR (see *Interrupt Control State Register (ICSR)* on page B3-12) are always available to software.

All other interrupts can be disabled. Interrupts can be set to or cleared from the Pending state by software, and interrupts other than PendSV can be set to the Pending state by hardware.

See *Fault behavior* on page B1-39 for a definitive list of all the possible causes of faults, the type of fault reported, and the fault status register bits used to identify the faults.

1. A service (system) call is used by an application which requires a service from an underlying operating system. The service call associated with PendSV executes when the interrupt is taken. For a service call which executes synchronously with respect to program execution use the `SVC` instruction (the SVCall exception).

B1.5.2 Exception number definition

All exceptions have an associated exception number as defined in Table B1-5.

Table B1-5 Exception numbers

Exception number	Exception
1	Reset
2	NMI
3	HardFault
4	MemManage
5	BusFault
6	UsageFault
7-10	RESERVED
11	SVCall
12	Debug Monitor
13	RESERVED
14	PendSV
15	SysTick
16	External Interrupt(0)
…	…
16 + N	External Interrupt(N)

B1.5.3 The vector table

The vector table contains the initialization value for the stack pointer on reset, and the entry point addresses for all exception handlers. The exception number (see above) defines the order of entries in the vector table associated with exception handler entry as illustrated in Table B1-6.

Table B1-6 Vector table format

word offset	Description – all pointer address values
0	SP_main (reset value of the Main stack pointer)
Exception Number	Exception using that Exception Number

- the configurable fault is not enabled
- an SVC instruction when PRIMASK is set to 1.

Note

Enabled interrupts are not escalated – they are set to the Pending state. Disabled interrupts are ignored.

Asynchronous faults (Imprecise BusFaults) are set to the Pending state and are entered according to normal priority rules when enabled. They are treated as HardFault exceptions when disabled.

SVCall, PendSV and critical region code avoidance

Context switching typically requires a critical region of code where interrupts must be disabled to avoid context corruption of key data structures during the change. This can be a severe constraint on system design and deterministic performance. ARMv7-M can support context switching with no critical region such that interrupts never need to be disabled.

An example usage model supporting critical region avoidance is to configure both SVCall and PendSV with the same, lowest exception priority. SVCall can be used for supervisor calls from threads, and PendSV can be used to handle context critical work offloaded from the exception handlers, including the equal priority SVCall handler. Because SVCall and PendSV have the same execution priority they will never pre-empt each other, therefore one will always process to completion before the other starts. SVCall and PendSV exceptions are always enabled, which means they will each execute at some point once all other exceptions have been handled. In addition, the associated exception handlers do not need to check whether they are returning to a process on exit with this usage model, as the PendSV exception will occur when returning to a process.

The example has all context switch requests issued by setting PendSV to Pending, however, both SVCall and PendSV exceptions can be used for context switching because they do not interfere with each other. While not the only usage model, support of critical region software avoidance is a key feature of ARMv7-M, specifically the support provided by the SVCall and PendSV exception specifications.

B1.5.5 Reset behavior

The assertion of reset causes the current execution state to be abandoned without being saved. On the de-assertion of reset, all registers controlled by the reset assertion contain their reset values, and the following actions are performed.

For global declarations see *Register related definitions for pseudocode* on page B1-12.
For helper functions and procedures see *Miscellaneous helper procedures and functions* on page AppxG-22.

```
// TakeReset()
// ===========

integer NestedActivation;           /* used for Handler => Thread check when value == 1 */
bit ExceptionActive[*];             /* conceptual array of 1-bit values for all exceptions */
bits(32) vectortable = '00':VTOR<29:7>:'0000000';
Mode CurrentMode;
```

```
TakeReset()
    R[0..12] = bits(32) UNKNOWN;
    SP_main = MemA[vectortable,4] & 0xFFFFFFFC;
    SP_process = ((bits(30) UNKNOWN):'00');
    LR = 0xFFFFFFFF;                /* preset to an illegal exception return value */
    tmp = MemA[vectortable+4,4]
    PC = tmp AND 0xFFFFFFFE;        /* address of reset service routine */
    tbit = tmp<0>;
    CurrentMode = Mode_Thread;
    APSR = bits(32) UNKNOWN;        /* flags UNPREDICTABLE from reset */
    IPSR<8:0> = 0x0;                /* Exception Number cleared */
    EPSR.T = tbit;                  /* T bit set from vector */
    EPSR.IT<7:0> = 0x0;             /* IT/ICI bits cleared */
    PRIMASK<0> = '0';               /* priority mask cleared at reset */
    FAULTMASK<0> = '0';             /* fault mask cleared at reset */
    BASEPRI<7:0> = 0x0;             /* base priority disabled at reset */
    CONTROL<1:0> = '00';            /* current stack is Main, thread is privileged */
    ResetSCSRegs();                 /* catch-all function for System Control Space reset */
    NestedActivation = 0x0;         /* initialised value for base thread */
    ExceptionActive[*] = '0';       /* all exceptions Inactive */
    ClearExclusiveLocal();          /* Synchronization (LDREX*/STREX*) monitor support */
                                    /* to open access state. */
    ClearEventRegister()              /* see WFE instruction for more details */
```

ExceptionActive[*] is a conceptual array of active flag bits for all exceptions (fixed priority system exceptions, configurable priority system exceptions, and external interrupts). The fixed priority active flags are conceptual only, and are not required to exist in a system register.

B1.5.6 Exception entry behavior

On pre-emption of an instruction stream, context state is saved by the hardware onto a stack pointed to by one of the SP registers (see *The SP registers* on page B1-8). The stack that is used depends on the mode of the processor at the time of the exception.

The stacked context supports the ARM Architecture Procedure Calling Standard (AAPCS). The support allows the exception handler to be an AAPCS-compliant procedure.

A full-descending stack format is used, where the stack pointer is decremented immediately before storing ~~a 32-bit word~~ (when pushing context) onto the stack, and incremented after reading ~~a 32-bit word~~ (popping context) from the stack. Eight 32-bit words are saved in descending order, with respect to their address in memory, as listed:

xPSR, `ReturnAddress()`, LR (R14), R12, R3, R2, R1, and R0

The exception entry pseudocode is:

```
// ExceptionEntry()
// ================

// NOTE: PushStack() can abandon memory accesses if a fault occurs during the stacking
//       sequence.
//       Exception entry is modified according to the behavior of a derived exception,
//       see DerivedLateArrival() and associated text.
```

```
PushStack();
ExceptionTaken(ExceptionType);    // ExceptionType is encoded as its exception number
```

For global declarations see *Register related definitions for pseudocode* on page B1-12.
For a definition of ExceptionActive[*] and NestedActivation see *Reset behavior* on page B1-20.
For helper functons and procedures see *Miscellaneous helper procedures and functions* on page AppxG-22.

The PushStack() and ExceptionTaken() pseudo-functions are defined as follows:

```
// PushStack()
// ===========

PushStack()
    if CONTROL<1> == '1' AND CurrentMode == Mode_Thread then
        frameptralign = SP_process<2> AND CCR.STKALIGN;
        SP_process = (SP_process - 0x20) AND NOT(ZeroExtend(CCR.STKALIGN:'00',32));
        frameptr = SP_process;
    else
        frameptralign = SP_main<2> AND CCR.STKALIGN;
        SP_main = (SP_main - 0x20) AND NOT(ZeroExtend(CCR.STKALIGN:'00',32));
        frameptr = SP_main;
       /* only the stack locations, not the store order, are architected */
    MemA[frameptr,4]      = R[0];
    MemA[frameptr+0x4,4]  = R[1];
    MemA[frameptr+0x8,4]  = R[2];
    MemA[frameptr+0xC,4]  = R[3];
    MemA[frameptr+0x10,4] = R[12];
    MemA[frameptr+0x14,4] = LR;
    MemA[frameptr+0x18,4] = ReturnAddress();
    MemA[frameptr+0x1C,4] = (xPSR<31:10>:frameptralign:xPSR<8:0>);
                            // see ReturnAddress() in-line note for information on xPSR.IT bits
    if CurrentMode==Mode_Handler then
        LR = 0xFFFFFFF1;
    else
        if CONTROL<1> == '0' then
            LR = 0xFFFFFFF9;
        else
            LR = 0xFFFFFFFD;
    return;


// ExceptionTaken()
// ================

ExceptionTaken(bits(9) ExceptionNumber)

    bit tbit;
    bits(32) tmp;

    R[0..3] = bits(32) UNKNOWN;
    R[12] = bits(32) UNKNOWN;
    tmp = MemA[VectorTable+4*ExceptionNumber,4];
```

```
    PC = tmp AND 0xFFFFFFFE;
    tbit = tmp<0>;
    CurrentMode = Mode_Handler;
    APSR = bits(32) UNKNOWN;                // Flags UNPREDICTABLE due to other activations
    IPSR<8:0> = ExceptionNumber                 // ExceptionNumber set in IPSR
    EPSR.T = tbit;                              // T-bit set from vector
    EPSR.IT<7:0> = 0x0;                         // IT/ICI bits cleared
    /* PRIMASK, FAULTMASK, BASEPRI unchanged on exception entry*/
    CONTROL<1> = '0';                           // current Stack is Main, CONTROL<0> is unchanged
    /* CONTROL<0> unchanged */
    NestedActivation = NestedActivation + 1;
    ExceptionActive[ExceptionNumber]= '1';
    SCS_UpdateStatusRegs();                     // update SCS registers as appropriate
    ClearExclusiveLocal();
    SetEventRegister()                          // see WFE instruction for more details
    InstructionSynchronizationBarrier();
```

For more details on the registers with UNKNOWN values, see *Exceptions on exception entry* on page B1-33. For updates to system status registers, see section *System Control Space (SCS)* on page B3-6.

`ReturnAddress()` is the address to which execution will return after handling of the exception:

```
// ReturnAddress()
// ===============

Bits(32) ReturnAddress() returns the following values based on the exception cause
         // NOTE: ReturnAddress() is always halfword aligned - bit<0> is always zero
         //       xPSR.IT bits saved to the stack are consistent with ReturnAddress()

// Exception Type          Address returned
// ==============          ================

// NMI:                      Address of Next Instruction to be executed
// HardFault (precise):      Address of the Instruction causing fault
// HardFault (imprecise):    Address of Next Instruction to be executed
// MemManage:                Address of the Instruction causing fault
// BusFault (precise):       Address of the Instruction causing fault
// BusFault (imprecise):     Address of Next Instruction to be executed
// UsageFault:               Address of the Instruction causing fault
// SVC:                      Address of the Next Instruction after the SVC
// DebugMonitor (precise):   Address of the Instruction causing fault
// DebugMonitor (imprecise): Address of Next Instruction to be executed
// IRQ:                      Address of Next Instruction to be executed after an interrupt
```

Note

A fault which is escalated to the priority of a HardFault retains the ReturnAddress() behavior of the original fault. For a description of priority escalation see *Priority escalation* on page B1-19.

IRQ includes SysTick and PendSV

B1.5.7 Stack alignment on exception entry

ARMv7-M supports a configuration option to ensure that all exceptions are entered with 8-byte stack alignment. The stack pointers in ARMv7-M are guaranteed to be at least 4-byte aligned. As exceptions can occur on any instruction boundary, it is possible that the current stack pointer is not 8-byte aligned when an exception activates.

The AAPCS requires that the stack-pointer is 8-byte aligned on entry to a conforming function[1]. Since it is anticipated that exception handlers will be written as AAPCS conforming functions, the system must ensure natural alignment of the stack for all arguments passed. The 8-byte alignment requirement is guaranteed in hardware using a configuration feature.

The STKALIGN bit (see *Configuration and Control Register (CCR)* on page B3-16) is used to enable the 8-byte stack alignment feature. Whether the bit is programmable in software and its value on reset are IMPLEMENTATION DEFINED . The bit should be set in the system boot sequence prior to needing 8-byte alignment support.

Note

Software must ensure that any exception handler that can activate while CCR.STKALIGN == '0' does not require 8-byte alignment. An example is an NMI exception entered from reset, where the implementation resets to 4-byte alignment.

If the bit is cleared between the entry to and return from an exception, and if the stack was not 8-byte aligned on entry to the exception, system corruption can occur. Support of a 4-byte aligned stack (CCR.STKALIGN == '0') in ARMv7-M is deprecated.

Theory of operation

On an exception entry when STKALIGN == 1, the stack pointer (SP_main or SP_process) in use before the exception entry is forced to have 8-byte alignment by adjusting its alignment as part of the exception entry sequence. The xPSR that is saved as part of the exception entry sequence records the alignment of this stack pointer prior to the exception entry sequence. The alignment status is merged and stored to memory as bit [9] of the xPSR (a reserved bit within the xPSR) in the saved context information.

On an exception exit when STKALIGN == 1, the stack pointer returned to takes its alignment from the value recovered from bit [9] of the xPSR in the restored context from the exception exit sequence. This reverses the forced stack alignment performed on the exception entry.

See *Exception entry behavior* on page B1-21 and *Exception return behavior* on page B1-25 for pseudocode details of the effect of the STKALIGN feature on exception entry and exception return.

1. The AAPCS requires conforming functions to preserve the natural alignment of primitive data of size 1, 2, 4, and 8 bytes. In return, conforming code is permitted to rely on that alignment. To support unqualified reliance the stack-pointer must in general be 8-byte aligned on entry to a conforming function. If a function is entered directly from an underlying execution environment, that environment must accept the stack alignment obligation in order to give an unqualified guarantee that conforming code can execute correctly in all circumstances.

Note

Stack pointer alignment on exception exit is architecturally defined as an OR function. If the exception exit sequence is started with a stack pointer which is only 4 byte aligned, then this change has no effect.

In the event that the exception exit causes a derived exception, the derived exception is entered with the same stack alignment as was in use before the exception exit sequence started.

A side-effect when STKALIGN is enabled is that the amount of stack used on exception entry becomes a function of the alignment of the stack at the time that the exception is entered. As a result, the average and worst case stack usage will increase. The worst case increase is 4 bytes per exception entry.

Maintaining the stack alignment information in an unused bit within the saved xPSR makes the feature transparent to context switch code within operating systems, provided that the reserved status of unused bits with the xPSR have been respected.

Compatibility

Some operating systems can avoid saving and restoring R14 when switching between different processes in Thread mode if it is known that the values held in an EXC_RETURN value are invariant between the different processes. This provides a small improvement in context switch time, but at the cost of future compatibility. The STKALIGN feature does not affect the EXC_RETURN value. ARM does not guarantee that this software optimization will be possible in future revisions of the ARMv7-M architecture, and recommends for future compatibility that the R14 value is always saved and restored on a context switch.

B1.5.8 Exception return behavior

Exception returns occur when one of the following instructions loads a value of `0xFXXXXXXX` into the PC while in Handler mode:

- `POP/LDM` which includes loading the PC.
- `LDR` with `PC` as a destination.
- `BX` with any register.

When used in this way, the value written to the PC is intercepted and is referred to as the EXC_RETURN value.

EXC_RETURN[28:4] are reserved with the special condition that all bits should be written as one or preserved. Values other than all 1s are UNPREDICTABLE. EXC_RETURN[3:0] provide return information as defined in Table B1-8.

Table B1-8 Exception return behavior

EXC_RETURN[3:0]	
0bXXX0	RESERVED
0b0001	Return to Handler Mode; Exception return gets state from the Main stack; On return execution uses the Main Stack.
0b0011	RESERVED
0b01X1	RESERVED
0b1001	Return to Thread Mode; Exception return gets state from the Main stack; On return execution uses the Main Stack.
0b1101	Return to Thread Mode; Exception return gets state from the Process stack; On return execution uses the Process Stack.
0b1X11	RESERVED

RESERVED entries in this table result in a chained exception to a UsageFault.

If an EXC_RETURN value is loaded into the PC when in Thread mode, or from the vector table, or by any other instruction, the value is treated as an address, not as a special value. This address range is defined to have eXecute Never (XN) permissions, and will result in a MemManage exception, an INVSTATE UsageFault[1] exception, or the exception will escalate to a HardFault.

Integrity checks on exception returns

The ARMv7-M profile provides a number of integrity checks on an exception return. These exist as a guard against errors in the system software. Incorrect exception return information could be inconsistent with the state of execution which must be held in processor hardware or other state stored by the exception mechanisms.

The hardware related integrity checks ensure that the tracking of exception activation within the interrupt controller (NVIC) and System Control Block (SCB) hardware is consistent with the exception returns.

1. It is IMPLEMENTATION DEFINED whether a MemManage or UsageFault exception occurs when an EXC_RETURN value is treated as a branch address, and bit [0] of the value is clear.

Integrity checks are provided to check the following conditions on an exception return:

- The Exception Number being returned from (as held in the IPSR at the start of the return) must be listed in the SCB as being active.
- ~~If no exceptions other than the returning exception are active, the mode being returned to must be Thread mode. This checks for a mismatch of the number of exception returns.~~
- If at least one exception other than the returning exception is active, under normal circumstances the mode being returned to must be Handler mode. This checks for a mismatch of the number of exception returns. This check can be disabled using the NONBASETHRDENA control bit in the SCB.
- On return to Thread mode, the Exception Number restored into the IPSR must be 0.
- On return to Handler mode, the Exception Number restored into the IPSR must not be 0.
- The EXC_RETURN[3:0] must not be listed as reserved in Table B1-8 on page B1-26

An exception return error causes an INVPC UsageFault, with the illegal EXC_RETURN value in the link register (LR).

Exception return operation

For global declarations see *Register related definitions for pseudocode* on page B1-12.
For `ExceptionTaken()` see *Exception entry behavior* on page B1-21.
For a definition of ExceptionActive[*] ~~and NestedActivation~~ see *Reset behavior* on page B1-20.
For helper functions and procedures see *Miscellaneous helper procedures and functions* on page AppxG-22.

```
// ExceptionReturn()
// =================

ExceptionReturn(bits(28) EXC_RETURN)
    assert CurrentMode == Mode_Handler;
    if !IsOnes(EXC_RETURN<27:4>) then UNPREDICTABLE;

    integer ReturningExceptionNumber = UInt(IPSR<8:0>);

    if ExceptionActive[ReturningExceptionNumber] == '0' then
        DeActivate(ReturningExceptionNumber);
        UFSR.INVPC = '1';
        LR = 0xF0000000 + EXC_RETURN;
        ExceptionTaken(UsageFault);                  // returning from an inactive handler
        return;
    else
        case  EXC_RETURN<3:0> of
            when '0001'                              // return to Handler
                if NestedActivation == 1 then
                    DeActivate(ReturningExceptionNumber);
                    UFSR.INVPC = '1';
                    LR = 0xF0000000 + EXC_RETURN;
                    ExceptionTaken(UsageFault);      // return to Handler exception mismatch
```

```
            return;
        else
            frameptr = SP_main;
            CurrentMode = Mode_Handler;
            CONTROL<1> = '0';
    when '1001'                                 // returning to Thread using Main stack
        if NestedActivation != 1 && CCR.NONBASETHRDENA == '0' then
            DeActivate(ReturningExceptionNumber);
            UFSR.INVPC = '1';
            LR = 0xF0000000 + EXC_RETURN;
            ExceptionTaken(UsageFault);         // return to Thread exception mismatch
            return;
        else
            frameptr = SP_main;
            CurrentMode = Mode_Thread;
            CONTROL<1> = '0';
    when '1101'                                 // returning to Thread using Process stack
        if NestedActivation != 1 && CCR.NONBASETHRDENA == '0' then
            DeActivate(ReturningExceptionNumber);
            UFSR.INVPC = '1';
            LR = 0xF0000000 + EXC_RETURN;
            ExceptionTaken(UsageFault);         // return to Thread exception mismatch
            return;
        else
            frameptr = SP_process;
            CurrentMode = Mode_Thread;
            CONTROL<1> = '1';
    otherwise
        DeActivate(ReturningExceptionNumber);
        UFSR.INVPC = '1';
        LR = 0xF0000000 + EXC_RETURN;
        ExceptionTaken(UsageFault);             // illegal EXC_RETURN
        return;

DeActivate(ReturningExceptionNumber);
PopStack(frameptr);

if CurrentMode==Mode_Handler AND IPSR<8:0> =='000000000' then
    UFSR.INVPC = '1';
    PushStack();                                // to negate PopStack()
    LR = 0xF0000000 + EXC_RETURN;
    ExceptionTaken(UsageFault);                 // return IPSR is inconsistent
    return;
if CurrentMode==Mode_Thread AND IPSR<8:0> !='000000000' then
    UFSR.INVPC = '1';
    PushStack();                                // to negate PopStack()
    LR = 0xF0000000 + EXC_RETURN;
    ExceptionTaken(UsageFault);                 // return IPSR is inconsistent
    return;

ClearExclusiveLocal();
SetEventRegister()                              // see WFE instruction for more details
InstructionSynchronizationBarrier();
```

```
    if CurrentMode==Mode_Thread AND NestedActivation == 0 AND SCR.SLEEPONEXIT == '1' then
        PushStack();                                // to negate PopStack()
        SleepOnExit();                              // IMPLEMENTATION DEFINED
```

The DeActivate() and PopStack() pseudo-functions are defined as follows:

```
// DeActivate()
// ============

DeActivate(integer ReturningExceptionNumber)
    ExceptionActive[ReturningExceptionNumber] = '0';
    /* PRIMASK and BASEPRI unchanged on exception exit */
    if IPSR<8:0> != '000000010' then
        FAULTMASK<0> = '0';                 // clear FAULTMASK on any return except NMI
    NestedActivation = NestedActivation - 1;
    return;


// PopStack()
// ==========

PopStack(bits(32) frameptr)   /* only stack locations, not the load order, are architected */
    R[0]  = MemA[frameptr,4];
    R[1]  = MemA[frameptr+0x4,4];
    R[2]  = MemA[frameptr+0x8,4];
    R[3]  = MemA[frameptr+0xC,4];
    R[12] = MemA[frameptr+0x10,4];
    LR    = MemA[frameptr+0x14,4];
    PC    = MemA[frameptr+0x18,4];         // UNPREDICTABLE if the new PC not halfword aligned
    psr   = MemA[frameptr+0x1C,4];
    case  EXC_RETURN<3:0> of
        when '0001'                             // returning to Handler
            SP_main = (SP_main + 0x20) OR ZeroExtend((psr<9> AND CCR.STKALIGN):'00',32);
        when '1001'                             // returning to Thread using Main stack
            SP_main = (SP_main + 0x20) OR ZeroExtend((psr<9> AND CCR.STKALIGN):'00',32);
        when '1101'                             // returning to Thread using Process stack
            SP_process = (SP_process + 0x20) OR ZeroExtend((psr<9> AND CCR.STKALIGN):'00',32);
    APSR<31:27> = psr<31:27>;                   // valid APSR bits loaded from memory
    IPSR<8:0> = psr<8:0>;                       // valid IPSR bits loaded from memory
    EPSR<26:24,15:10> = psr<26:24,15:10>;       // valid EPSR bits loaded from memory
    return;
```

B1.5.9 Exceptions in single-word load operations

To support instruction replay, single-word load instructions must not update the destination register when a fault occurs during execution. By example, this allows replay of the following instruction:

```
LDR R0, [R2, R0];
```