TriCore™ TC1.8 architecture manual volume 1

32-bit microcontroller

Core architecture

Volume 1 (of 2)

About this document

Scope and purpose

The TriCore™ architecture manual describes the core architecture and instruction set for Infineon Technologies TriCore™ microcontroller architecture. TriCore™ is a unified, 32-bit microcontroller-DSP, single-core architecture optimized for real-time embedded systems.

This document has been written for system developers and programmers, and hardware and software engineers.

- Volume 1 provides a detailed description of the core architecture and system interaction
- Volume 2 gives a complete description of the TriCore™ instruction set including optional extensions for the Memory Management Unit (MMU) and Floating Point Unit (FPU)

It is important to note that this document describes the TriCore™ architecture, not an implementation. An implementation may have features and resources which are not part of the core architecture. The product documentation for that implementation will describe all implementation specific features.

When working with a specific TriCore™ based product always refer to the appropriate supporting documentation.

TriCore™ versions

There have been several versions of the TriCore™ architecture implemented in production devices.

- This document is specific to the version(s) identified on the cover page
- Information specific to a particular version of the architecture only, will be labeled as such

Additional documentation

For the latest documentation and additional TriCore™ information, please visit the TriCore™ home page at:

http://www.infineon.com/TriCore

The following additional documents are also available for download from the TriCore™ architecture and core section:

TriCore™ DSP optimization guide

TriCore™ EABI (Embedded ABI) user's manual

Text conventions

This document uses the following text conventions:

- The default radix is decimal
  - Hexadecimal constants are suffixed with a subscript letter 'H', as in: FFC_H
  - Binary constants are suffixed with a subscript letter 'B', as in: 111_B
- Register reset values are not generally architecturally defined, but require setting on start-up in a given implementation of the architecture. Only those reset values that are architecturally defined are shown in this document. Where no value is shown, the reset value is not defined. Refer to the documentation for a specific TriCore™ implementation
- Bit-field and bits in registers are in general referenced as 'Register name.Bit-field', for example PSW.IS. The Interrupt Stack Control bit of the PSW register
- Units are abbreviated as follows:
  - MHz = Megahertz
  - kBaud, kBit = 1000 characters/bits per second

  - MBaud, MBit = 1,000,000 characters/bits per second
  - KByte = 1024 bytes
  - MByte = 1048576 bytes of memory
  - GByte = 1,024 megabytes
- Data format quantities referenced are as follows:
  - Byte = 8-bit quantity
  - Half-word = 16-bit quantity
  - Word = 32-bit quantity
  - Double-word = 64-bit quantity
  - Quad-word = 128-bit quantity

In tables where register bit-fields are defined, the conventions shown below are used in this document.

Table 1 **Bit type abbreviations**

Abbreviation	Description
r	Read-only. The bit or bit-field can only be read
w	Write-only. The bit or bit-field can only be written
rw	The bit or bit-field can be read and written
h	The bit or bit-field can be modified by hardware (such as a status bit). ‘h’ can be combined with ‘rw’ or ‘r’ bits to form ‘rwh’ or ‘rh’ bits
-	Reserved field. Read value is undefined, must be written with 0

***Note**: In register layout tables, a ‘Reserved Field’ is indicated with ‘RES’ in the field column and ‘-’ in the type column.*

Table of contents

About this document 1

Table of contents 3

1 Architecture overview 11
1.1 Introduction 11
1.1.1 Feature summary 11
1.2 Programming model overview 12
1.2.1 Architectural registers 12
1.2.2 Data types 13
1.2.3 Memory model 14
1.2.4 Addressing modes 14
1.3 Tasks and contexts 14
1.4 Interrupt system 15
1.5 Trap system overview 15
1.6 Protection system 16
1.7 Core debug controller 16
1.8 TriCore™ coprocessor interface 17

2 Programming model 18
2.1 Data types 18
2.1.1 Boolean 18
2.1.2 Bit string 18
2.1.3 Byte 18
2.1.4 Signed fraction 18
2.1.5 Address 18
2.1.6 Signed and unsigned integers 19
2.1.7 IEEE 754-2019 single-precision floating-point number 19
2.1.8 IEEE 754-2019 double-precision floating-point number 19
2.2 Data formats 19
2.2.1 Alignment requirements 20
2.2.2 Byte ordering 21
2.3 Memory model 22
2.4 Semaphores and atomic operations 23
2.5 Data addressing modes 23
2.5.1 Absolute addressing 24
2.5.2 Base + offset addressing 24
2.5.3 Pre-increment and pre-decrement addressing 24
2.5.4 Post-increment and post-decrement addressing 25
2.5.5 Circular addressing 25
2.5.6 Bit-reverse addressing 26
2.5.7 Synthesized addressing modes 27

3 General purpose and system registers . . . 29
3.1 General purpose registers (GPRs) . . . 29
3.1.1 Data general purpose registers . . . 30
3.1.2 Address general purpose registers . . . 31
3.2 Program state information registers . . . 31
3.2.1 Program counter (PC) . . . 31
3.2.2 Program status word register (PSW) . . . 32
3.2.3 PSW user status bits . . . 35
3.2.4 Access privilege level control (I/O privilege) . . . 36
3.2.5 Previous context information and pointer register (PCXI) . . . 36
3.2.6 Previous protection register set register (PPRS) . . . 37
3.3 Stack management registers . . . 37
3.3.1 Address register A[10] (SP) . . . 38
3.3.2 Interrupt stack pointer register (ISP) . . . 38
3.4 Core configuration registers . . . 39
3.4.1 Core control register (CORECON) . . . 39
3.4.2 CPU identification register (CPU_ID) . . . 40
3.4.3 Core identification register (CORE_ID) . . . 41
3.4.4 Boot configuration register (BOOTCON) . . . 41
3.4.5 TriCore™ core configuration register (TCCON) . . . 42
3.5 Compatibility mode register (COMPAT) . . . 43
3.5.1 Compatibility mode register (COMPAT) . . . 43
3.6 Access control registers . . . 43
3.6.1 SIST mode access control register (SMACON) . . . 43
3.7 Interrupt registers . . . 44
3.8 Memory protection registers . . . 44
3.9 Trap registers . . . 44
3.10 Memory configuration registers . . . 44
3.11 Core performance counter registers . . . 44
3.12 Core debug controller registers . . . 44
3.13 Floating point registers . . . 45
3.14 Virtualization registers . . . 45
3.15 Accessing core special function registers (CSFRs) . . . 45
4 Tasks and functions . . . 46
4.1 Context types . . . 46
4.1.1 Context save area . . . 47
4.2 Task switching operation . . . 48
4.3 Context save areas (CSAs) and context lists . . . 49
4.4 Context switching with interrupts and traps . . . 50
4.5 Context switching for function calls . . . 51
4.6 Return target fetch (A[11] and PPRS) . . . 51

4.7 Fast function calls with FCALL/FRET 51
4.8 Context save and restore examples 52
4.8.1 Context save 52
4.8.2 Context restore 53
4.9 Context management registers 54
4.9.1 Registers 55
4.9.1.1 Free CSA list head pointer register (FCX) 55
4.9.1.2 Previous context pointer register (PCX) 55
4.9.2 Free CSA list limit pointer register (LCX) 56
4.9.2.1 Free CSA list limit pointer register (LCX) 56
4.10 Accessing CSA memory locations 57
4.11 Context save area placement 57

5 Interrupt system 58
5.1 General operation 58
5.1.1 Interrupt control register (ICR) 58
5.1.2 CPU operation on an interrupt request 58
5.2 Entering an interrupt service routine (ISR) 58
5.3 Exiting an interrupt service routine (ISR) 59
5.4 Interrupt vector table 59
5.5 Using the TriCore™ interrupt system 61
5.5.1 Spanning interrupt service routines across vector entries 61
5.5.2 Interrupt priority groups 61
5.5.3 Dividing ISRs into different priorities 62
5.5.4 Using different priorities for the same interrupt source 63
5.5.5 Interrupt control registers 63
5.5.5.1 Interrupt control register (ICR) 63
5.5.5.2 Base interrupt vector table pointer (BIV) 64

6 Trap system 66
6.1 Trap types 66
6.1.1 Synchronous traps 67
6.1.2 Asynchronous traps 68
6.1.3 Hardware traps 68
6.1.4 Software traps 68
6.1.5 Unrecoverable traps 68
6.2 Trap handling 68
6.2.1 Trap vector format 68
6.2.2 Accessing the trap vector table 69
6.2.3 Return address (RA) 69
6.2.4 Trap vector table 69
6.2.5 Entering a trap handler 69
6.3 Trap descriptions 70

6.3.1 MMU traps (trap class 0) 70
6.3.2 Internal protection traps (trap class 1) 71
6.3.3 Instruction errors (trap class 2) 71
6.3.4 Context management (trap class 3) 73
6.3.5 System bus and peripheral errors (trap class 4) 74
6.3.6 Assertion traps (trap class 5) 75
6.3.7 System call (trap class 6) 76
6.3.8 Non-maskable interrupt (trap class 7) 76
6.3.9 Debug traps 76
6.4 Exception priorities 76
6.5 Trap control registers 78
6.5.1 Base trap vector table pointer (BTV) 78
6.5.2 Program synchronous error trap register (PSTR) 79
6.5.3 Data synchronous error trap register (DSTR) 79
6.5.4 Data asynchronous error trap register (DATR) 79
6.5.5 Data error address register (DEADD) 80

7 Memory integrity error mitigation 81
7.1 Memory integrity error classification 81
7.2 Memory integrity error traps 81
7.2.1 Program memory integrity error (PIE) 81
7.2.2 Data memory integrity error (DIE) 81
7.3 Summary 82
7.4 Integrity error registers 82
7.4.1 Program integrity error trap register (PIETR) 82
7.4.2 Program integrity error address register (PIEAR) 82
7.4.3 Data integrity error trap register (DIETR) 83
7.4.4 Data integrity error address register (DIEAR) 83

8 Address map and memory configuration 84
8.1 Overview 84
8.2 Scratchpad RAM 84
8.3 Address segments and memory access types 84
8.3.1 Memory access types 84
8.3.1.1 Cacheable memory 85
8.3.1.2 Non-cacheable memory 85
8.3.1.3 Peripheral space 85
8.3.2 Speculation 85
8.3.3 Cacheability of segments 85
8.3.4 Default memory types for all segments 86
8.4 Memory configuration register definitions 86
8.4.1 Programmable memory access register-0 (PMA0) 86
8.4.2 Programmable memory access register-1 (PMA1) 87

8.4.3 Programmable memory access register-2 (PMA2) 87
8.4.4 Program memory configuration registers (PCON0, PCON1, PCON2) 88
8.4.4.1 Program memory configuration register PCON0 88
8.4.4.2 Program memory configuration registers PCON1 88
8.4.4.3 Program memory configuration registers PCON2 89
8.4.5 Data memory configuration registers (DCON0, DCON1, DCON2) 89
8.4.5.1 Data memory configuration register DCON0 89
8.4.5.2 Data memory configuration register DCON1 89
8.4.5.3 Data memory configuration register DCON2 90

9 Floating point unit (FPU) 91
9.1 Functional overview 91
9.2 IEEE 754-2019 compliance 92
9.2.1 IEEE 754-2019 half precision 92
9.2.1.1 IEEE 754-2019 half precision data format 92
9.2.1.2 IEEE 754-2019 half precision NaNs 92
9.2.1.3 Half precision restrictions 93
9.2.2 IEEE 754-2019 single precision 93
9.2.2.1 IEEE 754-2019 single precision data format 93
9.2.2.2 IEEE 754-2019 single precision NaNs 94
9.2.2.3 Subnormal numbers 94
9.2.2.4 Single precision underflow 95
9.2.3 IEEE 754-2019 double precision 95
9.2.3.1 IEEE 754-2019 double precision data format 95
9.2.3.2 IEEE 754-2019 double precision NaNs 96
9.2.3.3 Double precision subnormal numbers 96
9.2.3.4 Double precision underflow 96
9.2.4 Traps 97
9.2.5 Software routines 97
9.3 Rounding 99
9.3.1 Round to nearest with ties towards even 99
9.3.2 Round to nearest: subnormals and zero substitution 100
9.3.3 Round Towards ± ∞: subnormals and zero substitution 100
9.4 Exceptions 100
9.5 Asynchronous and synchronous traps 104
9.6 FPU CSFR Registers 106
9.6.1 FPU trap control register 106
9.6.2 FPU trapping instruction program counter register 107
9.6.3 FPU trapping instruction opcode register 107
9.6.4 FPU trapping instruction operand SRC1_L register 108
9.6.5 FPU trapping instruction operand SRC1_U register 109
9.6.6 FPU trapping instruction operand SRC2_L register 109
9.6.7 FPU trapping instruction operand SRC2_U register 109

9.6.8 FPU trapping instruction operand SRC3_L register 110
9.6.9 FPU trapping instruction operand SRC3_U register 110
9.6.10 FPU synchronous trap control register 111
9.6.11 FPU synchronous trapping instruction opcode register 112
9.6.12 PSW user status bits (FPU view) 113

10 Memory protection system 114
10.1 Memory protection subsystems 114
10.2 Range based memory protection 114
10.2.1 Access permissions for intersecting memory ranges 116
10.2.2 Crossing protection boundaries 116
10.3 Using the range based memory protection system 116
10.3.1 Protection enable bit 117
10.3.2 Protection set selection 117
10.3.3 Address range 117
10.3.4 Protection range register naming convention 118
10.3.5 Protection set enable register naming convention 118
10.4 Range based memory protection registers 118
10.4.1 Data protection range register lower bound 118
10.4.2 Data protection range register upper bound 119
10.4.3 Code protection range register lower bound 119
10.4.4 Code protection range register upper bound 120
10.4.5 Data protection read enable set configuration register 120
10.4.6 Data protection write enable set configuration register 121
10.4.7 Code protection execute enable set configuration register 121

11 Temporal protection system 123
11.1 Temporal protection timers 123
11.2 Temporal protection system registers 123
11.2.1 TPS timer register 123
11.2.2 TPS control register 124

12 Core Performance Measurement and Analysis 125
12.1 Performance counter registers 126
12.1.1 Counter control 126
12.1.2 CPU clock cycle count 127
12.1.3 Instruction count 127
12.1.4 Multi-count register 1 128
12.1.5 Multi-count register 2 128
12.1.6 Multi-count register 3 129

13 Core debug controller 130
13.1 Run control features 130
13.2 Debug events 132
13.2.1 External debug event 132

13.2.2 Debug instruction ... 132
13.2.3 MTCR, MFCR, MTDCR and MFDCR instructions ... 132
13.2.4 Trigger event unit ... 133
13.3 Debug triggers ... 134
13.3.1 Combining debug triggers ... 134
13.3.2 Accumulated debug trigger information ... 134
13.3.3 Debug support for virtualization ... 134
13.4 Debug actions ... 135
13.4.1 Update Debug Status Register (DBGSR) ... 135
13.4.2 Indicate on core break-out signal ... 135
13.4.3 Indicate on core suspend-out signal ... 135
13.4.4 Halt ... 135
13.4.5 Breakpoint trap ... 136
13.4.6 Breakpoint interrupt ... 137
13.4.7 Suspend Out ... 138
13.4.8 Performance counter start/stop ... 138
13.4.9 None ... 138
13.4.10 Disabled ... 139
13.4.11 Suspend in halt ... 139
13.5 Priority of Debug Events ... 139
13.6 Call tracing ... 139
13.7 The debug control registers ... 140
13.8 Debug control registers - summary ... 141
13.9 Debug control registers ... 141
13.9.1 Debug Status Register ... 141
13.9.2 External Event Register ... 143
13.9.3 Core Register Access Event ... 144
13.9.4 Software Debug Event ... 144
13.9.5 Debug action Configuration ... 145
13.9.6 Trigger Event n ... 146
13.9.7 Trigger Address n ... 147
13.9.8 Trigger Accumulator ... 148
13.9.9 Debug Monitor Start Address ... 149
13.9.10 Debug Context Save Area Pointer ... 149
13.9.11 Debug Trap Control Register ... 149
13.9.12 Debug Configuration Register ... 150
13.10 Core tracing configuration ... 151
13.11 Trace configuration registers ... 151
13.11.1 Trace Configuration Register ... 151
13.11.2 Trace Filter ... 152
13.11.3 Trace Bandwidth Limiter ... 153
14 Real-time virtualization extension ... 155

14.1 Functional overview . . . 155
14.2 Definitions . . . 155
14.3 Virtualization register overview . . . 156
14.3.1 Virtualization control register 0 (VCON0) . . . 157
14.3.2 Virtualization control register 1 (VCON1) . . . 158
14.3.3 Virtualization configuration register 2 (VCON2) . . . 158
14.3.4 Base hypervisor cector table pointer (BHV) . . . 159
14.3.5 Virtual machine interrupt control register (VMn_ICR) . . . 159
14.3.6 Virtual machine pre-emption threshold (VMn_PETHRESH) . . . 160
14.4 Virtual machine handling . . . 161
14.4.1 Hypervisor and virtual machine switching operation . . . 161
14.4.2 Context save areas (CSAs) and context lists . . . 162
14.4.3 Context switch with hypervisor interrupts and hypervisor traps . . . 162
14.4.4 Considerations for starting a virtual machine . . . 163
14.4.5 Hypervisor context examples . . . 163
14.4.5.1 Transition from guest virtual machine to hypervisor . . . 163
14.4.5.2 Transition from hypervisor to guest virtual machine . . . 165
14.4.5.3 Switching execution from guest virtual machine in HRA to guest virtual machine in HRB . 166
14.4.5.4 Swapping execution from guest virtual machine x in HRB to guest virtual machine y in HRB . . . 169
14.5 Disabling virtualization . . . 172
14.6 Virtualization: Interrupt system . . . 172
14.6.1 Interrupt use cases . . . 174
14.6.2 Hypervisor interrupt entry sequence . . . 175
14.6.3 Exiting an interrupt service routine (ISR) . . . 176
14.6.4 Interrupts for the hypervisor . . . 176
14.7 Hypervisor trap system . . . 176
14.7.1 Hypervisor trap types . . . 176
14.7.2 Hypervisor trap handling . . . 177
14.7.3 Hypervisor trap descriptions . . . 179
14.7.4 Exception priorities (virtualization present and enabled) . . . 180
14.8 Virtualization changes to trap system . . . 182
14.9 HRHV memory protection unit . . . 183
14.9.1 Using the range based level 2 memory protection system . . . 184

15 Core register table . . . 186

Revision history . . . 194

Disclaimer . . . 196

1 Architecture overview

This chapter gives an overview of the TriCore™ architecture.

1.1 Introduction

TriCore™ is the first unified, single-core, 32-bit microcontroller-DSP architecture optimized for real-time embedded systems. The TriCore™ Instruction Set Architecture (ISA) combines the real-time capability of a microcontroller, the computational power of a DSP, and the high performance/price features of a RISC load/store architecture, in a compact re-programmable core.

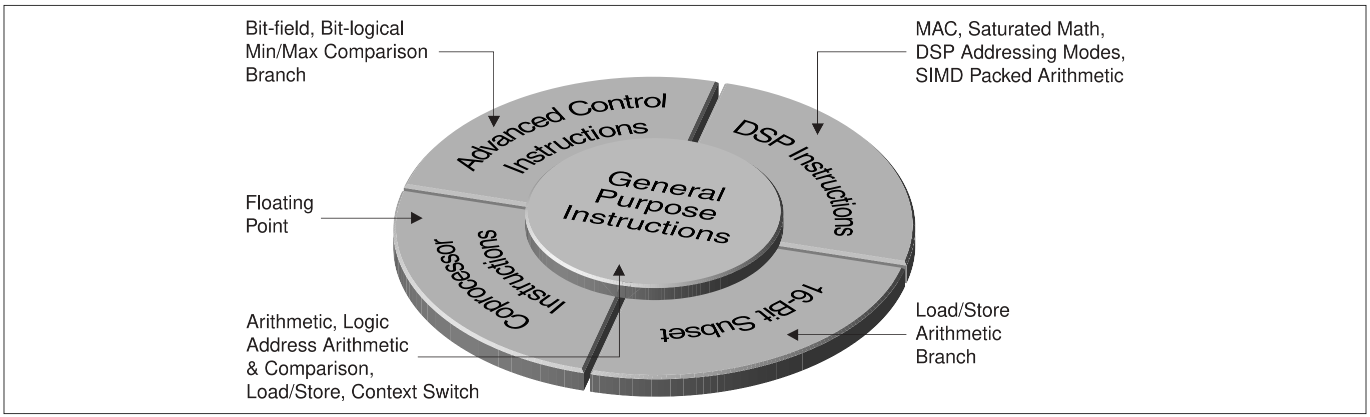


Figure 1 TriCore™ architecture overview

The ISA supports a uniform, 32-bit address space, with optional virtual addressing and memory-mapped I/O. The architecture allows for a wide range of implementations, ranging from scalar through to superscalar, and is capable of interacting with different system architectures, including multiprocessing. This flexibility at the implementation and system levels allows for different trade-offs between performance and cost at any point in time.

The architecture supports both 16-bit and 32-bit instruction formats. All instructions have a 32-bit format. The 16-bit instructions are a subset of the 32-bit instructions, chosen because of their frequency of use. These instructions significantly reduce code space, lowering memory requirements, system and power consumption.

Real-time responsiveness is largely determined by interrupt latency and context-switch time. The high-performance architecture minimizes interrupt latency by avoiding long multi-cycle instructions and by providing a flexible hardware-supported interrupt scheme. The architecture also supports fast-context switching.

1.1.1 Feature summary

The key features of the TriCore™ Instruction Set Architecture (ISA) are:

- 32-bit architecture
- 4 GBytes of address space
- 16-bit and 32-bit instructions for reduced code size
- Branch instructions (using branch prediction)
- Low interrupt latency with fast automatic context switch
- Dedicated interface to application-specific coprocessors to allow the addition of customized instructions
- Zero overhead loop capabilities
- Dual, single-clock-cycle, 16x16-bit multiply-accumulate unit (with optional saturation)
- Extensive bit handling capabilities

- Single Instruction Multiple Data (SIMD) packed data operations (2x16-bit or 4x8-bit operands)
- Flexible interrupt prioritization scheme
- Byte and bit addressing
- Little-endian byte ordering for memory and CPU registers
- Memory protection
- Debug support
- Optional single-precision Floating-Point Unit (FPU)
- Optional double-precision Floating-Point Unit (FPU)
- Optional Memory Management Unit (MMU)
- Optional extension for real-time virtualization

1.2 Programming model overview

This section covers aspects of the architecture that are visible to software:

- Architectural registers
- Data types
- Memory model
- Addressing modes

The programming model is described in detail in the chapter Programming model.

1.2.1 Architectural registers

The architectural registers consist of:

- 32 General Purpose Registers (GPRs)
- Program Counter (PC)
- Three 32-bit registers containing status flags, previous execution information and protection information (PCXI - Previous Context Information register, PPRS - Previous Protection Register Set and PSW -Program Status Word)
- One 32-bit register containing a pointer to the free Context Save Area (CSA) list head (FCX - Free CSA list head pointer)

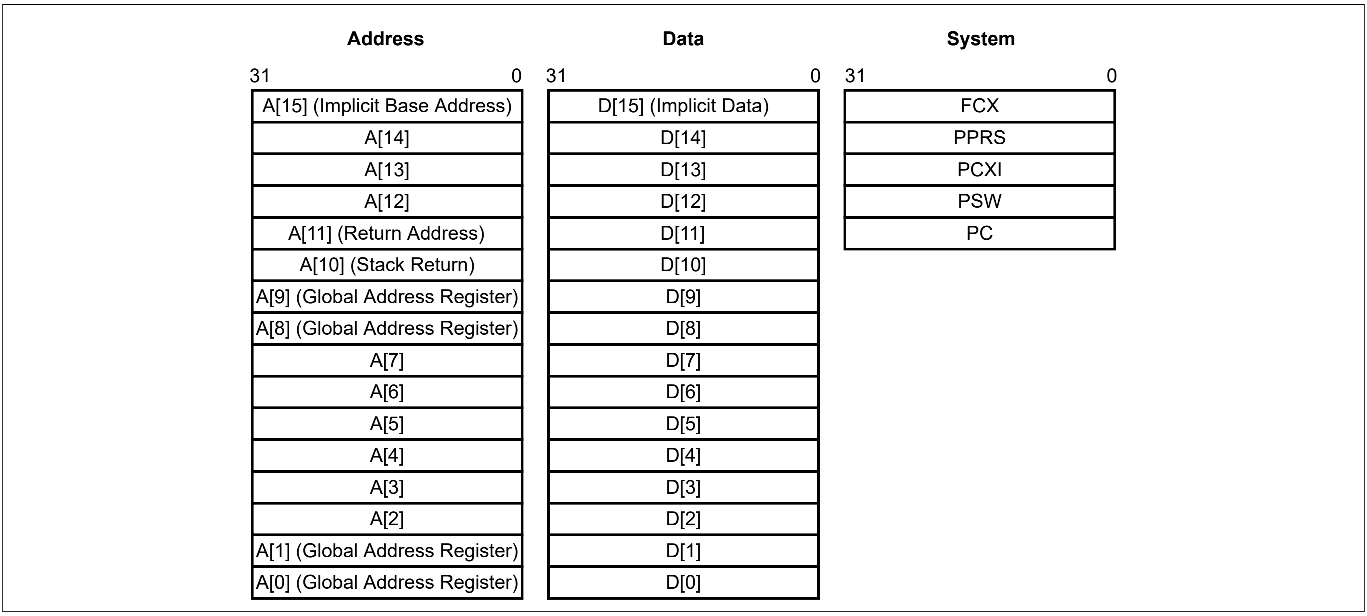


Figure 2 Architectural registers

The FCX, PCXI, PPRS, PSW and PC registers are crucial to the procedure for storing and restoring a task's context.

The 32 General Purpose Registers (GPRs) are divided into sixteen 32-bit data registers (D[0] through D[15]) and sixteen 32-bit address registers (A[0] through A[15]).

Four of the General Purpose Registers (GPRs) also have special functions:

- D[15] is used as an Implicit Data register
- A[10] is the Stack Pointer (SP) register
- A[11] is the Return Address (RA) register
- A[15] is the Implicit Address register

Registers [0_H - 7_H] are referred to as the 'lower registers' and registers [8_H - F_H] are called the 'upper registers'.

Registers A[0], A[1], A[8], and A[9] are defined as system global registers. These are not included in either the upper or lower context (see Tasks and functions) and are not saved and restored across calls or interrupts. They are normally used by the operating system to reduce system overhead.

In addition to the General Purpose Registers (GPRs), the core registers are composed of a number of Core Special Function Registers (CSFRs). See General purpose and system registers.

1.2.2 Data types

The instruction set supports operations on:

- Boolean
- Bit string
- Byte
- Signed fraction
- Address
- Signed/Unsigned integer
- IEEE 754-2019 single-precision floating-point
- IEEE 754-2019 double-precision floating-point

Most instructions work on a specific data type, while others are useful for manipulating several data types.

1.2.3 Memory model

The architecture can access up to 4 GBytes (address width is 32-bit) of unified program and I/O memory.

The address space is divided into 16 regions labeled as segments [0_H - F_H], each of 256 MBytes. The upper four bits of an address select the specific segment.

1.2.4 Addressing modes

Addressing modes allow load and store instructions to efficiently access simple data elements within data structures such as records, randomly and sequentially accessed arrays, stacks and circular buffers.

The TriCore™ architecture supports seven addressing modes. The simple data elements are 8-bit, 16-bit, 32-bit and 64-bit wide.

A subset of addressing is provided for larger data elements that are 128-bit wide and 16-word contexts.

These addressing modes support efficient compilation of C/C++ programs, easy access to peripheral registers and efficient implementation of typical DSP data structures (circular buffers for filters and bit-reversed indexing for Fast Fourier Transformations).

Addressing modes which are not directly supported in the hardware can be synthesized through short instruction sequences.

For more information see Synthesized addressing modes.

1.3 Tasks and contexts

A task is an independent thread of control. There are two types: Software Managed Tasks and Interrupt Service Routines (ISRs).

Software Managed Tasks are created through the services of a real-time kernel or Real-Time Operating System (RTOS), and are dispatched under the control of scheduling software. ISRs are dispatched by hardware in response to an interrupt. An ISR is the code that is invoked directly by the processor on receipt of an interrupt. Software Managed Tasks are sometimes referred to as user tasks, assuming that they execute in User Mode.

Each task is allocated its own mode, depending on the task's function:

- User-0 Mode: Used for tasks that do not access peripheral devices. This mode cannot enable or disable interrupts
- User-1 Mode: Used for tasks that access common, unprotected peripherals. Typically this would be a read or write access to serial port, a read access to timer, and most I/O status registers. Tasks in this mode may disable interrupts for a short period. (The interrupt and I/O permissions of this mode may be overridden by the core control register)
- Supervisor Mode: Permits read/write access to system registers and all peripheral devices. Tasks in this mode may disable interrupts or change the CPU's interrupt priority
- Hypervisor Mode: Permits read/write access to virtualization registers (when Virtualization is implemented)

The mode of a task is set through the I/O mode bits in the Processor Status Word (PSW).

A set of state elements are associated with any task, and these are known collectively as the task's context. The context is everything the processor needs to define the state of the associated task and enable its continued execution. This includes the CPU General Registers that the task uses, the task's Program Counter (PC), and its Program Status Information (PCXI and PSW). The architecture efficiently manages and maintains the context of the task through hardware. The context is subdivided into the upper context and the lower context.

Context save areas

The architecture uses linked lists of fixed-size Context Save Areas (CSAs). A CSA consists of 16 words of memory storage, aligned on a 16-word boundary. Each CSA can hold exactly one upper or one lower context. CSAs are linked together through a Link Word.

The architecture saves and restores context more quickly than conventional microprocessors and microcontrollers. The unique memory subsystem design with a wide data path allows the architecture to perform rapid data transfers between processor registers and on-chip memory.

Context switching occurs when an event or instruction causes a break in program execution. The CPU then needs to resolve this event before continuing with the program.

The events and instructions which cause a break in program execution are:

- Interrupt or service requests
- Traps
- Function calls
- Hypervisor traps (when Virtualization is implemented)

See Tasks and functions.

1.4 Interrupt system

A key feature of the architecture is its powerful and flexible interrupt system. The TriCore™ architecture supports being a Interrupt Service Provider and arbitration is built around programmable Service Request Nodes (SRNs).

A service request is a hardware indication from the system (or system on chip) containing the TriCore™ CPU indicating that an interrupt service requires the execution of an interrupt service routine (ISR). The service request might be to the CPU in its entirety or to a specific virtual machine when Virtualization is implemented. The system is responsible for routing requests to individual CPUs, or other types of interrupt service providers (such as DMAs). Service requests can originate from an on-chip peripheral, external hardware, or software.

Conventional architectures generally take a long time to service interrupt requests, and they are normally handled by loading a new Program Status from a vector table in data memory. In the TriCore™ architecture, a service request which is accepted commences the ISR by jumping to vectors in code memory to reduce response time. The entry code for the ISR is a block within a vector of code blocks. Each code block provides an entry for one interrupt source.

Interrupt priority

Service requests are prioritized, and prioritization allows for nested interrupts. The rules for prioritization are:

- A service request can interrupt the servicing of a lower priority interrupt
- Interrupt sources with the same priority cannot interrupt each other
- The system will implement an interrupt routing unit which will present to the CPU which interrupt has currently the highest priority number targeting that CPU as its service provider

All Service Requests are assigned Priority Numbers (SRPNs). Every ISR has its own priority number. Different service requests must be assigned different priority numbers.

The maximum number of interrupt priorities is 255. Programmable handling of priorities allows treating this differently for each system with extremes of range from one priority level with 255 sources, up to 255 priority levels with one source each.

Interrupt numbers are assumed to be assigned in linear order of interrupt priority. This is feasible because interrupt priorities are not hardwired to individual sources, but are assigned by software.

See Interrupt system.

1.5 Trap system overview

A trap occurs as a result of an event such as a Non-Maskable Interrupt (NMI), an instruction exception, a timing exception or illegal access. The TriCore™ architecture has eight trap classes and these traps are further classified as synchronous or asynchronous, hardware or software. Each distinguished set of traps is assigned a Trap Identification Number (TIN) that identifies the cause of the trap within its class. The entry code for the trap handler is a block within a vector of code blocks. Each code block provides an entry for one trap class. When a trap is taken, the TIN is placed in data register D[15].

The trap classes are:

- Reserved
- Internal protection
- Instruction error
- Context management
- System bus and peripherals
- Assertion trap
- System call
- Non-Maskable Interrupt (NMI)

See Trap system.

1.6 Protection system

One of the application domains that TriCore™ supports is safety-critical embedded systems. The architecture features a protection system designed to protect core system functionality from the effects of software errors in less critical application tasks, and to prevent unauthorized tasks from accessing critical system peripherals.

The protection system also facilitates debugging. It detects and traps errors that might otherwise go unnoticed until it is too late to identify the cause of the error.

The overall protection system is composed of four main subsystems:

1. The trap system: Described briefly in Trap system overview, and covered in detail in Trap system
2. The I/O privilege level: TriCore™ supports three I/O modes: User-0 mode, User-1 mode and Supervisor mode. The User-1 mode allows application tasks to directly access non-critical system peripherals. This allows embedded systems to be implemented efficiently, without the loss of security inherent in the common practice of running all code with Supervisor mode. (The interrupt and I/O permissions of this mode may be overridden by the core control register)
3. The memory protection system: This protection system provides control over which regions of memory a task is allowed to access, and what types of access it is permitted
4. The temporal protection system. This protection system provides protection against run-time overrun

For applications that require virtual memory, the optional Memory Management Unit (MMU) supports a familiar page-based model for memory protection. That model gives each memory page its own access permissions. The relatively conventional MMU design and the page-based memory protection model facilitate porting of standard operating systems that expect this model.

For applications that do not require virtual memory there is a range-based memory protection system. This system and its interaction with I/O privilege level for access to peripherals, is detailed in Memory protection system.

1.7 Core debug controller

The Core Debug Controller (CDC) is designed to support real-time systems that require non-intrusive debugging. Most of the architectural state in the CPU core and on-chip memories can be accessed through the system address map. The debug functionality is an interface of architecture, implementation and software tools.

Access to the CDC is typically provided through the On-Chip Debug Support (OCDS) of the system containing the CPU.

A general description of the core debug mechanism and registers is detailed in Core debug controller.

1.8 TriCore™ coprocessor interface

TriCore™ implementations may choose to implement optional coprocessors. The TriCore™ 1.8 architecture supports the standard coprocessor 0 operations and optionally the coprocessors 1 and 2 for single and double precision floating point operations.

2 Programming model

This chapter discusses the following aspects of the TriCore™ architecture that are visible to software:

- Supported Data types
- Data formats in registers and memory
- The Memory model
- Data addressing modes

2.1 Data types

The instruction set supports operations on the following data types:

- Boolean
- Bit string
- Byte
- Signed fraction
- Address
- Signed and unsigned integers
- IEEE 754-2019 single-precision floating-point number
- IEEE 754-2019 double-precision floating-point number

Most instructions operate on a specific data type, while others are useful for manipulating several data types.

2.1.1 Boolean

A Boolean is either TRUE or FALSE:

- TRUE is the value one (1) when generated and non-zero when tested
- FALSE is the value zero (0)

Booleans are produced as the result in comparison and logic instructions, and are used as source operands in logical, conditional add, subtract, select and move operations as well as conditional jump instructions.

2.1.2 Bit string

A bit string is a packed field of bits.
Bit strings are produced and used by logical, shift, and bit-field instructions.

2.1.3 Byte

A byte is an 8-bit value that can be used for a character or a very short integer. No specific coding is assumed.

2.1.4 Signed fraction

The architecture supports 16-bit, 32-bit and 64-bit signed fractional data for DSP arithmetic. Data values in this format have a single high-order sign bit, where 0 represents positive (+) and 1 represents negative (-), followed by an implied binary point and fraction. Their values are therefore in the range [-1,1).

2.1.5 Address

An address is a 32-bit unsigned value.

2.1.6 Signed and unsigned integers

Signed and unsigned integers are normally 32 bits. Shorter signed or unsigned integers are sign-extended or zero-extended to 32 bits when loaded from memory into a register.

Multi-precision integers

Multi-precision integers are supported with addition and subtraction using carry. Integers are considered to be bit strings for shifting and masking operations. Multi-precision shifts can be made using a combination of single-precision shifts and bit-field extracts.

2.1.7 IEEE 754-2019 single-precision floating-point number

Depending on the particular implementation of the core architecture, IEEE 754-2019 floating-point numbers are supported by coprocessor hardware instructions or by software calls to a library.

2.1.8 IEEE 754-2019 double-precision floating-point number

Depending on the particular implementation of the core architecture, IEEE 754-2019 double-precision floating-point numbers are supported by coprocessor hardware instructions or by software calls to a library.

2.2 Data formats

All general purpose registers (GPRs) are 32 bits wide, and most instructions operate on word (32-bit) values. When byte or half-word data elements are loaded from memory, they are automatically sign-extended or zero-extended to fill the register. The type of filling is implicit in the load instruction. For example, LD.B to load a byte with sign extension, or LD.BU to load a byte with zero extension.

The supported data formats are:

- Bit
- Byte: signed, unsigned
- Half-word: signed, unsigned, fraction
- Word: signed, unsigned, fraction, floating-point
- 48-bit: signed, unsigned, fraction
- Double-word: signed, unsigned, fraction, floating-point
- Quad-word: signed

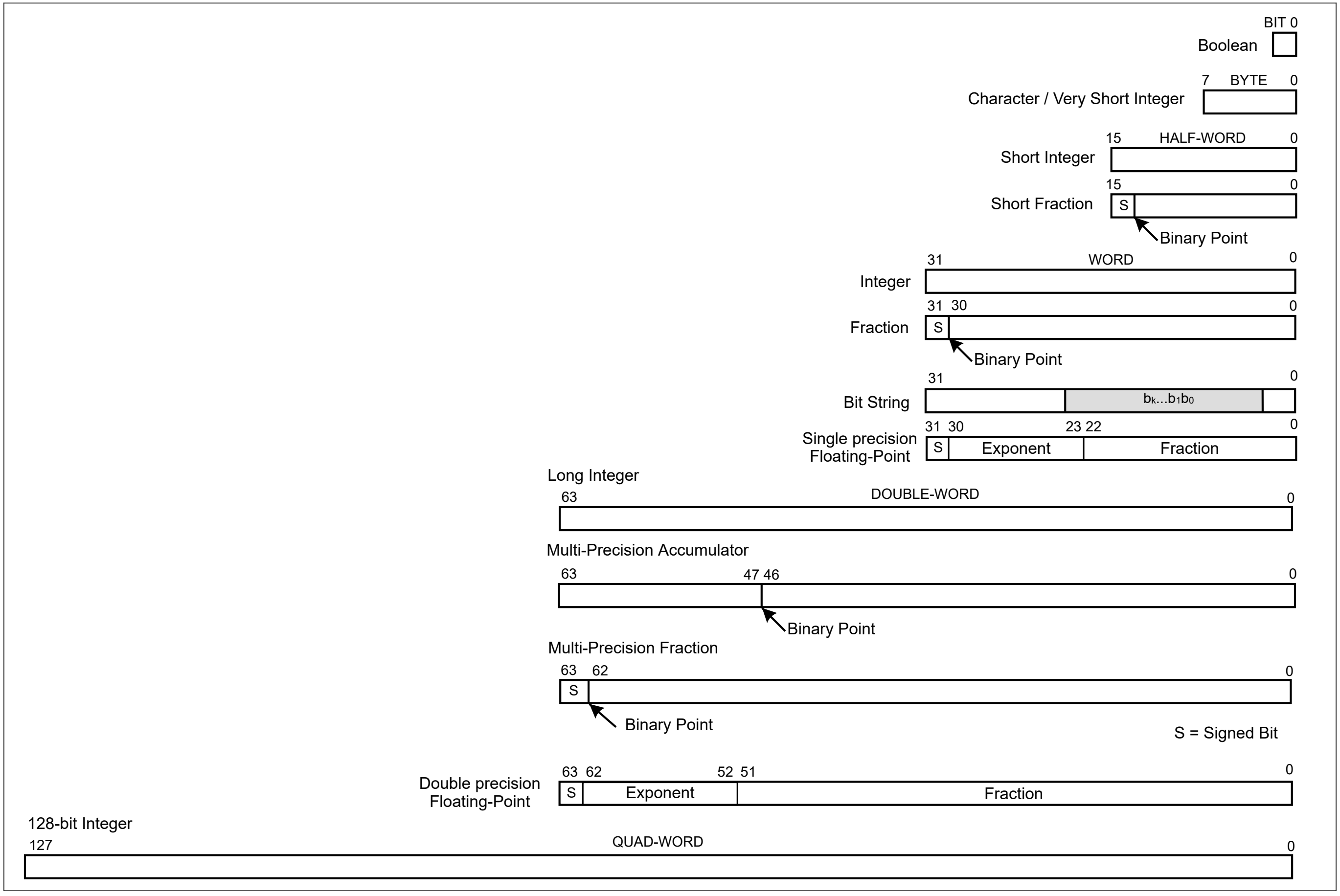


Figure 3 **Supported data formats**

2.2.1 Alignment requirements

Alignment requirements differ for addresses and data (see Table 2). Address variables loaded into or stored from address registers, must always be word-aligned.

Most data can be aligned on any half-word boundary, regardless of size, except where noted below. This facilitates the use of packed arithmetic operations in DSP applications, by allowing two or four packed 16-bit data elements to be loaded or stored together on any half-word boundary.

Programming restrictions

There are some restrictions to the half-word alignment of which programmers must be aware, specifically:

- Byte operations LD.B, LD.BU, ST.B, and ST.T may be byte aligned
- The CMPSWAP.W, LDMST, SWAPMSK.W and SWAP.W instructions require their operands to be word-aligned
- The LD.DD and ST.DD instructions required their operand to be word-aligned
- All accesses to peripheral space must be naturally aligned

Alignment rules

Table 2 **Alignment rules for non-peripheral space**

Access type	Access size	Alignment of address in memory
Load, Store Data Register	Byte	Byte (1_H)

(table continues...)

Table 2 (continued) Alignment rules for non-peripheral space

Access type	Access size	Alignment of address in memory
	Half-word	2 bytes (2_H)
	Word	2 bytes (2_H)
	Double-word	2 bytes (2_H)
	Quad-word	4 bytes (4_H)
Load, Store Address Register	Word	4 bytes (4_H)
	Double-word	4 bytes (4_H)
CMPSWAP.W, LDMST, SWAPMSK.W, SWAP.W	Word	4 bytes (4_H)
ST.T	Byte	Byte (1_H)
Context Load/Store/Restore/Save	16 x 32-bit registers	64 bytes (40_H)

Table 3 Alignment rules for peripheral space

Access type	Access size	Alignment of address in memory
Load, Store Data Register	Byte	Byte (1_H)
	Half-word	2 bytes (2_H)
	Word	4 bytes (4_H)
	Double-word	8 bytes (8_H)
	Quad-word	16 bytes (10_H)
Load, Store Address Register	Word	4 bytes (4_H)
	Double-word	8 bytes (8_H)
CMPSWAP.W, LDMST, SWAPMSK.W, SWAP.W	Word	4 bytes (4_H)
ST.T	Byte	Byte (1_H)
Context Load/Store/Restore/Save	16 x 32-bit registers	Not Permitted

2.2.2 Byte ordering

The data memory and CPU registers store data in little-endian byte order (the least-significant bytes are at lower addresses). The following figure illustrates byte ordering. Little-endian memory referencing is used consistently for data and instructions.

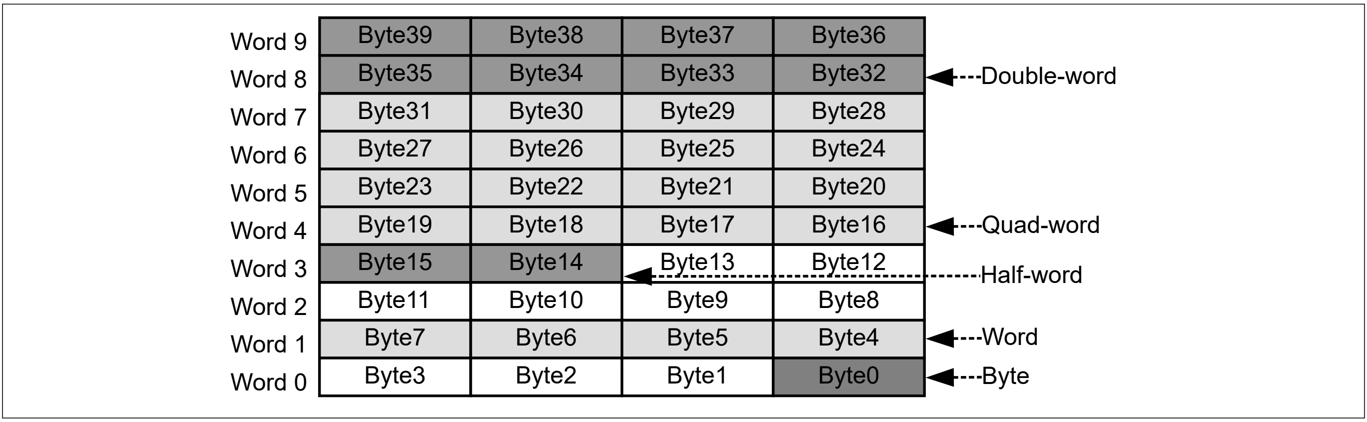

Word 9
Word 8
Word 7
Word 6
Word 5
Word 4
Word 3
Word 2
Word 1
Word 0
Byte39 Byte38 Byte37 Byte36
Byte35 Byte34 Byte33 Byte32
Byte31 Byte30 Byte29 Byte28
Byte27 Byte26 Byte25 Byte24
Byte23 Byte22 Byte21 Byte20
Byte19 Byte18 Byte17 Byte16
Byte15 Byte14 Byte13 Byte12
Byte11 Byte10 Byte9 Byte8
Byte7 Byte6 Byte5 Byte4
Byte3 Byte2 Byte1 Byte0
Double-word
Quad-word
Half-word
Word
Byte

Figure 4 **Byte ordering**

2.3 Memory model

The architecture has an address width of 32 bits and can access up to 4 GBytes of memory. The address space is divided into 16 regions or segments, [0_H - F_H]. Each segment is 256 MBytes. The upper 4 bits of an address select the specific segment. The first 16 KBytes of each segment can be accessed for loading and storing data using absolute addressing, or the first 2 MBytes of each segment can be accesses as code for calls or jumps using absolute addressing.

Many data accesses use addresses computed by adding a displacement to the value of a base address register. Using a displacement to cross one of the segment boundaries is not allowed and if attempted causes a MEM trap. This restriction allows direct determination of the accessed segment from the base address.

See Trap system for more information on traps.

Physical memory addresses

Physical memory addresses in segment E_H and F_H are guaranteed to be peripheral space and therefore all accesses are non-speculative and are not accessible to User-0 mode.

The Core Special Function Registers (CSFRs) are mapped to a 64 KBytes region in the system address map. The base location of this 64 KBytes region is implementation-dependent. When virtualization is present, each hardware resource is mapped to individual 64 KBytes contiguous region in the system address map.

Segments 8_H to D_H have further limitations placed upon them in some implementations. For example, specific segments for program and data may be defined by device-specific implementations. Other details of the segment mapping are implementation-specific.

Physical memory attributes

The physical memory attributes of segments zero to seven are implementation dependent.

For more information see Address segments and memory access types.

Table 4 **Physical address space**

Address	Segments	Description
FFFF $FFFF_H$: E000 0000_H	E_H - F_H	Peripheral space
DFFF $FFFF_H$: 8000 0000_H	8_H - D_H	Detailed limitations are implementation specific
7FFF $FFFF_H$: 0000 0000_H	0_H - 7_H	Implementation dependent

2.4 Semaphores and atomic operations

The following instructions read then write memory atomically. The system will guarantee no other master may modify the addressed location between the two accesses.

- LDMST (Load, Modify, Store)
- SWAP.W (Swap register with memory)
- ST.T (Store bit)
- CMPSWAP.W (Compare and swap)
- SWAPMSK.W (Swap under mask)

LDMST uses a mask register to write selected bits from a source register into a memory word. However it does not return a value, so it can not be used as an atomic "test and set" type operations for binary semaphores. The SWAPMSK.W is provided for this purpose.

The CMPSWAP.W instruction conditionally swaps a source register with a memory word. The SWAPMSK.W instructions swaps through a mask the contents of a source register with a memory word.

When memory protection is enabled, the effective address of the LDMST, CMPSWAP.W, SWAPMSK.W, SWAP.W or ST.T instruction must lie within a range which has both read and write permissions enabled.

The execution of an atomic instruction forces the completion of all data accesses semantically ahead of the instruction. This ensures that any buffered state is written to memory prior to the atomic operation.

2.5 Data addressing modes

Addressing modes allow load and store instructions to access simple data elements such as records, randomly and sequentially accessed arrays, stacks, and circular buffers.

The simple data elements are 8-bit, 16-bit, 32-bit, 64-bit or 128-bit wide. The architecture supports seven addressing modes.

The addressing modes support efficient compilation of C/C++, give easy access to peripheral registers, and efficient implementation of typical DSP data structures (circular buffers for filters and bit-reversed indexing for FFTs).

Table 5 Addressing modes

Addressing mode	Address register use
Absolute	None
Base + short offset	Address register
Base + long offset	Address register
Pre-increment	Address register
Post-increment	Address register
Circular	Address register pair
Bit-reverse	Address register pair

Addressing modes which are not directly supported in the hardware can be synthesized through short instruction sequences.

For more information see Synthesized addressing modes.

Instruction formats

The instruction formats provide as many bits of address as possible for absolute addressing, and as large a range of offsets as possible for base + offset addressing.

It is possible for an address register to be both the target of a load and an update associated with a particular addressing mode. In the following case for example, the contents of the address register are not architecturally defined:

```
ld.a a0, [a0+]4
```

Similarly, consider the following case:

```
st.a [+a0]4, a0
```

It is not architecturally defined whether the original or updated value of A[0] is stored into memory. This is true for all addressing modes in which there is an update of the address register.

2.5.1 Absolute addressing

Absolute addressing is useful for referencing I/O peripheral registers and global data.

Absolute addressing uses an 18-bit constant specified by the instruction as the memory address. The full 32-bit address results from moving the most significant 4 bits of the 18-bit constant specify the segment number of the 32-bit address (Figure 5). The remaining address bits are zero-filled.

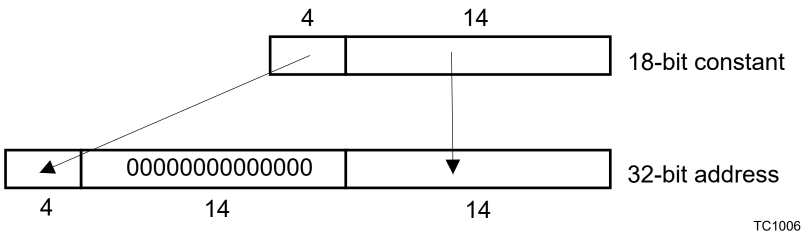


Figure 5 **Translation of absolute address to full effective address**

2.5.2 Base + offset addressing

Base + offset addressing is useful for referencing record elements, local variables (using Stack Pointer (SP) as the base), and static data (using an address register pointing to the static data area). The full effective address is the sum of an address register and the sign-extended 10-bit offset.

A subset of the memory operations are provided with a Base + Long Offset addressing mode. In this mode the offset is a 16-bit sign-extended value. This allows any location in memory to be addressed using a two instruction sequence [1].

2.5.3 Pre-increment and pre-decrement addressing

Pre-increment and pre-decrement addressing (where pre-decrement addressing is obtained by the use of a negative offset), may be used to push onto an upward or downward-growing stack, respectively.

The pre-increment addressing mode uses the sum of the address register and the offset both as the effective address and as the value written back into the address register.

[1] Accesses to the last 32-KByte section at the end of a segment is not possible and leads to a MEM trap

2.5.4 Post-increment and post-decrement addressing

Post-increment and post-decrement addressing (where post-decrement addressing is obtained by the use of a negative offset), may be used for forward or backward sequential access of arrays respectively. Furthermore, the two versions of the mode may be used to pop from a downward-growing or upward-growing stack, respectively.

The post-increment addressing mode uses the value of the address register as the effective address and then updates this register by adding the 10-bit offset to its previous value.

2.5.5 Circular addressing

The primary use of circular addressing (Figure 6) is for accessing data values in circular buffers while performing filter calculations.

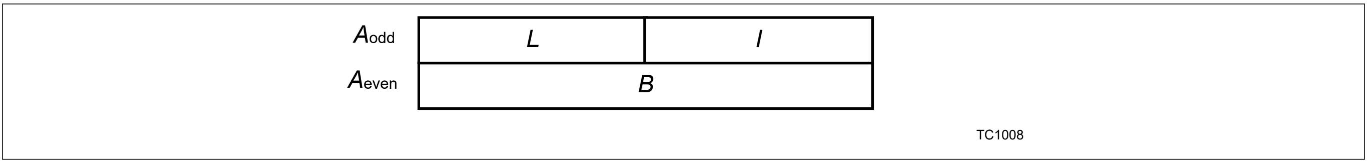


Figure 6 Circular addressing mode

Circular addressing uses an address register pair to hold the required state:

- The even register is always a base address of the circular buffer (B)
- The least significant half holds the byte-index into the buffer (I)
- The most significant half of the odd register is the buffer size (L) in bytes
- The effective address is (B+I)
- The buffer occupies memory from addresses B to B+L-1

The index is post-incremented using the following algorithm:

```
tmp = I + sign_ext(offset10);
if (tmp < 0)
    I = tmp + L;
else if (tmp >= L)
    I = tmp - L;
else
    I = tmp;
```

TC1009

Figure 7 Circular addressing index algorithm

The 10-bit offset is specified in the instruction word and is a byte-offset that can be either positive or negative. Note that correct ‘wrap around’ behavior is guaranteed as long as the magnitude of the offset is smaller than the size of the buffer.

To illustrate the use of circular addressing, consider a circular buffer consisting of 25, 16-bit values. If the current index is 48, then the next item is obtained using an offset of two (2-byte per value). The new value of the index ‘wraps around’ to zero. If we are at an index of 48 and use an offset of four, the new value of the index is two. If the current index is four and we use an offset of -8, then the new index is 46 (4-8+50).

In the end case, where a memory access runs off the end of the circular buffer (Figure 8), the data access also wraps around to the start of the buffer. For example, consider a circular buffer containing n+1 elements where each element is a 16-bit value. If a load word is performed using the circular addressing mode and the effective address of the operation points to element n, the 32-bit result contains element n in the bottom 16 bits and element 0 in the top 16 bits.

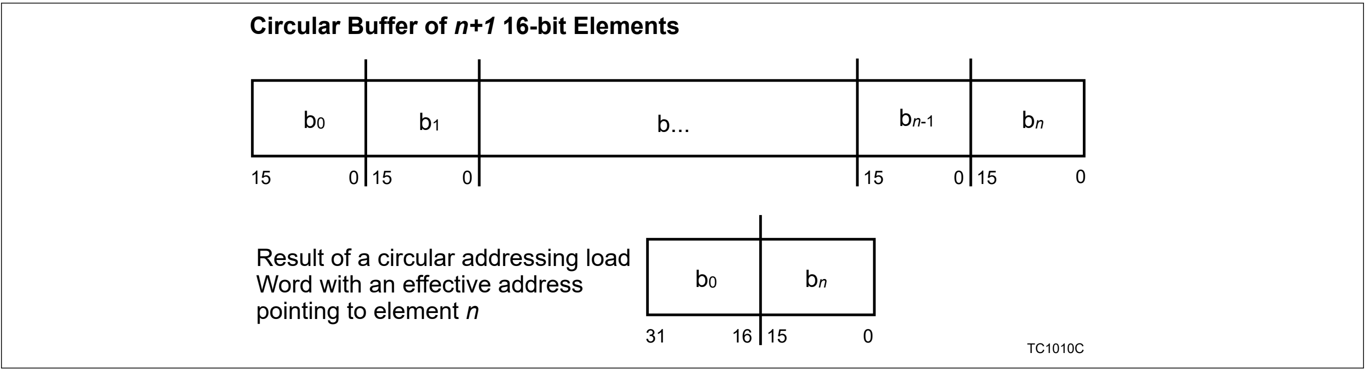

Circular Buffer of n+1 16-bit Elements
b0
b1
b...
bn-1
bn
15
0
15
0
15
0
15
0
Result of a circular addressing load Word with an effective address pointing to element n
b0
bn
31
16
15
0
TC1010C

Figure 8 Circular buffer end case

The size and length of a circular buffer has the following restrictions:

- The start of the buffer must be aligned to a 64-bit boundary. An implementation is free to advise the programmer of optimal alignment of circular buffers, but must support alignment to the 64-bit boundary
- The length of the buffer must be a multiple of the data size, where the data size is determined from the instruction being used to access the buffer. For example, a buffer accessed using a load-word instruction must be a multiple of 4 bytes in length, and a buffer accessed using a load double-word instruction must be a multiple of 8 bytes in length

If these restrictions are not met the implementation takes an alignment trap (ALN). An alignment trap is also taken if the index (I) >= length (L).

Accesses to peripheral space using circular addressing are not permitted. Such accesses will result in a MEM trap.

2.5.6 Bit-reverse addressing

Bit-reverse addressing is used to access arrays used in FFT algorithms. The most common implementation of the FFT ends with results stored in bit-reversed order (Figure 9).

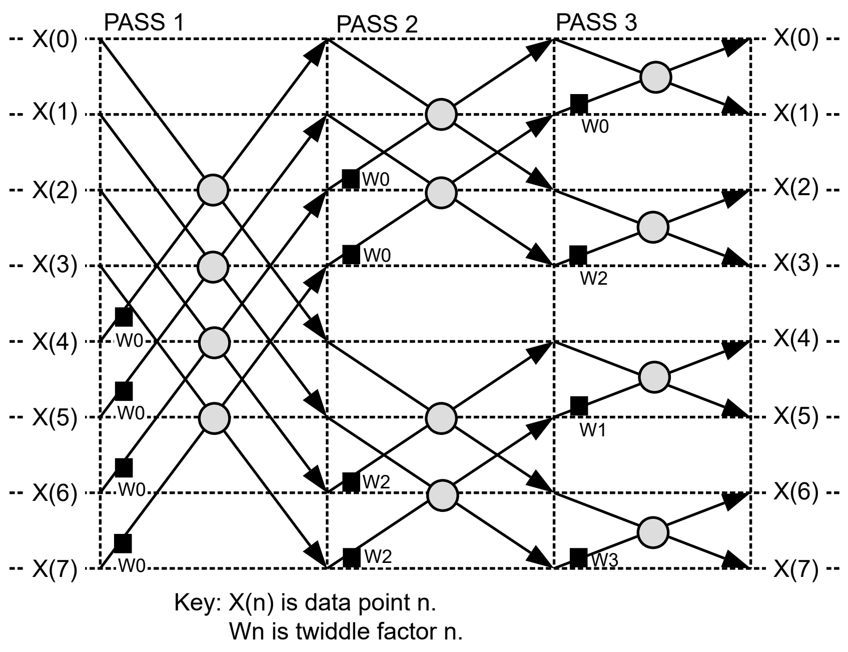


Figure 9 Bit-reverse addressing

Bit-reverse addressing uses an address register pair to hold the required state:

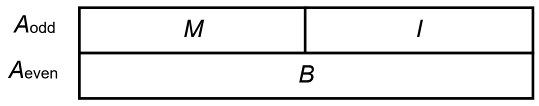


Figure 10 **Register pair for bit-reverse addressing**

- The even register is the base address of the array (B)
- The least-significant half of the odd register is the index into the array (I)
- The most-significant half is the modifier (M), used to update I after every access
- The effective address is B+I
- The index, I, is post-incremented and its new value is reverse [reverse (I) + reverse (M)][2)]

To illustrate for a 1024 point real FFT using 16-bit values, the buffer size is 2048 bytes. Stepping through this array using a bit-reverse index would give the sequence of byte indices: 0, 1024, 512, 1536, and so on. This sequence can be obtained by initializing I to 0 and M to 0400_H.

Table 6 **1024-point FFT using 16-bit values**

I (decimal)	I (binary)	Reverse(I)	Rev[Rev(I) + Rev(M)]
0	0000000000000000_B	0000000000000000_B	0000010000000000_B
1024	0000010000000000_B	0000000000100000_B	0000001000000000_B
512	0000001000000000_B	0000000001000000_B	0000011000000000_B
1536	0000011000000000_B	0000000001100000_B	0000000100000000_B

The required value of M is given by; buffer size/2, where the buffer size is given in bytes.

2.5.7 Synthesized addressing modes

This section describes how addressing that is not directly supported in the instruction addressing modes, can be synthesized through short instruction sequences.

Indexed addressing

The indexed addressing mode can be synthesized using the ADDSC.A instruction (Add Scaled Index to Address), which adds a scaled data register to an address register. The scale factor can be 1, 2, 4 or 8 for addressing indexed arrays of bytes, half-words, words, or double-words.

Bit indexed addressing

To support addressing of indexed bit arrays, the ADDSC.AT instruction scales the bit-index value by 1/8 (shifts right 3 bits) and adds it to the address register.

The two low-order bits of the resulting byte address are cleared to give the address of the word containing the indexed bit.

To extract the bit, the word in which it is contained, is loaded. The bit index is then used in an EXTR.U instruction. A bit-field, beginning at the indexed bit position, can also be extracted.

To store a bit or bit-field at an indexed bit position, ADDSC.AT is used in conjunction with the LDMST (Load/Modify/Store) instruction.

[2] The reverse(I) function exchanges bit n with bit (15–n) for n = 0, ...7

Program counter relative addressing

Program counter relative (PC relative) addressing is the normal mode for branches and calls. However the architecture does not directly support PC relative addressing of data. This is because the separate on-chip instruction and data memories make data access to the program memory more costly in time than the data memory.

When PC relative addressing of data is required, the address of a nearby code label is placed into an address register and used as a base register in base + offset mode to access the data. Once the base register is loaded it can be used to address other PC relative data items nearby.

A code address can be loaded into an address register in various ways. If the code is statically linked, then the absolute address of the code label is known and can be loaded using the LEA instruction (Load Effective Address), or with a sequence to load an extended absolute address.

For position-independent code, the appropriate way to load a code address for use in PC relative code or data addressing is to use the JL (Jump and Link) instruction. A jump and link to the next instruction is executed, placing the address of that instruction into the return address (RA) register A[11]. Before this is done though, it is necessary to save the actual return address of the current function, and then restore it before returning.

The JRI (Jump Relative Indirect) instruction provides support for position-independent code. A relative jump to the target instruction is executed, with an offset contained in an address register. The target instruction address is computed by adding the offset of the target instruction (allowing a full 32-bit range) to the program counter (PC). The offset could be computed or loaded from a table of relative offsets. This provides supports of both switch jump tables, and some table schemes for multi-section code linking either statically or dynamically.

3 General purpose and system registers

There are two types of core registers, the general purpose registers (GPRs) and the core special function registers (CSFRs). The GPRs consist of 16 general purpose data and 16 general purpose address registers. The CSFRs control the operation of the core and provide status information about the core.

- General purpose registers
- System registers (PSW, PC, PCXI, PPRS)
- Stack management registers (A[10] and ISP)
- CORECON and CPU_ID registers
- Trap registers
- Context management registers
- Memory protection registers
- Memory management registers
- Debug registers
- Floating point registers
- Virtualization registers
- Special function registers associated with the core

Reset values

It should be noted that because this manual describes the TriCore™ architecture, not an implementation of that architecture, some reset values are not given. Where they are not given, the values are implementation specific.

Initialization protection

The architecture supports the concept of an initialization state prior to an operational state.

When in the initialization state, all core special function registers can be modified, using the MTCR instruction for single CSFRs, or using the MTDCR for CSFR pairs. In the operational state only a subset of CSFRs can be modified in this way. All other functions remain identical between these states.

CSFRs that are only writable in the initialization state are described as ENDINIT protected.

The transition between the initialization state and the operational state is controlled by the system implementation. This facility adds an extra level of protection to critical CSFRs by only allowing them to be changed in the initialization state.

The following registers are ENDINIT protected:

- BTV, BIV, ISP, PMA0, PMA1, PMA2, PCON0, DCON0, SEGEN, BHV

A safety specific version of ENDINIT protection is provided. The following registers are SAFETY_ENDINIT protected:

- SMACON, CORECON, COMPAT

3.1 General purpose registers (GPRs)

The General Purpose Registers (GPRs) are split evenly into:

- 16 data registers (DGPRs), D[0] to D[15]
- 16 address registers (AGPRs), A[0] to A[15]

The separation of data and address registers facilitates efficient implementations in which arithmetic and memory operations are performed in parallel. Several instructions allow the interchange of information between data and address registers (used for example, to create or derive table indexes). Two consecutive even-odd data registers can be concatenated to form eight extended-size registers (E[0], E[2], E[4], E[6], E[8], E[10], E[12], and E[14]), in order to support 64-bit values. Four consecutive data registers can be concatenated to form four quad-size registers (Q[0], Q[4], Q[8], and Q[12]). Two consecutive even-odd address registers can be concatenated to form eight paired address registers (P[0], P[2], P[4], P[6], P[8], P[10], P[12], and P[14]) containing circular buffer pointers and bit-reverse array pointers as well as just two addresses for efficient memory access.

Registers A[0], A[1], A[8], and A[9] are defined as system global registers. Their contents are not saved or restored across calls, traps or interrupts.

Register A[10] is used as the Stack Pointer (SP). See Stack management registers.

Register A[11] is used to store the Return Address (RA) for calls and linked jumps, and to store the return Program Counter (PC) value for interrupts and traps.

While the 32-bit instructions have unlimited use of the GPRs, many 16-bit instructions implicitly use A[15] as their address register and D[15] as their data register. This implicit use eases the encoding of these instructions into 16 bits.

Support of 64-bit data values is provided with the use of even-odd register pairs. In the assembler syntax these register pairs are either referred to as a pair of 32-bit registers (for example, D[9]/D[8]) or as an extended 64-bit register. For example, E[8] is the concatenation of D[9] and D[8], where D[8] is the least significant word of E[8].

Support of 128-bit data values is provided with the use of quad registers. For example, Q[4] is the concatenation of D[7], D[6], D[5] and D[4], where D[4] is the least significant word of Q[4].

In order to support extended addressing modes, an even-odd address register pair holds the extended address reference as a pair of 32-bit address registers (P[8] or A[8]/A[9] for example).

There are no separate floating-point registers. The data registers are used to perform floating-point operations. The floating-point data is saved and restored automatically using the fast context switch support.

The following figure shows the 32-bit wide GPRs.

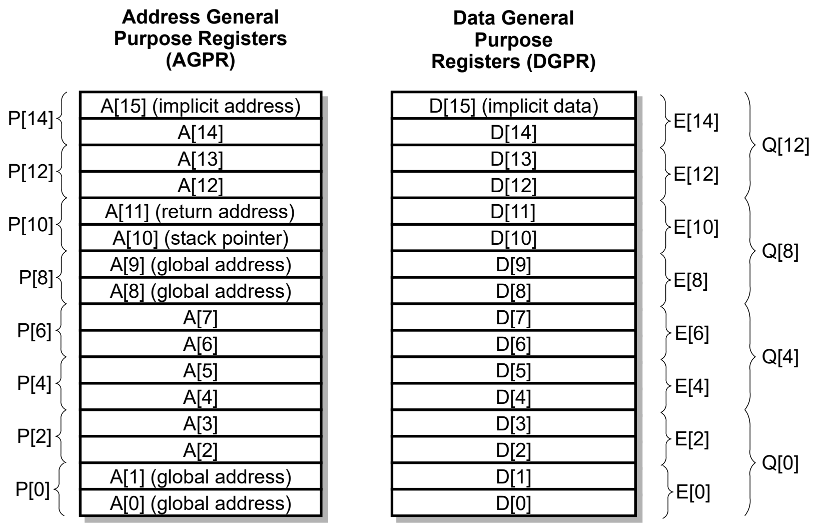


Figure 11 **General purpose registers (GPRs)**

The GPRs are an essential part of a task's context. When saving or restoring a task's context to and from memory the context is split into the upper and lower contexts:

- Registers A[2] to A[7] and D[0] to D[7] are part of the lower context
- Registers A[10] to A[15] and D[8] to D[15] are part of the upper context

***Note**: Upper and lower contexts are described in detail in Tasks and functions.*

3.1.1 Data general purpose registers

Dn (n=0-15) Address: $(FF00+n*4)_H$

Data register n Reset value: $\text{Implementation specific}_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DATA															
rw															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA															
rw															

Field	Bits	Type	Description
DATA	31:0	rw	Data register n value

3.1.2 Address general purpose registers

An (n=0-15) Address: $(FF80+n*4)_H$

Address register n Reset value: $\text{Implementation specific}_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ADDR															
rw															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDR															
rw															

Field	Bits	Type	Description
ADDR	31:0	rw	Address register n value

3.2 Program state information registers

The PC, PSW, PCXI, and PPRS registers hold and reflect program state information. These registers are an important part of storing and restoring a task's context, when the contents are stored, restored or modified during this process.

- PC: Program Counter
- PSW: Program Status Word
- PCXI: Previous Context Information
- PPRS: Previous Protection Register Set

3.2.1 Program counter (PC)

The 32-bit Program Counter (PC) shown below, holds the address of the instruction that is currently running. The Program Counter is part of a task's state information. The PC should only be written via the Core Debug Controller when the core is halted. If the core is not in halt a write will have no effect.

PC Address: $FE08_H$

Program counter register Reset value: Implementation $specific_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PC															
rw															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PC															**RES**
rw															-

Field	Bits	Type	Description
PC	31:1	rw	Program counter
RES	0	-	Reserved

3.2.2 Program status word register (PSW)

The Program Status Word register (PSW) is a 32-bit register that contains task-specific architectural state not captured in the General Purpose Register values. The lower half holds control values and parameters related to the protection system, including:

- The Protection Register Set (PRS)
- The I/O privilege level (IO)
- The Interrupt Stack flag (IS)
- The Global register Write permission flag (GW)
- The Call Depth Counter (CDC)
- The Call Depth Count Enable field (CDE)

PSW Address: $FE04_H$

Program status word Reset value: $0000\ 0B80_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
USB								**RES**							
rw								-							

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PRS2	**S**	**PRS1:0**		**IO**		**IS**	**GW**	**CDE**	**CDC**						
-	rw	rw		rw		rw	rw	rw	rw						

Field	Bits	Type	Description
USB	31:24	rw	User status bits The eight most significant bits of the PSW are designated as User Status Bits. These bits may be set or cleared as execution side effects of user instructions. Refer to the PSW User Status Bits section which follows this table
RES	23:16	-	Reserved

(table continues...)

(continued)

Field	Bits	Type	Description
PRS[2]	15	-	Protection register set bit[2] Selects the active Data and Code Memory Protection Register Set. The memory protection register values control load, store and instruction fetches within the current process. Up to eight sets are supported, the number of protection sets available is implementation dependent
S	14	rw	Alternate task identifier selector The task uses the alternate master tag identifier
PRS[1:0]	13:12	rw	Protection register set bits[1:0] Selects the active Data and Code Memory Protection Register Set. The memory protection register values control load, store and instruction fetches within the current process. Up to eight sets are supported, the number of protection sets available is implementation dependent
IO	11:10	rw	Access privilege level control (I/O privilege) Determines the access level to special function registers and peripheral devices. 00_B : User-0 mode No peripheral access. Access to memory regions with the peripheral space attribute are prohibited and results in a PSE or MPP trap. This access level is given to tasks that need not directly access peripheral devices. Tasks at this level do not have permission to enable or disable interrupts. 01_B : User-1 mode Regular peripheral access. Enables access to common peripheral devices that are not specially protected, including read/write access to serial I/O ports, read access to timers, and access to most I/O status registers. Tasks at this level may disable interrupts. The default behavior of this mode may be overriden by the core control register. 10_B : Supervisor mode Enables access to all peripheral devices. It enables read/write access to core registers and protected peripheral devices. Tasks at this level may disable interrupts. 11_B : Reserved value
IS	9	rw	Interrupt stack control Determines if the current execution thread is using the shared global (interrupt) stack or a user stack. 0 : User stack If an interrupt is taken when the IS bit is 0, then the stack pointer register is loaded from the ISP register before execution starts at the first instruction of the Interrupt Service Routine (ISR). 1 : Shared global stack If an interrupt is taken when the PSW.IS bit is 1, then the current value of the stack pointer is used by the Interrupt Service Routine (ISR).

(table continues...)

(continued)

Field	Bits	Type	Description
GW	8	rw	Global address register write permission Determines whether the current execution thread has permission to modify the global address registers. Most tasks and ISRs use the global address registers as 'read only' registers, pointing to the global literal pool and key data structures. However a task or ISR can be designated as the 'owner' of a particular global address register, and is allowed to modify it. The system designer must determine which global address variables are used with sufficient frequency and/or in sufficiently time-critical code to justify allocation to a global address register. By compiler convention, global address register A[0] is reserved as the base register for short form loads and stores. Register A[1] is also reserved for compiler use. Registers A[8] and A[9] are not used by the compiler, and are available for holding critical system address variables. 0 : Write permission to global registers A[0], A[1], A[8], A[9] is disabled. 1 : Write permission to global registers A[0], A[1], A[8], A[9] is enabled.
CDE	7	rw	Call depth count enable Enables call-depth counting, provided that the PSW.CDC mask field is not all set to 1. 0 : Call depth counting is temporarily disabled. It is automatically re-enabled after execution of the next Call instruction. 1 : Call depth counting is enabled. If PSW.CDC = 1111111_B, call depth counting is disabled regardless of the setting on the PSW.CDE bit.

(table continues...)

(continued)

Field	Bits	Type	Description
CDC	6:0	rw	Call depth counter Consists of two variable width subfields. The first subfield consists of a string of zero or more initial 1 bits, terminated by the first 0 bit. The remaining bits form the second subfield (CDC.COUNT) which constitutes the call depth count value. The count value is incremented on each Call and is decremented on a Return. $0cccccc_B$: 6-bit counter; trap on overflow. $10ccccc_B$: 5-bit counter; trap on overflow. $110cccc_B$: 4-bit counter; trap on overflow. $1110ccc_B$: 3-bit counter; trap on overflow. $11110cc_B$: 2-bit counter; trap on overflow. $111110c_B$: 1-bit counter; trap on overflow. 1111110_B : Trap every call (call trace mode). 1111111_B : Disable call depth counting. When the call depth count (CDC.COUNT) overflows a trap (CDO) is generated. Setting the CDC to 1111110_B allows no bits for the counter and causes every call to be trapped. This is used for Call Depth Tracing. Setting the CDC to 1111111_B disables call depth counting.

3.2.3 PSW user status bits

The eight most significant bits of the PSW are designated as user status bits. These bits may be set or cleared as execution side effects of user instructions, typically recording result status. Individual bits can also be used to condition the operation of particular instructions. For example the ADDX (Add Extended) and ADDC (Add with Carry) instructions use bit 31 to record the carry out from the ADD operation, and the pre-execution value of the bit is reflected in the result of the ADDC instruction.

Table 7 PSW user status bits

Field	Bits	Type	Description
C	31	rw	Carry
V	30	rw	Overflow
SV	29	rw	Sticky overflow
AV	28	rw	Advance overflow
SAV	27	rw	Sticky advance overflow
RES	26:24	-	Reserved field

There are two classes of instructions that employ the user status bits:

- Arithmetic instructions that may produce carry and overflow results
- Implementation-specific coprocessor instructions which may use any or all of the eight bits, in a manner that is entirely implementation specific

Bits [23:16] of the PSW are reserved bits in the architecture. They read as zero when the PSW is read via the MFCR (Move from core register) or MFDCR (Move from core register pair) instructions after a system reset. Their value after writing to the PSW through the MTCR (Move to core register) or MTDCR (move to core register pair) instructions, is architecturally undefined and should be written as zero.

Related information

Exceptions on page 100

PSW user status bits (FPU view) on page 113

3.2.4 Access privilege level control (I/O privilege)

Software managed tasks are created through the services of a real-time kernel or Real-Time Operating System (RTOS), and are dispatched under the control of scheduling software. Interrupt Service Routines (ISRs) are dispatched by hardware in response to an interrupt. An ISR is the code that is invoked directly by the processor on receipt of an interrupt. Software managed tasks are sometimes referred to as user tasks, assuming that they execute in user mode.

Each task is allocated its own mode, depending on the task's function:

- User-0 mode: Used for tasks that do not access peripheral devices. This mode may not enable or disable interrupts
- User-1 mode: Used for tasks that access common, unprotected peripherals. Typically this would be a read or write access to serial port, a read access to timer, and most I/O status registers. Tasks in this mode may disable interrupts. (The interrupt and I/O permissions of this mode may be overridden by the core control register)
- Supervisor mode: Permits read/write access to system registers and all peripheral devices. Tasks in this mode may disable interrupts or change the CPU's interrupt priority
- Hypervisor mode: Permits read/write access to virtualization registers

A set of state elements are associated with any task, and these are known collectively as the task's context. The context is everything the processor needs to define the state of the associated task and enable its continued execution. This includes the CPU general registers that the task uses, the task's Program Counter (PC), and its Program Status Information (PCXI, PPRS and PSW). The architecture efficiently manages and maintains the context of the task through hardware.

3.2.5 Previous context information and pointer register (PCXI)

The Previous Context Information Register (PCXI) contains linkage information to the previous execution context, supporting interrupts and automatic context switching. The PCXI is part of a task's state information. The Previous Context Pointer (PCX) holds the address of the CSA of the previous task.

PCXI. PCX	Address:	$FE00_H$
Previous context information and pointer register	Reset value:	Implementation $specific_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES		**PCPN**								**PIE**	**UL**	**PCXS**			
-		rw								rw	rw	rw			

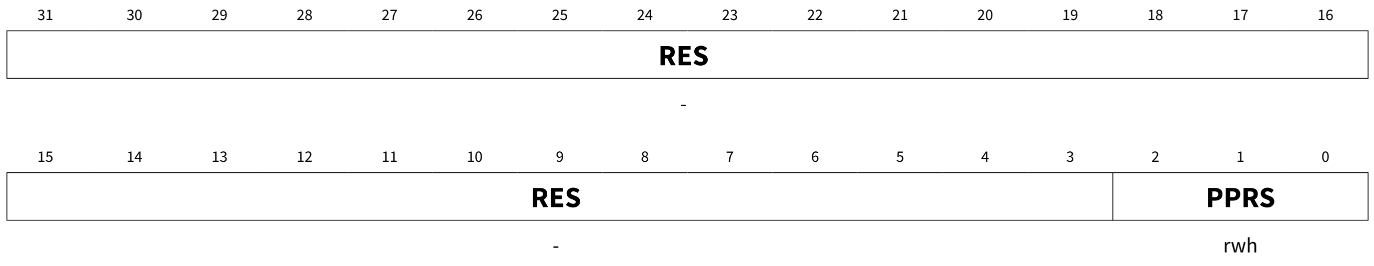
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PCXO															
rw															

Field	Bits	Type	Description
RES	31:30	-	Reserved
PCPN	29:22	rw	Previous CPU priority number Contains the priority level number of the interrupted task.
PIE	21	rw	Previous interrupt enable Indicates the state of the interrupt enable bit (ICR.IE) for the interrupted task.
UL	20	rw	Upper or lower context tag Identifies the type of context saved: 0 : Lower Context 1 : Upper Context If the type does not match the type expected when a context restore operation is performed, a trap is generated.
PCXS	19:16	rw	PCX segment address Contains the segment address portion of the PCX. This field is used in conjunction with the PCXO field.
PCXO	15:0	rw	Previous context pointer offset field The PCXO and PCXS fields form the pointer PCX, which points to the CSA of the previous context.

3.2.6 Previous protection register set register (PPRS)

The previous protection register set register (PPRS) contains the value of the protection register set used by the previous execution context. The PPRS is part of a task's state information.

PPRS	Address:	$FE34_H$
Previous PRS	Reset value:	00000000_H



Field	Bits	Type	Description
RES	31:3	-	Reserved
PPRS	2:0	rwh	Previous PRS

3.3 Stack management registers

Stack management in the architecture supports a user stack and an interrupt stack. Address register A[10], the Interrupt Stack Pointer (ISP) and bit PSW.IS are used in the management of the stack.

A[10] is used as the stack pointer. The initial contents of this register are usually set by an RTOS when a task is created, which allows a private stack area to be assigned to individual tasks.

The ISP helps to prevent Interrupt Service Routines (ISRs) from accessing the private stack areas and possibly interfering with the software managed task's context. An automatic switch to the use of the ISP instead of the private stack pointer is implemented in the architecture. The PSW.IS bit indicates which stack pointer is in effect. When an interrupt is taken and the interrupted task was using its private stack (PSW.IS == 0), the previous value of A[10](SP) is saved with the upper context of the interrupted task and A[10](SP) is loaded with the current contents of the ISP.

When an interrupt or trap is taken and the interrupted task was already using the interrupt stack (PSW.IS == 1), then no pre-loading of A[10](SP) is performed. The Interrupt Service Routine (ISR) continues to use the interrupt stack at the point where the interrupted routine had left it.

Usually it is only necessary to initialize the ISP once during the initialization routine. However, depending on application needs, the ISP can be modified during execution. Note that there is nothing preventing an ISR or system service routine from executing on a private stack.

***Note**: Use of A[10](SP) in an ISR is at the discretion of the application programmer.*

3.3.1 Address register A[10] (SP)

The A[10] Stack Pointer (SP) register is defined as follows:

A[10](SP) Address: $FFA8_H$

Address register A[10] (Stack pointer) Reset value: Implementation $specific_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A10(SP)															
rw															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A10(SP)															
rw															

Field	Bits	Type	Description
A[10](SP)	31:0	rw	Address register A[10], implicitly used as a stack pointer in some instructions.

3.3.2 Interrupt stack pointer register (ISP)

The Interrupt Stack Pointer is defined as follows.

***Note**: This register is ENDINIT protected.*

ISP Address: $FE28_H$

Interrupt stack pointer Reset value: Implementation $specific_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ISP															
rw															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ISP															
rw															

Field	Bits	Type	Description
ISP	31:0	rw	Interrupt stack pointer

3.4 Core configuration registers

The core configuration registers provide information and control relative to a given implementation. Typically these registers can be used to identify a particular feature set or the properties of a core instance. These registers are implementation specific.

3.4.1 Core control register (CORECON)

The Core Configuration Register provides the following functionality:

- Enable bit for temporal protection system
- Enable bit for memory protection system
- Bit for definition of the initial state of the PSW.S bit in interrupt handlers
- Bit for definition of the initial state of the PSW.S bit in trap handlers
- Enable for User-1 IO mode peripheral access as Supervisor
- Disable for User-1 IO mode ability to enable and disable interrupts
- Status indicator of the Free Context List Depletion condition

***Note**:* *The protection of this register is implementation specific.*

CORECON	Address:	FE14$_H$
Core configuration register	Reset value:	0000 0000$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES														U1_IOS	U1_IED
-														rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES											TS	IS	TPROTEN	PROTEN	FCDSF
-											rw	rw	rw	rw	rwh

Field	Bits	Type	Description
RES	31:18	-	Reserved

(table continues...)

(continued)

Field	Bits	Type	Description
U1_IOS	17	rw	User-1 peripheral access as Supervisor. 0 : User-1 mode tasks can only access unprotected peripheral registers 1 : Allow User-1 mode tasks to access peripherals as if in Supervisor mode. Enables User-1 access to all peripheral registers.
U1_IED	16	rw	User-1 instruction execution disable. 0 : User-1 mode instructions can be executed in User-1 IO mode 1 : Disable the execution of User-1 mode instructions in User-1 IO mode. Disables User-1 ability to enable and disable interrupts
RES	15:5	-	Reserved
TS	4	rw	Initial state of PSW.S bit in trap handler
IS	3	rw	Initial state of PSW.S bit in interrupt handler
TPROTEN	2	rw	Temporal protection enable Enable the Temporal Protection system. 0 : Temporal protection is disabled. 1 : Temporal protection is enabled.
PROTEN	1	rw	Memory protection enable Enables the memory protection system. Memory protection is controlled through the memory protection register sets. Note: Initialize the protection register sets prior to setting PROTEN to one. 0 : Memory Protection is disabled. 1 : Memory Protection is enabled.
FCDSF	0	rwh	Free context list depleted sticky flag This sticky bit indicates that a FCD (Free Context List Depleted) trap occurred since the bit was last cleared by software. 0 : No FCD trap occurred since the last clear. 1 : An FCD trap occurred since the last clear.

3.4.2 CPU identification register (CPU_ID)

Identification Registers identify the processor type and revision used. Only the CPU core ID register is described here. All other ID registers will be described in the product specific documentation. The CPU Identification Register identifies the CPU type and revision.

CPU_ID Address $FE18_H$

CPU module identification Reset value: Implementation $specific_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MOD															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOD_32B								**MOD_REV**							
r								r							

Field	Bits	Type	Description
MOD	31:16	r	Module identification number Used for module identification.
MOD_32B	15:8	r	32-Bit module enable A value of C0$_H$ in this field indicates a 32-bit module with a 32-bit module ID register.
MOD_REV	7:0	r	Module revision number Used for revision numbering. The value of the revision starts at 01$_H$ (first revision) up to FF$_H$.

3.4.3 Core identification register (CORE_ID)

In a multiprocessor system each logical processor core is given a unique identification number. In a system supporting virtualization each logical processor core can be used to run multiple virtual machines. The Core Identification Register holds both the core identification number and the current virtual machine number.

CORE_ID	Address:	FE1C$_H$
Core identification	Reset value:	Implementation specific$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES					**VMN**			**RES**					**CORE_ID**		
-					r			-					r		

Field	Bits	Type	Description
RES	31:11	-	Reserved
VMN	10:8	r	Virtual machine number The current virtual machine number. When virtualization is disabled or not present this field will read as 1
RES	7:3	-	Reserved
CORE_ID	2:0	r	Core identification number

3.4.4 Boot configuration register (BOOTCON)

The boot configuration register controls the execution behavior following a reset.

BOOTCON	Address:	FE60$_H$
Boot configuration register	Reset value:	Implementation specific$_H$

31–16
RES
-

15–1	0
RES	**BHALT**
-	rwh

Field	Bits	Type	Description
RES	31:1	-	Reserved
BHALT	0	rwh	Boot halt Following reset a CPU may be immediately placed in halt. In this case the BHALT bit will be set to “1”. The CPU will remain in halt until this bit is written to “0”. On a write from “1” to “0” the CPU will start execution from the program address defined in the program counter (PC) register. A write of this bit to “1” will be ignored.

3.4.5 TriCore™ core configuration register (TCCON)

The TriCore™ core configuration register holds information regarding the hardware support for implementation specific features of the core.

TCCON	Address:	$FE6C_H$
TriCore™ core configuration register	Reset value:	Implementation $specific_H$

31–16
RES
-

15–4	3	2	1	0
RES	**VIRT**	**OVL**	**DP_FPU**	**SP_FPU**
-	r	r	r	r

Field	Bits	Type	Description
RES	31:4	-	Reserved
VIRT	3	r	Virtualization hardware support 0_B Virtualization hardware extension not present or disabled 1_B Virtualization hardware extension present and enabled
OVL	2	r	Overlay hardware support[1)] 0_B No overlay 1_B Overlay present

(table continues...)

(continued)

Field	Bits	Type	Description
DP_FPU	1	r	Double precision FPU hardware support 0_B Double precision FPU hardware not present 1_B Double precision FPU hardware present
SP_FPU	0	r	Single precision FPU hardware support 0_B Single Precision FPU hardware not present 1_B Single Precision FPU hardware present

1) The overlay hardware provides the capability to redirect selected data accesses to the overlay memory

3.5 Compatibility mode register (COMPAT)

The COMPAT register is provided to allow implementations to selectively force compatibility of features with previous versions.

3.5.1 Compatibility mode register (COMPAT)

The contents of the register are implementation specific.

***Note**: This register is SAFETY_ENDINIT protected.*

COMPAT Address: 9400_H

Compatibility mode register Reset value: Implementation specific_H

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Implementation specific															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Implementation specific															
-															

Field	Bits	Type	Description
Implementation specific	31:0	-	Implementation specific

3.6 Access control registers

3.6.1 SIST mode access control register (SMACON)

Implementations may control the operation of Software in System Test (SIST) systems using the SMACON register. The contents of this register is implementation specific.

***Note**: This register is SAFETY_ENDINIT protected*

SMACON Address: $900C_H$

SIST mode access control Reset value: Implementation specific_H

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Implementation specific															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Implementation specific															
-															

Field	Bits	Type	Description
Implementation specific	31:0	-	Implementation specific

3.7 Interrupt registers

A typical service request control register in the TriCore™ architecture holds the individual control bits to enable or disable the request, to assign a priority number, and to direct the request to one of the service providers. The Core Special Function Registers (CSFR) which control the interrupts are described in Interrupt system.

3.8 Memory protection registers

The number of memory protection register sets is specific to each implementation of the architecture. There can be a maximum number of eight sets (one set includes both a data set and a code set). Each code or data set is made up of several address range registers.

Each address range is specialized for code or data addresses, and is enabled for code execute, read or write permission in each code or data set separately. The address range registers specifies the lower and upper boundary addresses of the memory range.

The Core Special Function Registers (CSFR) which control the memory protection registers are described in Memory protection system.

3.9 Trap registers

The Core Special Function Registers (CSFR) which control the trap registers are described in Trap system.

3.10 Memory configuration registers

The memory configuration registers are defined in the architecture but the contents of the registers are implementation specific.

The Core Special Function Registers (CSFR) which control the memory configuration are described in Memory configuration register definitions.

3.11 Core performance counter registers

TriCore™ registers that support performance measurement are described in Core Performance Measurement and Analysis.

3.12 Core debug controller registers

TriCore™ registers that support debugging are described in Core debug controller.

3.13 Floating point registers

The registers for the optional TriCore™ floating point unit are described in FPU CSFR Registers.

3.14 Virtualization registers

The registers for the optional TriCore™ virtualization extension are described in Virtualization register overview.

3.15 Accessing core special function registers (CSFRs)

Core special function registers are read with a MFCR (Move From Core Register) instruction and written with a MTCR (Move To Core register) instruction. The need for software updates to CSFRs is usually infrequent. Implementations are therefore not required to implement hardware structures to avoid hazard conditions that may result from the update of CSFRs. Such hazard conditions are avoided by the insertion of an ISYNC instruction immediately after the MTCR update of the CSFR. The ISYNC instruction ensures that the effects of the CSFR update are correctly seen by all following instructions.

Pairs of CSFRs can be read with a MFDCR (Move From Core Register pair) instruction and written with a MTDCR (Move to Core Register pair) instruction. The pair or registers need to be consecutive and naturally aligned to a double-word boundary.

Access to a single undefined register location by MTCR or MFCR will result in an OPD trap. Access to the location of a pair of registers which are both undefined by MTDCR or MFDCR will result in an OPD trap. The result of access to the location of a pair of registers where only one is defined is implementation defined.

A MTDCR or MFDCR instruction that accesses a misaligned register pair will result in an OPD trap.

4 Tasks and functions

Most embedded and real-time control systems are designed according to a model in which interrupt handlers and software-managed tasks are each considered to be executing on their own 'dedicated' microcontroller. That model is generally supported by the services of a real-time kernel or Real-time Operating System (RTOS), layered on top of the features and capabilities of the underlying machine architecture.

In the TriCore™ architecture, the RTOS layer can be very 'thin' and the hardware can efficiently handle much of the switching between one task and another. At the same time the architecture allows for considerable flexibility in the tasking model used. System designers can choose the real-time executive and software design approach that best suits the needs of their application, with relatively few constraints imposed by the architecture.

The mechanisms for low-overhead task switching and for function calling within the TriCore™ architecture are closely related.

Related information

Virtual machine handling on page 161

4.1 Context types

A task is an independent thread of control. The state of a task is defined by its context. When a task is interrupted, the processor uses that task's context to re-enable the continued execution of the task.

The context types are:

- Upper context: Consists of the upper address registers A[10] to A[15] and the upper data registers D[8] to D[15]. The upper context also includes PCXI, a subset of PSW and the PRS. The complete state of the upper context requires the updated PPRS register to be saved as well when contexts are managed. These registers are designated as non-volatile for purposes of function-calling (their contents are preserved across calls)
- Lower context: Consists of the lower address registers A[2] to A[7], the lower data registers D[0] to D[7], A[11] (return address) and PCXI

Contexts, when saved to memory, occupy 16 word blocks of storage, known as Context Save Areas (CSAs).

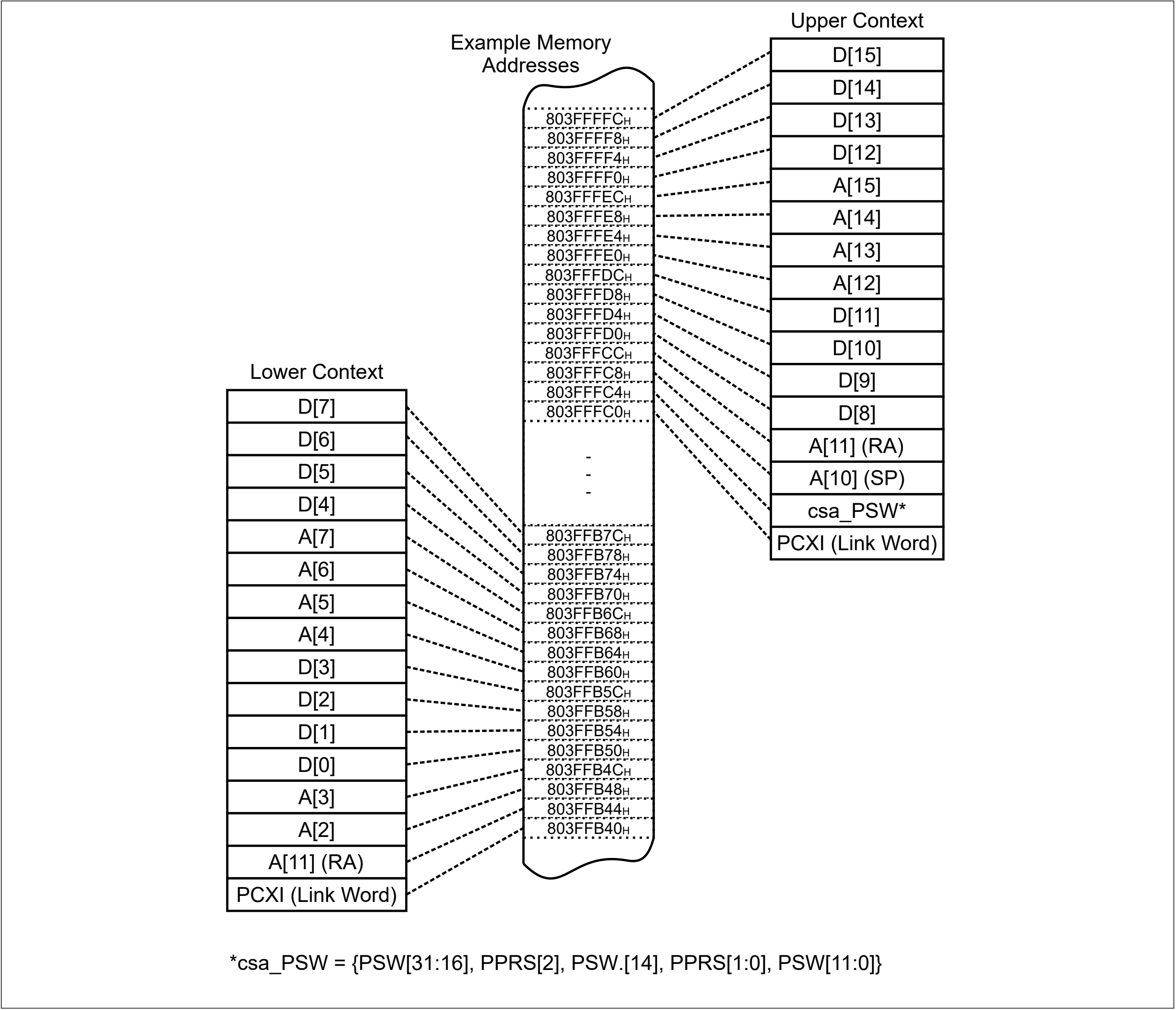


Figure 12 **Upper and lower contexts**

4.1.1 Context save area

The architecture uses linked lists of fixed-size context save areas. A CSA is 16 words of memory storage, aligned on a 16 word boundary. Each CSA can hold exactly one upper or one lower context. CSAs are linked together through a link word.

The link word includes two fields that link the given CSA to the next one in a chain. The fields are a 4-bit segment and a 16-bit offset. The segment number and offset are used to generate the Effective Address (EA) of the linked CSA. See Figure 13.

Incrementing the pointer offset value by one always increments the EA to the address that is 16 word locations above the previous one. The total usable range in each address segment for CSAs is 4 MBytes, resulting in storage space for 2^{16} CSAs.

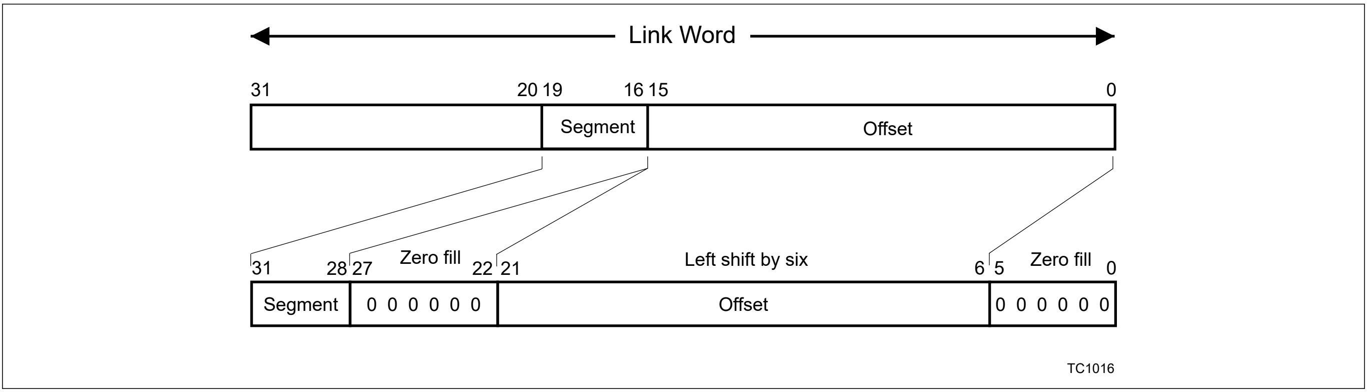


Figure 13 **Generation of the effective address of a context save area (CSA)**

If the CSA is in use (for example, it holds an upper or lower context image for a suspended task), then the link word also contains other information about the linked context. The entire link word is a copy of the PCXI register for the associated task.

For further information on how linked CSAs support context switching, refer to Context save areas (CSAs) and context lists.

4.2 Task switching operation

The architecture switches tasks when one of the events or instructions listed in Table 8, occurs. When one of these events or instructions is encountered, the upper or lower context of the task is saved or restored. The upper context is saved automatically as a result of an external interrupt, trap or function call. The lower context is saved explicitly through instructions. In Table 8 'Save' is a store through the Free CSA List Head Pointer register (FCX) after the next value for the FCX is read from the link word. 'Store' is a store through the effective address of the instruction with no change to the CSA list or the FCX register. 'Restore' is the converse of 'Save'. 'Load' is the converse of 'Store'.

There is an essential difference in the treatment of registers in the upper and lower contexts, in terms of how their contents are maintained. The lower context registers are similar to global registers in the sense that an interrupt handler, trap handler or called function, sees the same values that were present in the registers just before the interrupt, trap or call. Any changes made to those registers that are made in the interrupt, trap handler or called function, remains present after the return from the event, since they are not automatically restored as part of the return from call (RET) or return from exception (RFE) semantics. That means that the lower context registers can be used to pass arguments to called functions and pass return values from those functions. It also means that interrupt and trap handlers must save the original values they find in these registers before using the registers, and restore the original values before exiting.

With the exception of A[11](RA) the upper context register values are retained from the caller code to the called function. On a RET, the upper context is restored.

The upper context registers are not guaranteed to retain their values for exceptions. Conceptually, a trap or interrupt handler always begins execution with its own private set of upper context registers. For exceptions the upper context registers of the interrupted function are not inherited.

Only the A[10](SP), A[11](RA), PSW, PCXI, PPRS and D[15] registers start with architecturally defined values in trap handlers or interrupt handlers. A trap handler or interrupt handler that reads any of the other upper context registers before writing a value into it, is performing an undefined operation.

Table 8 Context related events and instructions

Event/instruction	Context operation	Complement instruction	Context operation
Interrupt	Save upper	RFE - Return from exception	Restore upper
Trap	Save upper	RFE - Return from exception	Restore upper
CALL - Function call	Save upper	RET - Return from call	Restore upper
BISR - Begin interrupt service routine	Save lower	RSLCX - Restore lower context	Restore lower
SVLCX - Save lower context	Save lower	RSLCX - Restore lower context	Restore lower
STLCX - Store lower context	Store lower	LDLCX - Load lower context	Load lower
STUCX - Store upper context	Store upper	LDUCX - Load upper context	Load upper

Related information

Hypervisor and virtual machine switching operation on page 161

4.3 Context save areas (CSAs) and context lists

The upper and lower contexts are saved in Context Save Areas (CSAs). Unused CSAs are linked together in the Free Context List (FCX). CSAs that contain saved upper or lower contexts are linked together in the Previous Context List (PCX). The following figure shows a simple configuration of CSAs within both context lists.

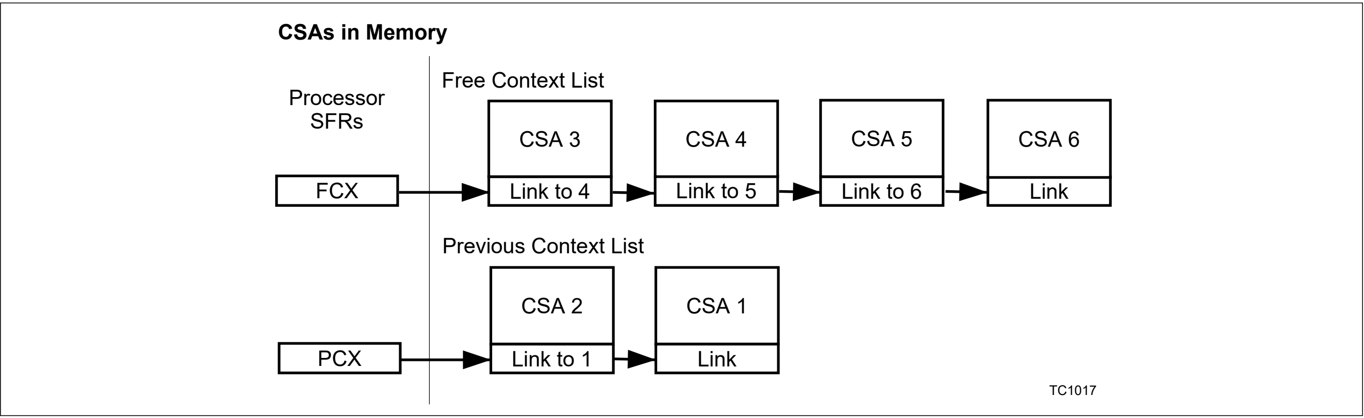


Figure 14 CSAs in context lists

The contents of the FCX register always points to an available CSA in the free context list. That CSAs link word points to the next available CSA in the free context list.

Before an upper or lower context is saved in the first available CSA, its link word is read, supplying a new value for the FCX. To the memory subsystem, context saving is therefore a read/modify/write operation. The new value of FCX, which points to the next available CSA, is available immediately for subsequent upper or lower context saves.

The LCX register points to one of the last CSAs in the free list and is used to recognize impending free CSA list depletion. If the value of FCX matches that of LCX when an operation that performs a context save is attempted, the operation completes and a free CSA list depletion trap (FCD) is taken on the next instruction; that means, the return address of the FCD trap is the first instruction of the trap/interrupt/called routine or the instruction following an SVLCX or BISR instruction. See Context switching with interrupts and traps.

The action taken by the trap handler depends on the software implementation. It might issue a system reset for example, if it is determined that the CSA list depletion resulted from an unrecoverable software error. Normally however it extends the free list, either by allocating additional memory or by terminating one or more tasks and reclaiming their CSA call chains. In those cases the trap handler exits with a RFE instruction.

The link word in the last CSA in a free context list must be set to null before it is first used. This is necessary to support the FCU trap. Before first use of the CSA, the PCX pointer value should be null. This is to support CSU (Call Stack Underflow) traps.

The PCXI.PCX field points to the CSA where the previous context was saved. The PCXI.UL bit identifies whether the saved context is upper (PCXI.UL == 1) or lower (PCXI.U == 0). If the type does not match the type expected when a context restore operation is performed, a CYTP exception occurs and a context management trap is taken.

After the context save operation has been performed the Return Address A[11](RA) is updated:

- For a call, the A[11](RA) is updated with the function return address
- For a synchronous trap, the A[11](RA) is updated with the PC of the instruction which raised the trap
- For a SYSCALL and an asynchronous trap or an interrupt, the A[11](RA) is updated with the PC of the next instruction to be executed

When a lower context save operation is performed the value of A[11](RA) is included in the saved context and is placed in the second word of the CSA. This A[11](RA) is correspondingly restored by a lower context restore.

The call depth control field (PSW.CDC) consists of two sub-fields; A call depth counter, and a mask that determines the width of the counter and when it overflows.

The call depth counter is incremented on calls and is restored to its previous value on returns. An exception occurs when the counter overflows. Its purpose is to prevent software errors from causing ‘runaway recursion’ and depleting the CSA free list.

4.4 Context switching with interrupts and traps

When an interrupt or trap (for example NMI or SYS trap) occurs, the processor saves the upper context of the current task in memory, suspends execution of the current task and then starts execution of the interrupt or trap handler.

If, when an interrupt or trap is taken, the processor is not using the interrupt stack (PSW.IS bit == 0), the stack pointer is then loaded with the current contents of the ISP (Interrupt Stack Pointer). The PSW.IS bit is then set to one (1) to indicate execution from the interrupt stack.

The Interrupt Control Register (ICR) holds the Current CPU Priority Number (ICR.CCPN), the Interrupt Enable bit (ICR.IE) and Pending Interrupt Priority Number (ICR.PIPN). These fields, together with the Previous CPU Priority Number (PCXI.PCPN) and Previous Interrupt Enable (PCXI.PIE) are all part of the interrupt management system.

ICR.CCPN is typically only non-zero within Interrupt Service Routines (ISRs) where it is used to order interrupt servicing. It is held in a register that is separate from the PSW and is not part of the context that the RTOS handles for switching among Software Managed Tasks.

PCXI.PIE is only typically zero within trap handlers started within ISRs, for example an NMI trap or SYSCALL instruction occurring during a peripheral service request.

For both interrupts and traps, the existing PCPN and PIE values in the current PCXI are saved in the CSA for the upper context, and the existing IE and CCPN values in the ICR are copied to the PCXI.PIE and PCXI.PCPN fields. Once the interrupt or trap is handled, the saved lower context is reloaded if necessary and execution of the interrupted task is resumed (RFE).

On an interrupt or trap the upper context of the current task context is saved by hardware as an explicit part of the exception entry sequence. For small exception handlers that can execute entirely within this set of registers saved on the exception, no further context saving is needed. The handler can execute immediately and return. Typically handlers that make calls or require more registers execute the BISR (Begin Interrupt Service Routine) or SVLCX (Save Lower Context) instruction to save the lower context registers that were not saved as part of the interrupt or trap sequence. That instruction must be issued before any of the associated registers are modified, but it need not be the first instruction in the handler.

Interrupt handlers with critical response time requirements can perform their initial, time-critical processing immediately, using upper context registers. After that they can execute a BISR and continue with less time-critical processing. The BISR re-enables interrupts, hence its use dividing time critical from less time critical processing.

Trap handlers typically do not have critical response time requirements, however those that can occur in an ISR or those which might hold off interrupts for too long can also take a similar approach to distinguish between non-interruptible and interruptible execution code.

4.5 Context switching for function calls

When a function call is made (a CALL, CALLA or CALLI instruction is executed), the context of the calling routine must be saved and then restored in order to resume the caller's execution after return from the function.

On a function call the entire set of upper context registers are saved by hardware. Furthermore, the saving of the upper context by the CALL operation happens in parallel with the call jump. In addition, restoring the upper context is performed by the RET (return) instruction and takes place in parallel with the return jump. The called function does not need to save and restore the caller's context and is freed of any need to restrict its usage of the upper context registers. The calling and called functions must co-operate on the use of the lower context registers.

4.6 Return target fetch (A[11] and PPRS)

The current context contains all the information required to efficiently return to the previous context. The return address is available in A[11](RA) while the previous protection set is saved in the PPRS register. PPRS is the architecturally defined protection register set for fetching the location pointed to by A[11]. PPRS is added to the list of additional registers for consideration with task switching .

On an upper context save (function call, exception, hypervisor exception) the current value of PPRS is saved in place of the PSW.PRS in the CSA.

- `EA = {FCX.FCXS, 6'b0, FCX.FCXO, 6'b0};`
- `csa_PSW = {PSW[31:16], PPRS[2], PSW[14], PPRS[1:0], PSW[11:0]}`
- `PPRS    = PSW.PRS`
- `M(EA,16 * word) = {PCXI, csa_PSW, A[10], A[11], D[8], D[9], D[10], D[11], A[12], A[13], A[14], A[15], D[12], D[13], D[14], D15]};`

On an upper context restore (return from call, return from exception, return from hypervisor) the value of PPRS is used to apply the correct protection to the target of A[11](RA) and is also restored from the CSA.

- `EA = {PCXI.PCXS, 6'b0, PCXI.PCXO, 6'b0};`
- `{new_PCXI, csa_PSW, A[10], A[11], D[8], D[9], D[10], D[11], A[12], A[13], A[14], A[15], D[12], D[13], D[14], D[15]} = M(EA, 16 * word);`
- `PSW  = {csa_PSW[31:16], PPRS[2], csa_PSW[14], PPRS[1:0], csa_PSW[11:0]};`
- `PPRS = {csa_PSW.PRS[2], csa_PSW.PRS[1:0]};`

4.7 Fast function calls with FCALL/FRET

In situations where the saving and restoring of the upper context registers is not required an FCALL, FCALLA or FCALLI instruction may be used in preference to a CALL operation. The FCALL operation performs a call jump and in parallel saves the current return address (A[11]) to the stack. No other state is saved. The called function therefore starts execution with the same context as the caller (with the exception of A[10] and A[11]).

To return from a function called using an FCALL operation, an FRET (fast return) instruction is executed. This performs a jump to the current return address (A[11]) and loads the previous A[11] back from the stack. No other state is loaded. The caller function therefore resumes execution with a context modified by the called function. The calling and called functions must co-operate on the use of all registers.

4.8 Context save and restore examples

This section provides an example of a context save operation and an example of a context restore operation.

4.8.1 Context save

The following figure shows the free and previous context lists for this example. The free context list (FCX) contains three free CSAs (3, 4, and 5), and the previous context list (PCX) contains two CSAs (2 and 1).

The FCX points to CSA3, the first available CSA. The link word of CSA3 points to CSA4; the link word of CSA4 points to CSA5. The PCX points to the most recently saved CSA in the previous context list. The link word of CSA2 points to CSA1. CSA1 contains the saved context prior to CSA2.

When the context save operation is performed, the first CSA in the free context list (CSA3) is pulled off and is placed on the front of the previous context list.

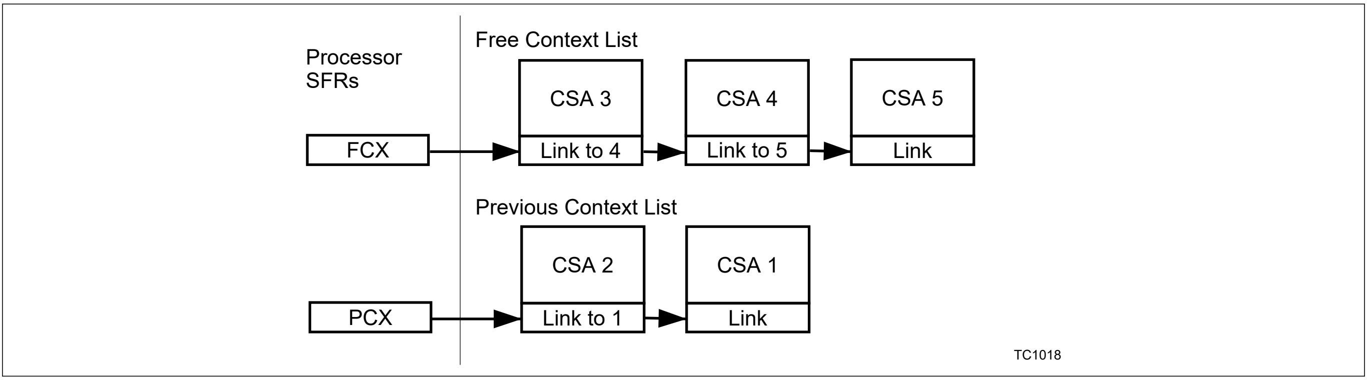


Figure 15 **CSAs and processor state prior to context save**

The following figure shows the steps taken during the context save operation. The numbers in the figure correspond to the steps listed after the figure.

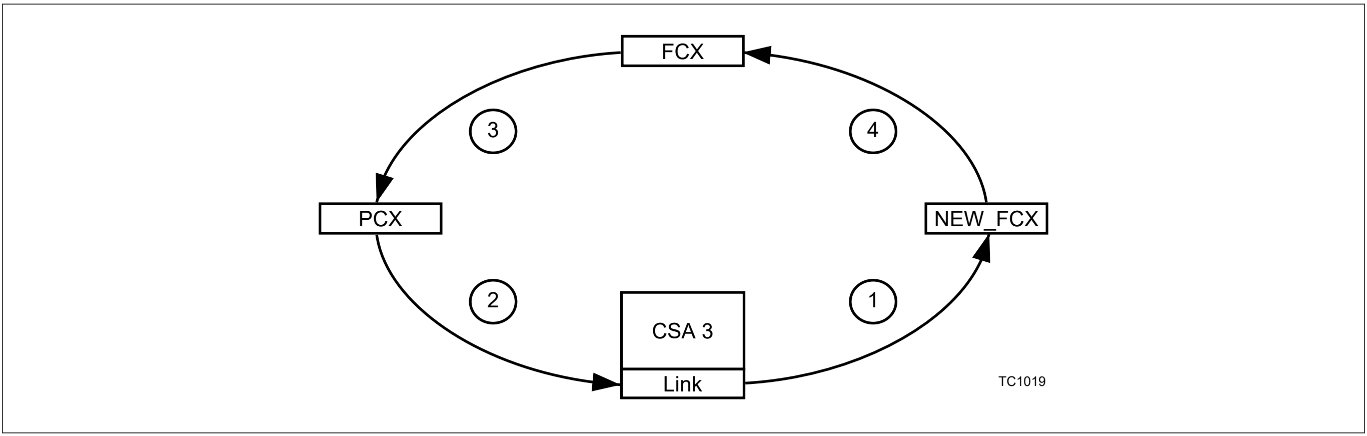


Figure 16 **CSA and processor SFR updates on a context save process**

1. The contents of the link word in CSA3 are loaded into the NEW_FCX. The NEW_FCX now points to CSA4. The NEW_FCX is an internal buffer and is not accessible by the software
2. The contents of the PCX are written into the link word of CSA3. The Link Word of CSA3 now points to CSA2

3. The contents of FCX are written into the PCX. The PCX now points to CSA3, which is at the front of the previous context list
4. The NEW_FCX is loaded into the FCX

The processor SFRs and CSAs look as shown in the following figure. The processor context to be saved is now written into the rest of CSA3.

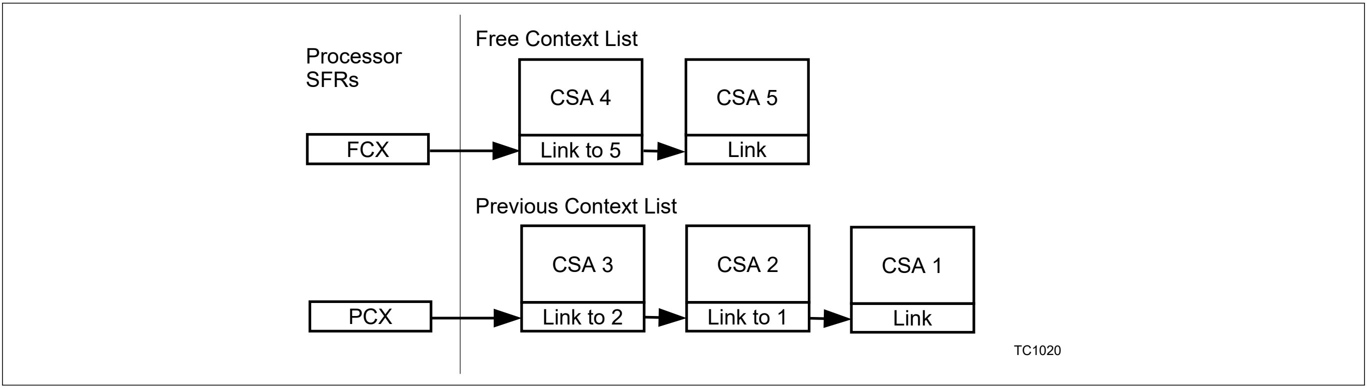


Figure 17 **CSAs and processor state after context save**

4.8.2 Context restore

The example in the following figure shows the previous context list (PCX) with three CSAs (3, 2, and 1) and the free context list (FCX) containing two CSAs (4 and 5).

The FCX points to CSA4, the first available CSA in the free context list. PCX points to CSA3, the most recently saved CSA in the previous context list.

The link word of CSA3 points to CSA2; the link word of CSA2 points to CSA1; the link word of CSA4 points to CSA5.

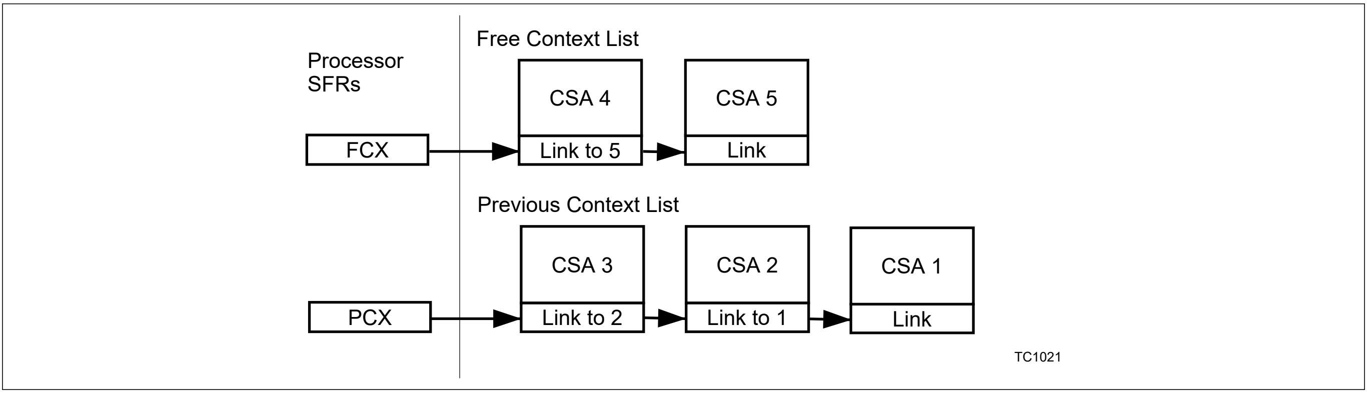


Figure 18 **CSAs and processor state prior to context restore**

When the context restore operation is performed, the first CSA in the previous context list (CSA3) is pulled off and is placed on the front of the free context list.

The following figure shows the steps taken during the context restore operation. The numbers in the figure correspond to the following steps:

1. The contents of the link word in CSA3 are loaded into the NEW_PCX. The NEW_PCX now points to CSA2. The NEW_PCX is an internal buffer and is not accessible by the software
2. The contents of the FCX are written into the link word of CSA3. The link word of CSA3 now points to CSA4
3. The contents of the PCX are written into the FCX. The FCX now points to CSA3, which is at the front of the free context list
4. The NEW_PCX is loaded into the PCX

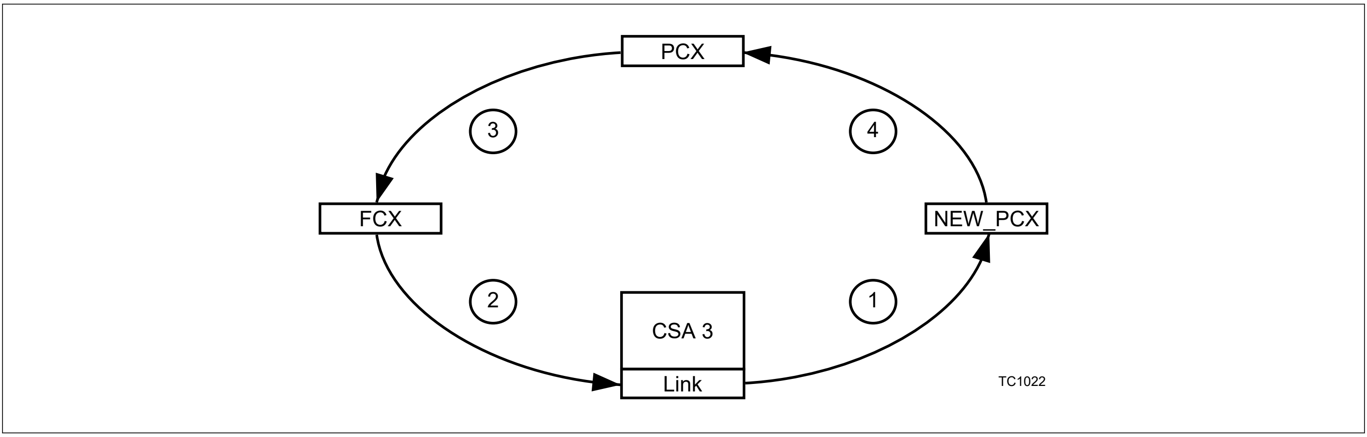


Figure 19 **CSA and processor SFR updates on a context restore process**

The processor SFRs and CSAs now look as shown in the following figure. The restored context is then written into the upper or lower context registers.

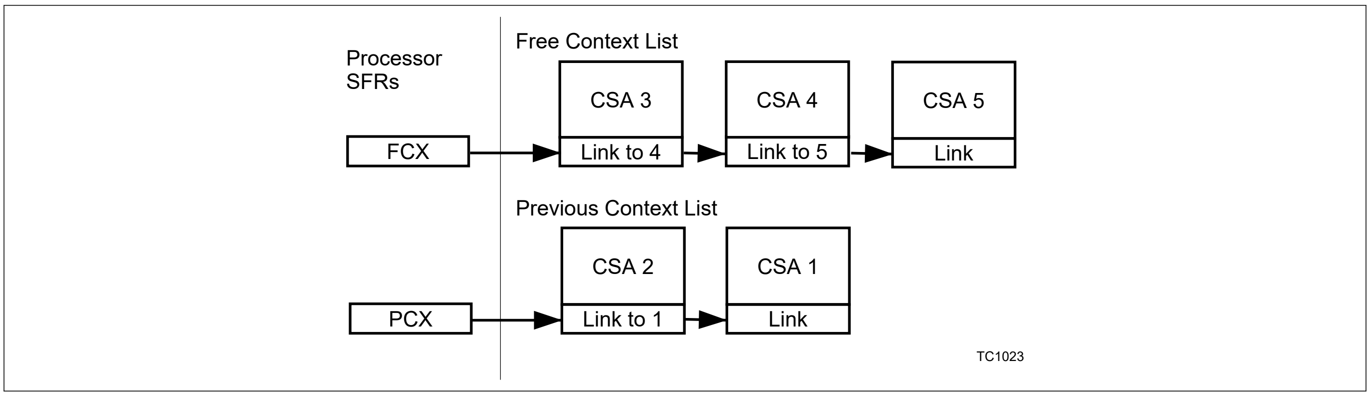


Figure 20 **CSAs and processor state after context restore**

4.9 Context management registers

The three context management registers are pointers that are used during context save and restore operations.

- FCX: Free CSA list head pointer register (FCX)
- PCX: Previous context pointer register (PCX)
- LCX: Free CSA list limit pointer register (LCX)

Each pointer consists of two fields:

- A 16-bit offset
- A 4-bit segment specifier

The following figure shows how the effective address of a context save area (CSA) is generated using these two fields. A context save area is an address range containing 16 word locations (64 bytes), which is the space required to save one upper or one lower context. Incrementing the pointer offset value by one always increments the effective address (EA) to the address that is 16 word locations above the previous one. The total usable range in each address segment for CSAs is 4 MBytes, resulting in storage space for 2^16 CSAs.

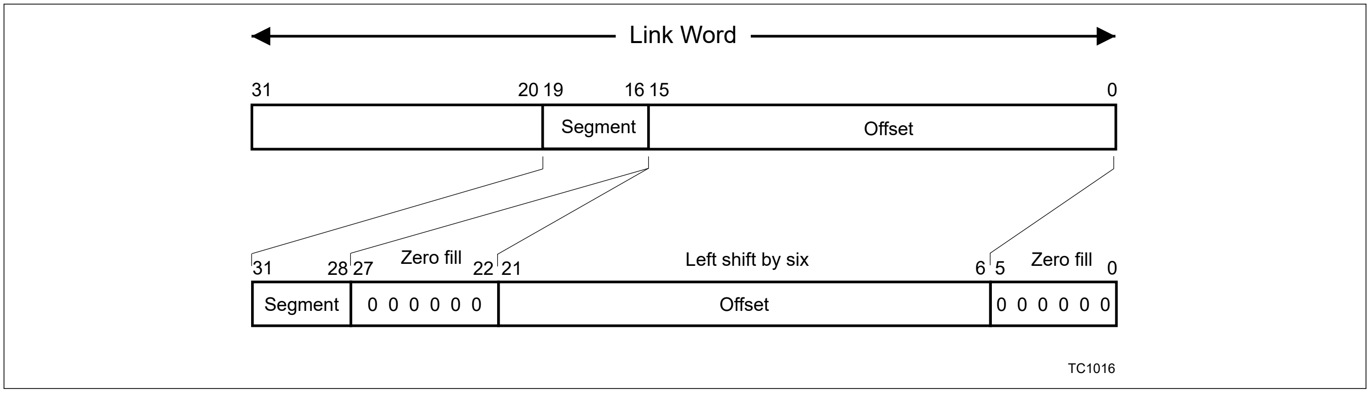


Figure 21 **Generation of the effective address of a context save area (CSA)**

***Note**: See Context save area for additional constraints on the effective address (EA).*

4.9.1 Registers

4.9.1.1 Free CSA list head pointer register (FCX)

The Free CSA List Head Pointer (FCX) register holds the free CSA list head pointer. This always points to an available CSA.

FCX	Address:	FE38$_H$
Free CSA list head pointer	Reset value:	Implementation specific$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES												**FCXS**			
-												rw			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FCXO															
rw															

Field	Bits	Type	Description
RES	31:20	-	Reserved
FCXS	19:16	rw	FCX segment address Used in conjunction with the FCXO field.
FCXO	15:0	rw	FCX offset address The FCXO and FCXS fields together form the FCX pointer, which points to the next available CSA.

4.9.1.2 Previous context pointer register (PCX)

The Previous Context Pointer (PCX) holds the address of the CSA of the previous task. The PCX is part of the PCXI register.

PCX	Address:	FE00$_H$
Previous context pointer register	Reset value:	Implementation specific$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES												**PCXS**			
-												rw			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PCXO															
rw															

Field	Bits	Type	Description
RES	31:20	-	Reserved
PCXS	19:16	rw	Previous context pointer segment address This field is used in conjunction with the PCXO field.
PCXO	15:0	rw	Previous context pointer offset The PCXO and PCXS fields form the pointer PCX, which points to the CSA of the previous context.

4.9.2 Free CSA list limit pointer register (LCX)

The free CSA List Limit Pointer (LCX) register is used to recognize impending free CSA list depletion. If a context save operation occurs and the value of FCX matches LCX then the ‘free context depletion’ condition is recognized, which triggers an FCD trap immediately after completion of the operation causing the context save; that means the return address of the FCD trap is the first instruction of the trap/interrupt/called routine, or the instruction following an SVLCX or BISR instruction.

***Note**: Please refer to the FCD trap description for details on the use and setting of LCX. See FCD - Free context list depletion (TIN 1).*

4.9.2.1 Free CSA list limit pointer register (LCX)

LCX	Address:	FE3C$_H$
Free CSA list limit pointer	Reset value:	Implementation specific$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES												**LCXS**			
-												rw			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LCXO															
rw															

Field	Bits	Type	Description
RES	31:20	-	Reserved
LCXS	19:16	rw	LCX segment address This field is used in conjunction with the LCXO field.

(table continues...)

(continued)

Field	Bits	Type	Description
LCXO	15:0	rw	LCX offset The LCXO and LCXS fields form the pointer LCX, which points to the last available CSA.

4.10 Accessing CSA memory locations

Implementations may internally buffer context information to increase performance. To ensure memory coherency with load and store operations on the same core, or with external agents, a DSYNC instruction must be executed prior to any access to an active CSA memory location. The DSYNC instruction forces all internally buffered CSA state to be written to memory.

4.11 Context save area placement

Context Save Areas (CSAs) may not be placed in memory segments which have the peripheral space attribute (see Peripheral space), or in memory areas that undergo address translation (if an MMU is present and enabled).

***Note**: Individual TriCore™ implementations may place additional restrictions on CSA placement. Such restrictions will be detailed in the relevant user manual accompanying a specific TriCore™ product.*

5 Interrupt system

In a TriCore™ system, multiple sources such as peripherals or external interrupts can generate interrupt requests to interrupt service providers such as CPUs or a DMAs channels. This chapter describes the interrupt processing capabilities of the CPU including the interrupt prioritization scheme and access to the vector table.

Related information

Virtualization: Interrupt system on page 172

5.1 General operation

Each interrupt source is assigned a unique interrupt priority number known as the Service Request Priority Number (SRPN). On receipt of an interrupt request from an interrupt source the SRPN is used by the Interrupt Router (IR) in the system to prioritize between multiple concurrent interrupt requests. The SRPN of the winning request is supplied to the CPU as a Pending Interrupt Priority Number (PIPN). The CPU determines whether to accept a requested interrupt by comparing the PIPN with its Current CPU Priority Number (CCPN). If the CPU will take the requested interrupt it signals the IR that it has been accepted so that the IR can clear that request.

5.1.1 Interrupt control register (ICR)

The Interrupt Control Register (ICR) holds the Current CPU Priority Number (CCPN), the global Interrupt enable/ disable bit (IE) and the current Pending Interrupt Priority Number (PIPN).

5.1.2 CPU operation on an interrupt request

The CPU checks the state of the global interrupt enable bit ICR.IE, and compares the current CPU priority number ICR.CCPN against the PIPN. The CPU can be interrupted only if ICR.IE == 1 and PIPN is greater than CCPN. If this is true the CPU can enter the service routine. The PIPN is used to determine the interrupt vector table entry point.

Several conditions could block the CPU from immediately responding to the interrupt request presented to it. These are:

- The interrupt system is disabled for this CPU (ICR.IE == 0)
- The current CPU priority (CCPN), is equal to or higher than the Pending Interrupt Priority Number (PIPN)
- The CPU is in the process of entering an interrupt or trap service routine
- The CPU is operating on non-interruptible trap services
- The CPU is executing a multi-cycle instruction
- The CPU is executing an instruction which modifies the ICR

The CPU responds to the interrupt request only when these conditions are no longer true.

5.2 Entering an interrupt service routine (ISR)

When all conditions are clear for the CPU to service an interrupt request, the following actions are performed to enter an Interrupt Service Routine (ISR):

- The upper context of the current task is saved
- The return address (A[11]) is updated with the current PC
- D[15] is set to 0
- If the processor was not previously using the interrupt stack (PSW.IS = 0), then the A[10] stack pointer is set to the interrupt stack pointer (ISP). The stack pointer bit is then set for using the interrupt stack: PSW.IS = 1
- The I/O mode is set to Supervisor mode, which means Supervisor permissions are enabled: PSW.IO = 10_B

- The current protection register set is set to 0: PSW.PRS = 000_B
- The Call Depth Counter (PSW.CDC) is cleared, and the call depth limit selector is set for 64: PSW.CDC = 0000000_B
- Call depth counter is enabled, PSW.CDE = 1
- PSW Safety bit is set to value defined in the CORECON register. PSW.S = CORECON.IS
- Write permission to global registers A[0], A[1], A[8], A[9] is disabled: PSW.GW = 0
- The interrupt system is disabled for this CPU: ICR.IE = 0. The old ICR.IE is saved into PCXI.PIE
- The Current CPU Priority Number (ICR.CCPN) is saved into the Previous CPU Priority Number (PCXI.PCPN) field
- The Pending Interrupt Priority Number (ICR.PIPN) is written to the Current CPU Priority Number (ICR.CCPN) field
- The interrupt vector table is accessed to fetch the first instruction of the ISR

***Note**:* *Global register write permission is disabled (PSW.GW == 0) whenever an interrupt service routine or trap handler is entered. This ensures that all traps and interrupts must assume they do not have write access to the registers controlled by PSW.GW by default.*

An interrupt service routine is entered with the interrupt system disabled for this CPU and the current CPU priority (CCPN) set to the priority (PIPN) of the interrupt being serviced. It is up to the software to enable the CPU's interrupt system again and optionally modify the priority number CCPN to implement interrupt priority levels or handle special cases. See Using the TriCore™ interrupt system.

The interrupt system can be enabled with the ENABLE instruction. ENABLE sets ICR.IE = 1 (interrupt system enabled). The BISR (Begin Interrupt Service Routine) instruction also enables the interrupt system, sets the ICR.CCPN to a new value, and saves the lower context of the interrupted task. The instruction RESTORE D[a] sets the ICR.IE value to that of bit 0 of D[a]. This is useful for example in trap handlers which might manipulate interrupt state then wish to return it to a preserved value. The interrupt enable bit (ICR.IE) and current CPU priority number (ICR.CCPN) can also be modified with the MTCR (Move To Core Register) instruction.

The ENABLE, BISR, RESTORE and DISABLE (disable interrupts) instructions are all executed such that the CPU is blocked from taking interrupt requests until the instruction executes. This avoids pipeline side effects and eliminates the need for an ISYNC (synchronize instruction stream) following these instructions. MTCR and MTDCR are exceptions and must be followed by an ISYNC instruction.

5.3 Exiting an interrupt service routine (ISR)

When an ISR exits with an RFE (return from exception) instruction, the hardware automatically restores the upper context. The upper context includes the PCXI register which holds the Previous CPU Priority Number (PCPN) and the Previous Global Interrupt Enable Bit (PIE). The values in these respective bits are used as follows:

- PCXI.PCPN is written to ICR.CCPN to set the CPU priority number to the value before interruption
- PCXI.PIE is written to ICR.IE to restore the state of this bit

The interrupted routine then continues.

5.4 Interrupt vector table

Interrupt service routines are associated with interrupts at a particular priority by way of the interrupt vector table. The interrupt vector table is an array of Interrupt Service Routine (ISR) entry points. The interrupt vector table is stored in code memory.

When the CPU takes an interrupt, it calculates an address in the interrupt vector table that corresponds with the priority of the interrupt (the ICR.PIPN bit-field). This address is loaded into the program counter. The CPU begins executing instructions at this address in the interrupt vector table. The code block at this address is the start of the selected Interrupt Service Routine (ISR). Depending on the code size of the ISR, the interrupt vector table may only contain the initial portion of the ISR, such as a jump instruction that redirects the CPU to the rest of the ISR elsewhere in memory.

The Base of Interrupt Vector Table (BIV) register stores the base address of the interrupt vector table. Interrupt vectors are ordered in the table by increasing priority. The BIV register can be modified using the MTCR instruction during the initialization phase of the system (the BIV is ENDINIT protected), before interrupts are enabled. With this arrangement, it is possible to have multiple interrupt vector tables and switch between them by changing the contents of the BIV register.

When interrupted, the CPU calculates the entry point of the appropriate interrupt service routine from the PIPN and the contents of the BIV register. Two vector table configurations are available with either 32-byte or 8-byte spacing between vectors. The spacing is selected by the Vector Size Select (VSS) bit of the BIV register.

To generate a pointer into the Interrupt vector table the PIPN is left-shifted by either five bits (VSS = 0), or three bits (VSS = 1) and ORd with the address in the BIV register to generate a pointer into the Interrupt Vector Table. Execution of the ISR begins at this address. Due to this operation, it is recommended that bits [12:5] (VSS = 0) or bits[10:3] (VSS = 1) of register BIV are set to 0.

```
if (BIV.VSS == 0B)
  ISR_Entry_PC = {BIV[31:1],0B} | {PIPN<<5};
else
  ISR_Entry_PC = {BIV[31:1],0B} | {PIPN<<3};
```

If an interrupt handler is very short it may fit entirely within the words available in the vector code block. Otherwise the code stored at the entry location can either span several vector entries, or should contain some initial instructions followed by a jump to the rest of the handler. See Spanning interrupt service routines across vector entries

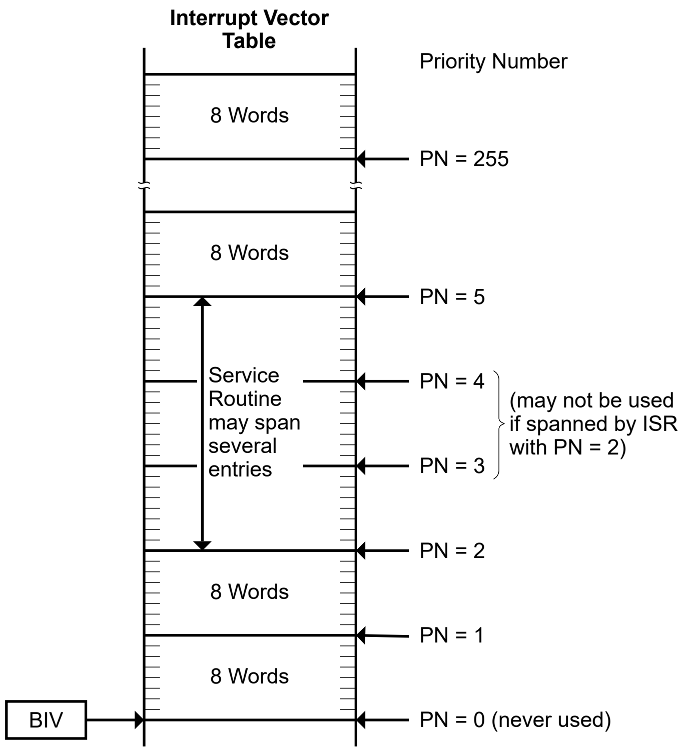


Figure 22 **Interrupt vector table (VSS = 0)**

The BIV register allows the interrupt vector table to be located anywhere in the available code memory. The default on power-up is implementation specific. The BIV register can be written to using the MTCR instruction during the initialization phase of the system, before interrupts are enabled. It is also possible to have multiple interrupt vector tables and switch between them simply by modifying the contents of the BIV register. Since the PC is loaded with the address of the ISR it is fetched in the same manner as all code. Therefore, frequently accessed blocks of the Interrupt Vector Table may be found in the instruction cache or the entire table located in the program scratchpad RAM for performance reasons.

5.5 Using the TriCore™ interrupt system

The following sections contain examples showing how the TriCore™ architectures flexible interrupt system can be used to solve both typical and special application requirements.

5.5.1 Spanning interrupt service routines across vector entries

Because vector entries are not tied to the interrupt source, it is easy to span Interrupt Service Routines (ISRs) across vector entry locations, as shown previously in Figure 22. Spanning eliminates the need of a jump to the rest of the interrupt handler if it would not fit into the available two or eight words between entry locations.

Note that priority numbers relating to entries occupied by a spanned service routine must not be used for any of the active Service Request Nodes (SRNs) which request service from the same service provider.

In Figure 22, vector locations three and four are covered through the service routine for vector entry two. Therefore these numbers must not be assigned to SRNs requesting CPU service, although they can be used to request another service provider. The next available vector entry is now entry five.

Use of this technique increases the range of priority numbers required in a given system, consequently the size of the vector table must be adjusted accordingly.

5.5.2 Interrupt priority groups

Interrupt priority groups describe a set of interrupts which cannot interrupt each others service routine. These groups are easily created with the TriCore™ interrupt system architecture.

When the CPU starts the service of an interrupt, the interrupt system is disabled for this CPU and the CPU priority CCPN is set to the priority of the interrupt being serviced. This blocks all further interrupts from being serviced until the interrupt system is either enabled again through software, or the service routine is terminated with the RFE (return from exception) instruction.

***Note**: The RFE instruction automatically re-installs the previous state of the ICR.IE bit. This will be one (ICE.IE = 1), otherwise that interrupt would not have been serviced.*

When Interrupt Service Routine (ISR) software enables the interrupt system again by setting ICR.IE without changing the CCPN, the effect is that all interrupt requests with the same or lower priority than the CCPN are still blocked from being serviced. This includes a re-occurrence of the current interrupt; that means it can not interrupt this service.

However this ISR will be interrupted by each request which has a higher priority number than the CCPN. A potential problem (that is easily overcome in the TriCore™ architecture) is that application requirements often require interrupt requests of similar significance to be grouped together in such a way that no request in that group can interrupt the ISR of another member of the same group.

Creating these interrupt priority groups is easily accomplished in the interrupt system. For a defined group of interrupt requests, the software of their respective service routines sets the CCPN to the number of the highest SRPN used in that group, before enabling the interrupt system again. The following figure shows an example.

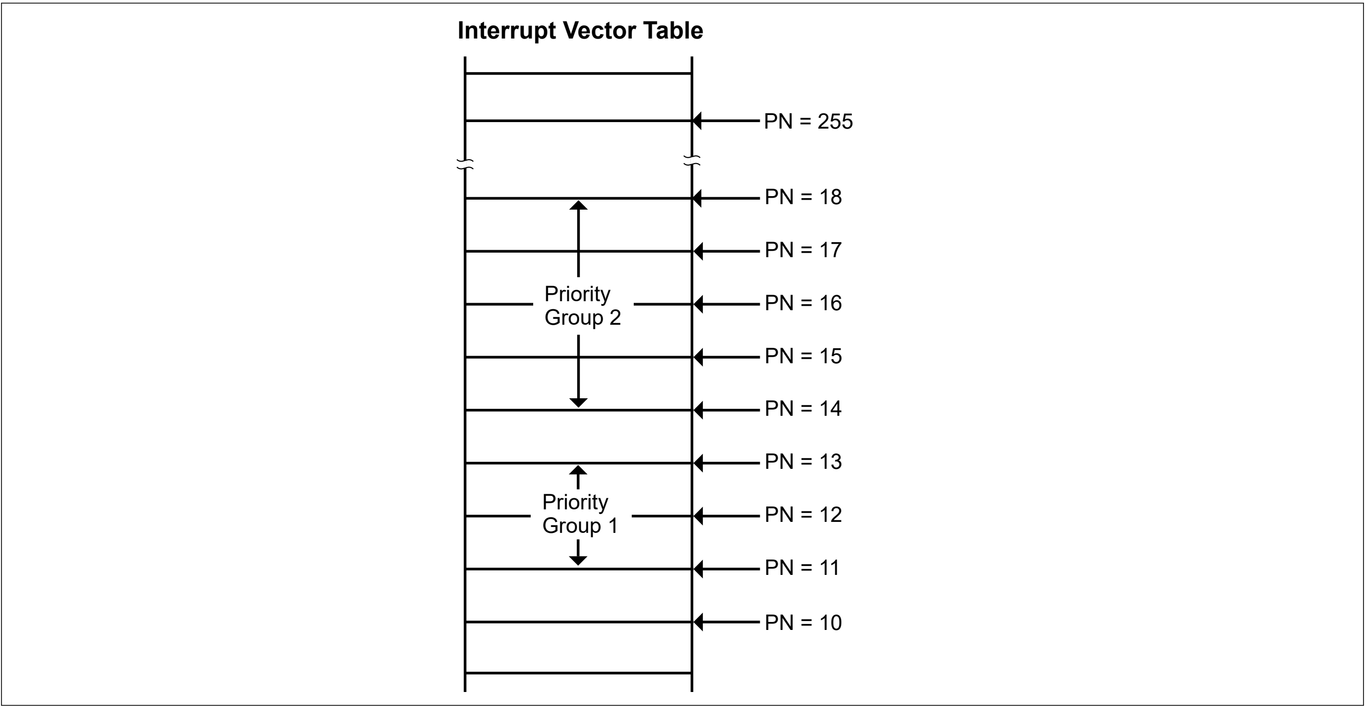


Figure 23 Interrupt priority groups

The interrupt requests with the priority numbers 11 and 12 form one group while the requests with priority numbers 14 to 17 inclusive form another group. Every time one of the interrupts from group one is serviced, the service routine sets the CCPN to 12, the highest number in that group, before re-enabling the interrupt system.

Every time one of the interrupts from group two is serviced, the service routine sets the CCPN to 17 before re-enabling the interrupt system. If interrupt 14 is serviced for example, it can only be interrupted by requests with a priority number higher than 17, but not through a request from its own priority group or requests with lower priority.

One can see the flexibility of this system and its superiority over systems with fixed priority levels. In the example above, the interrupt request with priority number 13 forms its own single member 'group'. Setting the CCPN to the maximum number 255 in each service routine has the same effect as not enabling the interrupt system again; that means all interrupt requests can be considered to be in one group.

The flexibility for interrupt priority levels ranges from all interrupts being in one group, to each interrupt request building its own group, and all possible combinations in between.

5.5.3 Dividing ISRs into different priorities

Interrupt service routines can be easily divided into parts with different priorities. For example, an interrupt is placed on a very high priority because response time and reaction to an event is critical, but further operations in that service routine can run on a lower priority. In this instance the service routine would be divided into two parts, one containing the critical actions, the other part the less critical ones.

The priority of the interrupt node is first set to the high priority, so that when the interrupt occurs the necessary actions are carried out immediately. The priority level of this interrupt is then lowered and the interrupt request bit is set again through software (indicating a pending interrupt) while still in the service routine. Returning to the interrupted program terminates the high priority service routine. The pending interrupt is serviced when the CPU priority is lower than its own priority. After entering the service routine, which is now at a different address in the program memory, the outstanding but low-priority actions of the interrupt are performed.

In other instances the priority of a service request might be low because the response time to an event is not critical, but once it has been granted service it should not be interrupted. To prevent any interruption the

TriCore™ architecture allows the priority level of the service request to be raised within the ISR, and also allows interrupts to be completely disabled.

5.5.4 Using different priorities for the same interrupt source

For some applications the priority of an interrupt request in relation to other requests is not fixed, but depends on the current situation in the system. This can be achieved simply by assigning different Service Request Priority Numbers (SRPNs) at different times to an interrupt source depending on the application needs. Usually the ISR for that interrupt executes different code depending on its priority.

In traditional interrupt systems, the ISR would have to check the current priority of that interrupt request and perform a branch to the appropriate code section, causing a delay in the response to the request. In the TriCore™ system however, the interrupt will automatically have different vector entries for the different priorities. An extra check and branch in the ISR is not necessary, therefore the interrupt latency is reduced.

In case the ISR is independent of the interrupt's priority, branches need to be placed to the common ISR code on each of the vector entries for that interrupt.

***Note**: The use of different priority numbers for one interrupt has to be taken into consideration when creating the vector table.*

5.5.5 Interrupt control registers

Two CSFRs support interrupt handling:

- ICR: Interrupt control register (ICR)
- BIV: Base interrupt vector table pointer (BIV)

The ICR holds the Current CPU Priority Number (CCPN), the enable/disable bit for the Interrupt System (IE), the Pending Interrupt Priority Number (PIPN), and an implementation specific control for the interrupt arbitration scheme. The BIV register holds the base addresses for the interrupt vector tables. Special instructions control the enabling and disabling of the interrupt system. For more information, see Interrupt system.

5.5.5.1 Interrupt control register (ICR)

The Interrupt Control register is defined as follows:

ICR Address: FE2C$_H$
Interrupt control Reset value: 0000 0000$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES								**PIPN**							
-								rh							

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IE	**RES**							**CCPN**							
rwh	-							rwh							

Field	Bits	Type	Description
RES	31:24	-	Reserved

(table continues...)

(continued)

Field	Bits	Type	Description
PIPN	23:16	rh	Pending interrupt priority number A read-only bit-field that indicates the priority number of the pending service request. ICR.PIPN is set to 0 when no request is pending, and at the beginning of each new arbitration process. 00_H : No valid pending request. 01_H : Request pending, lowest priority. … FF_H : Request pending, highest priority.
IE	15	rwh	CPU interrupt enable bit The interrupt enable bit enables this CPU's service request system. ICR.IE is automatically updated by hardware on entry and exit of an Interrupt Service Routine (ISR). ICR.IE is cleared to 0 when an interrupt is taken, and is restored to the previous value when the ISR executes an RFE instruction to terminate itself. ICR.IE can also be updated through the execution of the ENABLE, DISABLE, RESTORE, MTCR and BISR instructions. 0_B : Interrupt system is disabled for this CPU. 1_B : Interrupt system is enabled for this CPU.
RES	14:8	-	Reserved Field
CCPN	7:0	rwh	Current CPU priority number The Current CPU Priority Number (CCPN) bit-field indicates the current priority level of the CPU. It is automatically updated by hardware on entry or exit of Interrupt Service Routines (ISRs) and through the execution of a BISR instruction. CCPN can also be updated through an MTCR instruction.

5.5.5.2 Base interrupt vector table pointer (BIV)

The BIV register contains the base address of the interrupt vector table. When an interrupt is accepted, the entry address into the interrupt vector table is generated from the priority number (taken from the PIPN) of that interrupt, left shifted by either three or five bits, and then ORd with the contents of the BIV register. The left-shift of the interrupt priority number results in a spacing of either 8-byte or 32-byte between the individual entries in the vector table dependent on the vector spacing selected by the VSS bit.

BIV		Address:	$FE20_H$
Base interrupt vector table pointer		Reset value:	Implementation $specific_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BIV															
rw															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIV															**VSS**
rw															rw

Field	Bits	Type	Description
BIV	31:1	rw	Base address of interrupt vector table The address in the BIV register must be aligned to an even byte address (halfword address). Because of the simple ORing of the left-shifted priority number and the contents of the BIV register, the alignment of the base address of the vector table must be to a power of two boundary, dependent on the number of interrupt entries used.
VSS	0	rw	Vector spacing select 0 : 32-Byte Vector Spacing 1 : 8-Byte Vector Spacing

6 Trap system

A trap occurs as a result of an event such as a Non-maskable Interrupt (NMI), an instruction exception, memory-management exception or an illegal access. Traps are always active; they cannot be disabled by software action. This chapter describes the different traps that can occur and the TriCore™ architecture's trap handling mechanism.

Related information

Hypervisor trap system on page 176

6.1 Trap types

The TriCore™ architecture specifies eight general classes for traps. Each class has its own trap handler, accessed through a trap vector of 32 bytes per entry, indexed by the hardware-defined trap class number. Within each class, specific traps are distinguished by a Trap Identification Number (TIN) that is loaded by hardware into register D[15] before the first instruction of the trap handler is executed. The trap handler must test and branch on the value in D[15] to reach the sub-handler for a specific TIN.

Traps can be further classified as synchronous or asynchronous, and as hardware or software generated. These are explained after the following table which lists the trap classes, summarizing and classifying the pre-defined set of specific traps within each class.

In the following table: TIN = Trap Identification Number, Synch. = Synchronous, Asynch. = Asynchronous, HW = Hardware, SW = Software.

Table 9 Supported traps

TIN	Name	Synch./ Asynch.	HW/SW	Definition
Class 0 — MMU				
0	VAF	Synch.	HW	Virtual Address Fill
1	VAP	Synch.	HW	Virtual Address Protection
Class 1 — Internal Protection Traps				
1	PRIV	Synch.	HW	Privileged Instruction
2	MPR	Synch.	HW	Memory Protection Read
3	MPW	Synch.	HW	Memory Protection Write
4	MPX	Synch.	HW	Memory Protection Execute
5	MPP	Synch.	HW	Memory Protection Peripheral Access
6	MPN	Synch.	HW	Memory Protection Null Address
7	GRWP	Synch.	HW	Global Register Write Protection
Class 2 — Instruction Errors				
1	IOPC	Synch.	HW	Illegal Opcode
2	UOPC	Synch.	HW	Unimplemented Opcode
3	OPD	Synch.	HW	Invalid Operand specification
4	ALN	Synch.	HW	Data Address Alignment
5	MEM	Synch.	HW	Invalid Local Memory Address

(table continues...)

Table 9 (continued) Supported traps

TIN	Name	Synch./ Asynch.	HW/SW	Definition
6	CSE	Synch.	HW	Coprocessor Trap Synchronous Error
Class 3 — Context Management				
1	FCD	Synch.	HW	Free Context List Depletion (FCX = LCX)
2	CDO	Synch.	HW	Call Depth Overflow (CALL with PSW.CDC.COUNT at maximum level)
3	CDU	Synch.	HW	Call Depth Underflow (RET with PSW.CDC.COUNT == zero)
4	FCU	Synch.	HW	Free Context List Underflow (FCX = 0)
5	CSU	Synch.	HW	Call Stack Underflow (PCX = 0)
6	CTYP	Synch.	HW	Context Type (PCXI.UL wrong)
7	NEST	Synch.	HW	Nesting Error: RFE with non-zero call depth
Class 4 — System Bus and Peripheral Errors				
1	PSE	Synch.	HW	Program Fetch Synchronous Error
2	DSE	Synch.	HW	Data Access Synchronous Error
3	DAE	Asynch.	HW	Data Access Asynchronous Error
4	CAE	Asynch.	HW	Coprocessor Trap Asynchronous Error
5	PIE	Synch.	HW	Program Memory Integrity Error
6	DIE	Asynch.	HW	Data Memory Integrity Error
7	TAE	Asynch.	HW	Temporal Asynchronous Error
Class 5— Assertion Traps				
1	OVF	Synch.	SW	Arithmetic Overflow
2	SOVF	Synch.	SW	Sticky Arithmetic Overflow
Class 6 — System Call[1)]				
	SYS	Synch.	SW	System Call
Class 7 — Non-Maskable Interrupt				
0	NMI	Asynch.	HW	Non-Maskable Interrupt

1) For the system call trap, the TIN is taken from the immediate constant specified in the SYSCALL instruction. The range of values that can be specified is 0 to 255, inclusive.

6.1.1 Synchronous traps

Synchronous traps are associated with the execution or attempted execution of specific instructions, or with an attempt to access a virtual address that requires the intervention of the memory-management system. The instruction causing the trap is known precisely. The trap is taken immediately and serviced before execution can proceed beyond that instruction.

6.1.2 Asynchronous traps

Asynchronous traps are similar to interrupts, in that they are associated with hardware conditions detected externally and signaled back to the core. Some result indirectly from instructions that have been previously executed, but the direct association with those instructions has been lost. Others, such as the Non-Maskable Interrupt (NMI), are external events. The difference between an asynchronous trap and an interrupt is that asynchronous traps are routed through the trap vector instead of the interrupt vector. They can not be masked and they do not change the current CPU interrupt priority number.

6.1.3 Hardware traps

Hardware traps are generated in response to exception conditions detected by the hardware. In most, but not all cases, the exception conditions are associated with the attempted execution of a particular instruction. Examples are the illegal instruction trap, memory protection traps and data memory misalignment traps. In the case of the MMU traps (trap class 0), the exception condition is either the failure to find a TLB (Translation Lookaside Buffer) entry for the virtual page referenced by an instruction (VAF trap), or an access violation for that page (VAP trap).

6.1.4 Software traps

Software traps are generated as an intentional result of executing a system call or an assertion instruction. The supported assertion instructions are TRAPV (Trap on overflow) and TRAPSV (Trap on sticky overflow). System calls are generated by the SYSCALL instruction. System call traps are described further in System call (trap class 6).

6.1.5 Unrecoverable traps

An unrecoverable trap is one from which software can not recover; that means the task that raised the trap can not be simply restarted.

In the TriCore™ architecture, FCU (a fatal context trap) is an unrecoverable error. See FCU - Free context list underflow (TIN 4) for more information.

6.2 Trap handling

The actions taken on traps by the trap handling mechanisms are slightly different from those taken on external interrupts. A trap does not change the CPU interrupt priority, so the ICR.CCPN field is not updated.

Related information

Exception priorities on page 76

6.2.1 Trap vector format

The trap handler vectors are stored in code memory in the trap vector table. The BTV register specifies the base address of the trap vector table. The vectors are made up of a number of short code blocks, evenly spaced by eight words.

If a trap handler is very short it may fit entirely within the eight words available in the vector code block. If it does not fit the vector code block then it should contain some initial instructions, followed by a jump to the rest of the handler.

6.2.2 Accessing the trap vector table

When a trap occurs, a trap identifier is generated by hardware. The trap identifier has two components:

- The trap class number (TCN) used to index into the trap vector table
- The trap identification number (TIN) which is loaded into the data register D[15]

The trap class number is left shifted by five bits and ORd with the address in the BTV register to generate the entry address of the trap handler.

6.2.3 Return address (RA)

The return address is saved in the return address register A[11].

For a synchronous trap, the return address is the PC of the instruction that caused the trap. Only the SYS trap and FCD trap are different. On a SYS trap, triggered by the SYSCALL instruction, the return address points to the instruction immediately following SYSCALL. The behavior for the FCD trap is described in FCD - Free context list depletion (TIN 1).

For an asynchronous trap, the return address is that of the instruction that would have been executed next, if the asynchronous trap had not been taken. The return address for an interrupt follows the same rule.

6.2.4 Trap vector table

The entry-points of all trap service routines are stored in memory in the trap vector table. The BTV register specifies the base address of the trap vector table in memory. The trap vector table can be located in any available code memory. The BTV register can be modified using the MTCR instruction during the initialization phase of the system, (the BTV register is ENDINIT protected). This arrangement makes it possible to have multiple trap vector tables and switch between them by changing the contents of the BTV register.

When a trap event occurs, a trap identifier is generated by the hardware detecting the event. The trap identifier is made up of a trap class number (TCN) and a trap identification number (TIN).

The TCN is left-shifted by five bits and ORd with the address in the BTV register to form the entry address of the TSR. Because of this operation, it is recommended that bits [7:5] of register BTV are set to 0 (see Figure 24). Note that bit 0 of the BTV register is always 0 and can not be written to (instructions have to be aligned on even byte boundaries).

Left-shifting the TCN by 5 bits creates entries into the trap vector table which are evenly spaced 8 words apart. If a trap handler (TSR) is very short, it may fit entirely within the eight words available in the trap vector table entry. Otherwise, the code at the entry point must ultimately cause a jump to the rest of the TSR residing elsewhere in memory.

Unlike the interrupt vector table, entries in the trap vector table cannot be spanned.

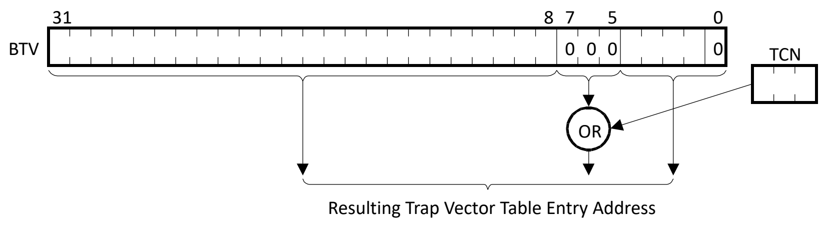


Figure 24 **Trap vector table entry address calculation**

6.2.5 Entering a trap handler

The initial state when a trap occurs is defined as follows:

- The upper context is saved

- The return address in A[11] is updated
- D[15] is set to the TIN
- The stack pointer in A[10] is set to the Interrupt Stack Pointer (ISP) when the processor was not previously using the interrupt stack (in case of PSW.IS = 0). The stack pointer bit is set for using the interrupt stack: PSW.IS = 1
- The I/O mode is set to Supervisor mode, which means all permissions are enabled: PSW.IO = 10_B
- The current protection register set is set to 0: PSW.PRS = 000_B
- The Call Depth Counter (CDC) is cleared, and the call depth limit is set for 64: PSW.CDC = 0000000_B
- Call depth counter is enabled, PSW.CDE = 1
- PSW safety bit is set to value defined in the CORECON register. PSW.S = CORECON.TS
- Write permission to global registers A[0], A[1], A[8], A[9] is disabled: PSW.GW = 0
- The interrupt system is disabled for this CPU: ICR.IE = 0. The 'old' ICR.IE and ICR.CCPN are saved into PCXI.PIE and PCXI.PCPN respectively. ICR.CCPN remains unchanged
- The trap vector table is accessed to fetch the first instruction of the trap handler

Although traps leave the ICR.CCPN unchanged, their handlers still begin execution with interrupts disabled. They can therefore perform critical initial operations without interruptions, until they specifically re-enable interrupts.

For the non-recoverable FCU trap, the initial state is different. The upper context cannot be saved. Only the following states are guaranteed:

- The TIN is loaded into D[15]
- The stack pointer in A[10] is set to the Interrupt Stack Pointer (ISP) when the processor was not previously using the interrupt stack (in case of PSW.IS == 0)
- The I/O mode is set to Supervisor mode, which means all permissions are enabled: PSW.IO = 10_B
- The current protection register set is set to 0: PSW.PRS = 000_B
- PSW safety bit is set to value defined in the CORECON register: PSW.S = CORECON.TS
- The interrupt system is disabled for this CPU: ICR.IE = 0. ICR.CCPN remains unchanged
- The trap vector table is accessed to fetch the first instruction of the FCU trap handler

6.3 Trap descriptions

The following sub-sections describe the trap classes and specific traps listed in Trap types.

6.3.1 MMU traps (trap class 0)

For those implementations that include a Memory Management Unit (MMU), trap class 0 is reserved for MMU traps. There are two traps within this class, VAF and VAP.

VAF - Virtual address fill (TIN 0)

The VAF trap is generated when the MMU is enabled and the virtual address referenced by an instruction does not have a page entry in the MMU Translation Lookaside Buffer (TLB).

VAP - Virtual address protection (TIN 1)

The VAP trap is generated (when the MMU is enabled) by a memory access undergoing Page Table Entries (PTE) translation that is not permitted by the PTE protection settings, or by a User-0 mode access to an upper segment that does not have the privileged peripheral property.

6.3.2 Internal protection traps (trap class 1)

Trap class 1 is for traps related to the internal protection system. The memory protection traps in this class, MPR, MPW, and MPX, are for the range-based protection system and are independent of the page-based VAP protection trap of trap class 0. See the Memory protection system for more details.

All memory protection traps (MPR, MPW, MPX, MPP, and MPN), are based on the virtual addresses that undergo direct translation.

The following internal protection traps are defined:

PRIV - Privilege violation (TIN 1)

A program executing in one of the user modes (User-0 or User-1 mode) attempted to execute an instruction not allowed by that mode.

A table of instructions which are restricted to Supervisor mode or User-1 mode, is supplied in the instruction set chapter of volume 2 of this manual.

MPR - Memory protection read (TIN 2)

The MPR trap is generated when the memory protection system is enabled and the effective address of a load, or atomic read and write sequence, or cache instruction does not lie within any range with read permissions enabled. This trap is not generated when an access violation occurs during a context save/restore operation.

MPW - Memory protection write (TIN 3)

The MPW trap is generated when the memory protection system is enabled and the effective address of a store, an atomic read and write sequence, or cache instruction does not lie within any range with write permissions enabled.

MPX - Memory protection Execute (TIN 4)

The MPX trap is generated when the memory protection system is enabled and the PC does not lie within any range with execute permissions enabled.

MPP - Memory protection peripheral access (TIN 5)

A program executing in User-0 mode attempted a load or store access to a segment that is configured to be a peripheral segment. See Programmable memory access register-2 (PMA2).

MPN - Memory protection null address (TIN 6)

The MPN trap is generated whenever any program attempts a load/store operation to effective address zero.

GRWP - Global register write protection (TIN 7)

A program attempted to modify one of the global address registers (A[0], A[1], A[8] or A[9]) when it did not have permission to do so.

6.3.3 Instruction errors (trap class 2)

Trap class 2 is for signaling various types of instruction errors. Instruction errors include errors in the instruction opcode, in the instruction operand encodings, or for memory accesses, in the operand address.

IOPC - Illegal opcode (TIN 1)

An invalid instruction opcode was encountered. An invalid opcode is one that does not correspond to any instruction known to the implementation.

TRAPINV is guaranteed to be an opcode which will always raise the IOPC trap.

UOPC - Unimplemented opcode (TIN 2)

An unimplemented opcode was encountered. An unimplemented opcode corresponds to a known instruction that is not implemented in a given hardware implementation. The instruction may be implemented through software emulation in the trap handler.

Example UOPC conditions are:

- A MMU instruction if the MMU is not present
- A FPU instruction if the FPU is not present
- An external coprocessor instruction if the external coprocessor is not present

OPD - Invalid operand (TIN 3)

The OPD trap may be raised for instructions that take an even-odd register pair as an operand, if the operand specifier is odd. The OPD trap may also be raised for instructions that take a quad register as an operand, if the specifier is not on of [0, 4, 8, 12]. The OPD trap may also be raised for other cases where operands are invalid.

The OPD may also be raised by move to and from core registers (MTCR, MFCR, MTDCR, MFDCR) instructions targeting non-existing or protected registers.

Implementations are not architecturally required to raise this trap, and may treat invalid operands in an implementation defined manner.

ALN - Data address alignment (TIN 4)

An ALN trap is raised when the address for a data memory operation does not conform to the required alignment rules. See Alignment requirements, for more information on these rules. An ALN trap is also raised when the size, length or index of a circular buffer is incorrect. See Circular addressing for more details.

MEM - Invalid memory address (TIN 5)

The MEM trap is raised when the address of an access can be determined to either violate an architectural constraint or an implementation constraint.

Defined MEM trap subclasses are different segment, segment crossing, CSFR access, CSA restriction and scratch range.

An implementation must define which implementation constraint MEM traps it will raise, or the alternative behavior if the MEM trap is not raised. It must also document any other implementation specific MEM traps it will raise.

Architectural constraints which will raise the MEM trap are:

- An addressing mode that adds an offset to a base address results in an effective address that is in a different segment to the base address (different segment)
- A data element is accessed with an address, such that the data object spans the end of one segment and the beginning of another segment (segment crossing)

Implementation constraints which can raise the MEM trap are:

- A memory address is used to access a core SFR (CSFR) rather than using a MTCR/MFCR instruction (CSFR access)
- A memory address is used for a CSA access and it is not valid for the implementation to place CSA there (CSA restriction)
- An access to Scratch memory is attempted using a memory address which lies outside the implemented region of memory (scratch range error)

CSE - Coprocessor trap synchronous error (TIN 6)

The CSE trap is generated by a coprocessor to report an error on input operands . Examples of typical errors that can cause a CSE trap are floating point unit invalid operations. CSE is shared amongst all coprocessors in a given system. A trap handler must therefore inspect the opcode to determine the coprocessor involved and potentially other state to determine the cause of a trap.

6.3.4 Context management (trap class 3)

Trap class 3 is for exception conditions detected by the context management subsystem, in the course of performing (or attempting to perform) context save and restore operations connected to function calls, interrupts, traps, and returns.

FCD - Free context list depletion (TIN 1)

The FCD trap is generated after a context save operation, when the operation causes the free context list to become 'almost empty'. The 'almost empty' condition is signaled when the CSA used for the save operation is the one pointed to by the context list limit register LCX. The operation responsible for the context save completes normally and then the FCD trap is taken.

If the operation responsible for the context save was the hardware interrupt or trap entry sequence, then the FCD trap handler will be entered before the first instruction of the original interrupt or trap handler is executed. The return address for the FCD trap will point to the first instruction of the interrupt or trap handler.

The FCD trap handler is normally expected to take some form of action to rectify the context list depletion. The nature of that action is OS dependent, but the general choices are to allocate additional memory for CSA storage, or to terminate one or more tasks, and return the CSAs on their call chains to the free list. A third possibility is not to terminate any tasks outright, but to copy the call chains for one or more inactive tasks to uncached external or secondary memory that would not be directly usable for CSA storage, and release the copied CSAs to the free list. In that instance the OS task scheduler would need to recognize that the inactive task's call chain was not resident in CSA storage, and restore it before dispatching the task.

The FCD trap itself uses one additional CSA beyond the one designated by the LCX register, so LCX must not point to the actual last entry on the free context list. In addition, it is possible that an asynchronous trap condition, such as an external bus error, will be reported after the FCD trap has been taken, interrupting the FCD trap handler and using one more CSA. Therefore, to avoid the possibility of a context list underflow, the free context list must include a minimum of two CSAs beyond the one pointed to by the LCX register. If the FCD trap handler makes any calls, then additional CSA reserves are needed.

In order to allow the trap handlers for asynchronous traps to recognize when they have interrupted the FCD trap handler, the FCDSF flag in the CORECON (core configuration) register is set whenever an FCD trap is generated. The FCDSF bit should be tested by the handler for any asynchronous trap that could be taken while an FCD trap is being handled. If the bit is found to be set, the asynchronous trap handler must avoid making any calls, but should queue itself in some manner that allows the OS to recognize that the trap occurred. It should then carry out an immediate return, back to the interrupted FCD trap handler. See Core register table.

CDO - Call depth overflow (TIN 2)

A program attempted to execute a CALL instruction with the call depth counter enabled and the call depth count value (PSW.CDC.COUNT) at its maximum value. Call depth counting guards against context list depletion, by enabling the OS to detect 'runaway recursion' in executing tasks.

CDU - Call depth underflow (TIN 3)

A program attempted to execute a RET (return) instruction with the call depth counter enabled and the call depth count value (PSW.CDC.COUNT) at zero. A call depth underflow does not necessarily reflect a software error in the currently executing task. An OS can achieve finer granularity in call depth counting by using a deliberately narrow call depth counter, and incrementing or decrementing a separate software counter for the current task on each call depth overflow or underflow trap. A program error would be indicated only if the software counter were already zero when the CDU trap occurred.

FCU - Free context list underflow (TIN 4)

The FCU trap is taken when a context save operation is attempted but the free context list is found to be empty (that means the FCX register contents are null). The FCU trap is also taken if any error is encountered during a context save or restore operation. The context operation cannot be completed. Instead a forced jump is made to the FCU trap handler and D15 updated with the FCU TIN value. Any pending asynchronous exception may be lost when an FCU condition occurs.

In failing to complete the context save or restore, architectural state is lost, so the occurrence of an FCU trap is a non-recoverable system error. The FCU trap handler should ultimately initiate a system reset.

CSU - Call stack underflow (TIN 5)

Raised when a context restore operation is attempted and when the contents of the PCX register were null. This trap indicates a system software error (kernel or OS) in task setup or context switching among software managed tasks. No software error or combination of errors in a user task can generate this condition, unless the task has been allowed write permission to the context save areas which, in itself, can be regarded as a system software error.

CTYP - Context type (TIN 6)

Raised when a context restore operation is attempted but the context type, as indicated by the PCXI.UL bit, is incorrect for the type of restore attempted; that means a restore lower context is attempted when PCXI.UL == 1, or a restore upper context is attempted when PCXI.UL == 0. As with the CSU trap, this indicates a system software error in context list management.

NEST - Nesting error (TIN 7)

A program attempted to execute an RFE (return from exception) instruction with the call depth counter enabled and the call depth count value (PSW.CDC.COUNT) non-zero. The return from an interrupt or trap handler should normally occur within the body of the interrupt or trap handler itself, or in code to which the handler has branched, rather than code called from the handler. If this is not the case there will be one or more saved contexts on the residual call chain that must be popped and returned to the free list, before the RFE can be legitimately issued.

6.3.5 System bus and peripheral errors (trap class 4)

PSE - Program fetch synchronous error (TIN 1)

The PSE trap is raised when:

- A bus error[3] occurred because of an instruction fetch
- An instruction fetch targets a segment that does not have the code fetch property. See Programmable memory access register-1 (PMA1)

DSE - Data access synchronous error (TIN 2)

The DSE trap is raised when:

- Whenever a bus error occurs because of a data load operation
- In the case of a data load or store operation from Data Scratchpad RAM (DSPR) (Scratchpad RAM) where the access is beyond the end of the memory range
- In the case of an error during the data load phase of a data cache refill

***Note**: There are implementation-dependent registers for DSE which can be interrogated to determine the source of the error more precisely. Refer to the User's Manual for a specific TriCore™ implementation for more details.*

DAE - Data access asynchronous error (TIN 3)

The DAE trap is raised when the memory system reports back an error which cannot immediately be linked to a currently executing instruction. Generally this means an error returned on the system bus from a peripheral or external memory.

3 A bus fetch error is also generated for an instruction fetch to the data scratch pad RAM region (D000 0000_H to D3FF $FFFF_H$) when the memory access is outside the range of the actual scratchpad RAMs

This DAE trap is raised when:

- A bus error occurred because of a data store operation
- There is an error caused by a cache management instruction
- There is an error caused by a cache line writeback

***Note**: There are implementation-dependent registers for DAE which can be interrogated to determine the source of the error more precisely. Refer to the User's Manual for a specific TriCore™ implementation for more details.*

CAE - Coprocessor trap asynchronous error (TIN 4)

This CAE asynchronous trap is generated by a coprocessor to report an error.

Examples of typical errors that can cause a CAE trap are floating point unit arithmetic errors.

CAE is shared amongst all coprocessors in a given CPU. A trap handler must therefore inspect the opcode to determine the coprocessor involved and potentially other state to determine the cause of a trap.

PIE - Program memory integrity error (TIN 5)

The PIE trap is raised whenever an uncorrectable memory integrity error is detected in an instruction fetch. The trap is synchronous to the erroneous instruction. A PIE trap is raised if any element within the fetch group contains an unrecoverable error. Hardware is not required to localize the error to a particular instruction.

An implementation may provide additional registers that can be interrogated to determine the source of the error more precisely. Refer to the user manual for a specific TriCore™ implementation for more details.

DIE - Data memory integrity error (TIN 6)

The DIE trap is raised whenever an uncorrectable memory integrity error is detected in a data access.

Implementations may choose to implement the DIE trap as either an asynchronous or synchronous trap.

A DIE trap is raised if any element accessed by a load or store contains an uncorrectable error. Hardware is not required to localize the error to the access width of the operation. DIE traps raised during context operations may result in loss of data.

An implementation may provide additional registers that can be interrogated to determine the source of the error more precisely. Refer to the user manual for a specific TriCore™ implementation for more details.

TAE - Temporal asynchronous error (TIN 7)

The TAE asynchronous trap is raised by the temporal protection system whenever an active timer decrements to zero. this may be used to guard against task overrun in time critical applications.

6.3.6 Assertion traps (trap class 5)

OVF - Arithmetic overflow (TIN 1)

Raised by the TRAPV instruction, if the overflow bit in the PSW is set (PSW.V == 1).

SOVF - Sticky arithmetic overflow (TIN 2)

Raised by the TRAPSV instruction, if the sticky overflow bit in the PSW is set (PSW.SV == 1).

6.3.7 System call (trap class 6)

SYS - System call (TIN = 8-bit unsigned immediate constant in SYSCALL)

The SYS trap is raised immediately after the execution of the SYSCALL instruction, to initiate a system call. The TIN that is loaded into D[15] when the trap is taken is specified as an 8-bit unsigned immediate constant in the SYSCALL instruction. The return address points to the instruction immediately following the SYSCALL.

6.3.8 Non-maskable interrupt (trap class 7)

NMI - Non-maskable interrupt (TIN 0)

The causes for raising a non-maskable interrupt are implementation dependent. Typically there is an external pin that can be used to signal the NMI, but it may also be raised in response to such things as a watchdog timer interrupt, or an impending power failure. Refer to the user's manual for a specific TriCore™ implementation for more details.

6.3.9 Debug traps

BBM - Break before make/BAM - Break after make

Please refer to the core debug controller chapter for information on debug traps. See Core debug controller.

6.4 Exception priorities

The priority order between an asynchronous trap, a synchronous trap, and an interrupt from the software architecture model, is as follows:

1. Asynchronous trap (highest priority)
2. Synchronous trap
3. Interrupt (lowest priority)

The following trap rules must also be considered:

1. The older the instruction in the instruction sequence which caused the trap, the higher the priority of the trap. All potential traps with lower priorities are void
2. Attempting to save a context with an empty free context list (FCX = 0) results in a FCU (free context list underflow) trap. This trap takes priority over all other exceptions
3. When the same instruction causes several synchronous traps anywhere in the pipeline, priorities follow those shown in the table below
4. Priority number 1 indicates the highest priority

Table 10 Synchronous trap priorities

Priority	Type of trap
Instruction fetch traps	
1	Breakpoint trap or halt - BBM (trigger on PC)
2	VAF-P[1]
3	VAP-P[1]
4	MPX
5	PSE

(table continues...)

Table 10 (continued) Synchronous trap priorities

Priority	Type of trap
6	PIE
Instruction format traps	
7	IOPC
8	OPD (instruction operands excluding CSFR checks)
9	UOPC
Instruction traps	
10	Breakpoint trap or halt - BBM (trigger on address, MxCR, MxDCR, debug)
11	PRIV
12	GRWP
13	SYS
14	OPD (CSFR checks)
15	CSE
Context traps	
16	FCD
17	FCU
18	CSU
19	CDO
20	CDU
21	NEST
22	CTYP
Data memory access traps	
23	MEM
24	ALN
25	MPN
26	VAF-D[1)]
27	VAP-D[1)]
28	MPP
29	MPR
30	MPW
31	DSE
General data traps	
32	SOVF
33	OVF
34	Breakpoint trap or halt - BAM

1) Only applicable if an MMU is present and enabled.

Table 11 Asynchronous trap priorities

Priority	Asynchronous traps
1	NMI
2	DAE
3	CAE
4	TAE
5	DIE

Related information

Exception priorities (virtualization present and enabled) on page 180

6.5 Trap control registers

6.5.1 Base trap vector table pointer (BTV)

The BTV contains the base address of the trap vector table. When a trap occurs, the entry address into the trap vector table is generated from the Trap Class of that trap, left-shifted by 5 bits and then ORd with the contents of the BTV register. The left-shift of the Trap Class results in a spacing of 8 words (32 bytes) between the individual entries in the vector table.

***Note**: This register is ENDINIT protected.*

BTV	Address:	$FE24_H$
Base trap vector table pointer	Reset value:	Implementation specific$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BTV															
rw															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BTV															**RES**
rw															-

Field	Bits	Type	Description
BTV	31:1	rw	Base address of trap vector table The address in the BTV register must be aligned to an even byte address (halfword address). Also, due to the simple ORing of the left-shifted trap identification number and the contents of the BTV register, the alignment of the base address of the vector table must be to a power of two boundary. There are eight different trap classes, resulting in Trap Classes from 0 to 7. The contents of BTV should therefore be set to at least a 256-byte boundary (8 Trap Classes * 8 word spacing).
RES	0	-	Reserved

6.5.2 Program synchronous error trap register (PSTR)

Implementations may provide information on the type of program synchronous error in the PSTR register. The contents of the register are implementation specific.

PSTR	Address:	9200_H
Program synchronous error trap register	Reset value:	Implementation $specific_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Implementation specific															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Implementation specific															
-															

Field	Bits	Type	Description
Implementation specific	31:0	-	Implementation specific

6.5.3 Data synchronous error trap register (DSTR)

Implementations may provide information on the type of data synchronous error in the DSTR register. The contents of the register are implementation specific.

DSTR	Address:	9010_H
Data synchronous error trap register	Reset value:	Implementation $specific_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Implementation specific															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Implementation specific															
-															

Field	Bits	Type	Description
Implementation specific	31:0	-	Implementation specific

6.5.4 Data asynchronous error trap register (DATR)

Implementations may provide information on the type of data asynchronous error in the DATR register. The contents of the register are implementation specific.

DATR	Address:	9018_H
Data asynchronous error trap register	Reset value:	Implementation $specific_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Implementation specific															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Implementation specific															
-															

Field	Bits	Type	Description
Implementation specific	31:0	-	Implementation specific

6.5.5 Data error address register (DEADD)

Implementations may provide information on the location of the data error in the DEADD register. The contents of the register are implementation specific.

DEADD	Address:	$901C_H$
Data error address register		Implementation specific$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Implementation specific															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Implementation specific															
-															

Field	Bits	Type	Description
Implementation specific	31:0	-	Implementation specific

7 Memory integrity error mitigation

This chapter describes the architectural features used to support the mitigation of memory integrity errors within the local memories of TriCore™ processors.

7.1 Memory integrity error classification

Memory integrity errors are classified as being either correctable or uncorrectable.

Uncorrectable memory integrity error

If hardware is not able to provide the expected data to the core on accessing a memory element containing a memory integrity error, the memory integrity error is defined as being uncorrectable.

Correctable memory integrity error

If hardware is able to provide the expected data to the core on accessing a memory element containing a memory integrity error, the memory integrity error is defined as being correctable.

Correctable memory integrity errors are further categorized as either resolved or unresolved. Correctable memory integrity errors always provide the correct data to the core. As part of the correction process hardware may also update the erroneous source data in memory with the corrected data. Such a memory integrity error is defined as being resolved. If the erroneous source data in memory is not updated the memory integrity error is defined as being unresolved.

7.2 Memory integrity error traps

When an uncorrectable memory integrity error is encountered either a PIE (Program Memory Integrity Error) or DIE (Data Memory Integrity Error) trap is raised.

7.2.1 Program memory integrity error (PIE)

The PIE trap is raised when an uncorrectable memory integrity error is detected in an instruction fetch from a local memory. The trap is synchronous to the erroneous instruction. The trap is of class 4, TIN 5.

A PIE trap is raised if any element within the fetch group contains an unrecoverable error. Hardware is not required to localize the error to a particular instruction.

***Note**: Implementation specific registers can be interrogated to more precisely determine the source of the error. Refer to the user manual for a specific TriCore™ product for details.*

7.2.2 Data memory integrity error (DIE)

The DIE trap is raised when an uncorrectable memory integrity error is detected in a data access to a local memory. The trap is of class 4, TIN 6.

A TriCore™ implementation may choose to implement the DIE trap as either an asynchronous or synchronous trap.

A DIE trap is raised if any element accessed by a load/store contains an uncorrectable error. Hardware is not required to localize the error to the access width of the operation.

***Note**: Implementation specific registers can be interrogated to more precisely determine the source of the error. Refer to the User manual for a specific TriCore™ product for more details.*

7.3 Summary

A detected memory integrity error in local instruction memory will lead to either:

- A correctable error
- An uncorrectable error triggering a PIE trap

A detected memory integrity error in local data memory will lead to either:

- A correctable error
- An uncorrectable error triggering a DIE trap

The actual method used for the detection of memory integrity errors is implementation dependent.

7.4 Integrity error registers

To provide information for memory integrity error handling and debug, a number of implementation specific registers are provided. The contents of these registers are implementation specific.

7.4.1 Program integrity error trap register (PIETR)

This register contains information allowing software to localise the source of the last detected program memory integrity error.

PIETR Address: 9214_H
Program integrity error trap register Reset value: $0000\ 0000_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Implementation specific															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Implementation specific															
-															

Field	Bits	Type	Description
Implementation specific	31:0	-	Implementation specific

7.4.2 Program integrity error address register (PIEAR)

The PIEAR register contains the address accessed by the last operation that caused a program memory integrity error.

PIEAR Address: 9210_H
Program integrity error address register Reset value: $0000\ 0000_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Implementation specific															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Implementation specific															
-															

Field	Bits	Type	Description
Implementation specific	31:0	-	Implementation specific

7.4.3 Data integrity error trap register (DIETR)

The DIETR register contains information allowing software to localise the source of the last detected data memory integrity error.

DIETR Address: 9024_H

Data integrity error trap register Reset value: $0000\ 0000_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Implementation specific															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Implementation specific															
-															

Field	Bits	Type	Description
Implementation specific	31:0	-	Implementation specific

7.4.4 Data integrity error address register (DIEAR)

The DIEAR register contains the address accessed by the last operation that caused a data memory integrity error.

DIEAR Address: 9020_H

Data integrity error address register Reset value: $0000\ 0000_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Implementation specific															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Implementation specific															
-															

Field	Bits	Type	Description
Implementation specific	31:0	-	Implementation specific

8 Address map and memory configuration

This chapter describes the TriCore™ physical address map and the architectural aspects of the memory system.

8.1 Overview

The TriCore™ architecture treats the 4 GBytes (32-bit) of physical address space as being divided into 16 equally sized 256-MByte segments. These segments are numbered from 0_H to F_H and are identified by the upper 4 bits of the address. Different segments may be configured to have different access characteristics as described in this chapter.

8.2 Scratchpad RAM

The TriCore™ architecture supports the use of closely coupled SRAMs known as scratchpad RAMs. Separate SRAMs are supported for both program and data. The program scratchpad RAMs (PSPR) are located in segment C_H. The data scratchpad RAMs (DSPR) are located in segment D_H

The size of the scratchpad RAMs is implementation dependent. Access to a segment outside of the implemented memory size may result in a trap.

In a multiprocessor system the DSPR and PSPR memories of all CPUs are accessible through the DSPR and PSPR image regions in segments 0_H to 7_H.

Table 12 Scratchpad RAM segments

Segment	Properties
D_H	DSPR region
C_H	PSPR region
7_H	CPU-0 PSPR and DSPR memory image region
6_H	CPU-1 PSPR and DSPR memory image region
5_H	CPU-2 PSPR and DSPR memory image region
4_H	CPU-3 PSPR and DSPR memory image region
3_H	CPU-4 PSPR and DSPR memory image region
2_H	CPU-5 PSPR and DSPR memory image region
1_H	CPU-6 PSPR and DSPR memory image region
0_H	CPU-7 PSPR and DSPR memory image region

8.3 Address segments and memory access types

The 4 GBytes (32-bit) of physical address space is divided into 16 equally sized 256-MByte segments. Each segment is selectable as being either peripheral space, cached or non-cached memory. The cacheability of a segment is independently selectable for code fetches and data accesses. The access characteristics (access types) of each segment are selected by the Programmable Memory Access registers (PMA0, PMA1 and PMA2).

8.3.1 Memory access types

The TriCore™ architecture defines three possible memory access types:

8.3.1.1 Cacheable memory

Features of cached memory:

- The cacheability of a segment is independently selectable for code fetches and data accesses
- Code fetches to the memory will be cached by the CPU if a code cache is present and enabled. The CPU is permitted to perform speculative code fetches to the memory
- Data accesses to the memory will be cached by the CPU if a data cache is present and enabled. The CPU is permitted to perform speculative data accesses to the memory

8.3.1.2 Non-cacheable memory

Features of non-cached memory:

- The cacheability of a segment is independently selectable for code fetches and data accesses
- Code fetches to the memory will not be cached by the CPU. The CPU is permitted to perform speculative code fetches to the memory
- Data accesses to the memory will not be cached by the CPU. The CPU is permitted to perform speculative data accesses to the memory

8.3.1.3 Peripheral space

Features of peripheral space:

- Only Supervisor and User-1 mode data accesses are permitted
- User-0 mode data accesses are not permitted and result in an MPP trap
- Code accesses are not permitted and will result in a PSE trap
- All CPU accesses to the memory segment are non-cached
- All CPU accesses to the memory segment are non-speculative
- Context operations and accesses using circular addressing are not permitted

8.3.2 Speculation

An implementation may perform both necessary and speculative accesses.

- Necessary accesses are those required to correctly compute the program and any implementation or simulation of the program execution must perform these accesses
- Speculative accesses are those that an implementation may make in order to improve performance either in correct or incorrect anticipation of a necessary access

Data read accesses and fetch accesses to both cached and non-cached memory may be speculative. The processor may read entire cache lines in physical memory and place them in a buffer for future access. The order of accesses is not guaranteed.

The processor never performs speculative write accesses which are visible in a memory region to other system elements.

8.3.3 Cacheability of segments

Cacheability of segments is subject to the following restrictions.

- Peripheral space may never be cached
- The contents of the local DSPR may never be held in the local data cache
- The contents of the local PSPR may never be held in the local program cache

These restrictions are enforced by hardware independent of the settings of PMA0 or PMA1.

8.3.4 Default memory types for all segments

The default defined memory types are shown in the following table:

Table 13 Default memory access types for all segments

Segment	Memory types
F_H	Peripheral space
E_H	Peripheral space
D_H	Non-cacheable memory
C_H	Non-cacheable memory
B_H	Non-cacheable memory
A_H	Non-cacheable memory
9_H	Cacheable memory
8_H	Cacheable memory
7_H- 0_H	Non-cacheable memory

8.4 Memory configuration register definitions

8.4.1 Programmable memory access register-0 (PMA0)

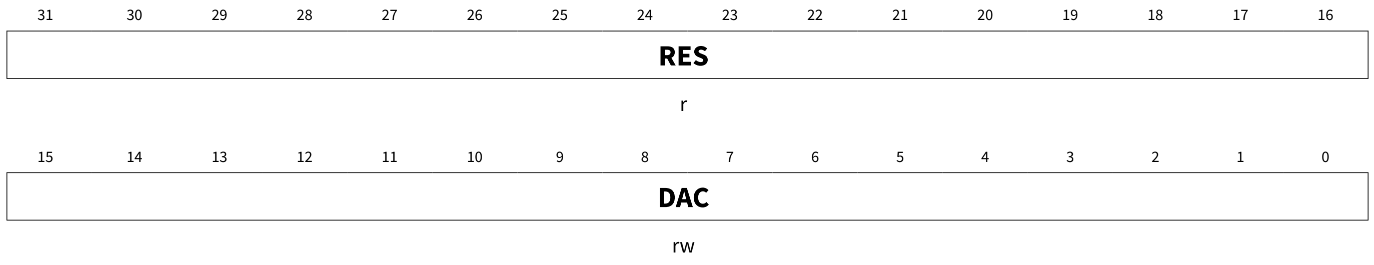
The PMA0 register defines the cacheability of data accesses for each segment in the physical address space. Segment-F is constrained to be peripheral space in all implementations and hence is non-cacheable. Segment-D is constrained to be non-cacheable for data accesses in all implementations. The data cacheability of all other segments is implementation defined.

Note that when changing the value of the PMA0 register, an implementation may require additional operations to be performed in order to maintain coherency of the processors view of memory.

***Note**: This register is ENDINIT protected*

PMA0 Address: 8100_H

Programmable memory access register-0 Reset value: $0000\ 0300_H$



Field	Bits	Type	Description
RES	31:16	r	Reserved
DAC	15:0	rw	Data access cacheability - Implementation defined.

8.4.2 Programmable memory access register-1 (PMA1)

The PMA1 register defines the cacheability of code accesses for each segment in the physical address space. Segment-F is constrained to be peripheral space in all implementations and hence is non-cacheable. Segment-C is constrained to be non-cacheable for code accesses in all implementations. The code cacheability of all other segments is implementation defined.

Note that when changing the value of the PMA1 register, an implementation may require additional operations to be performed in order to maintain coherency of the processors view of memory.

***Note**: This register is ENDINIT protected*

PMA1 Address: 8104_H

Programmable memory access register-1 Reset value: $0000\ 0300_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CAC															
rw															

Field	Bits	Type	Description
RES	31:16	r	Reserved
CAC	15:0	rw	Code accesses cacheability - Implementation defined

8.4.3 Programmable memory access register-2 (PMA2)

The PMA2 register defines the Peripheral Space designator for each segment in the physical address space. Segment-F is constrained to be peripheral space in all implementations The Peripheral Space Designator of all other segments is implementation defined and may be read-write or read-only.

Note that when changing the value of the PMA2 register, an implementation may require additional operations to be performed in order to maintain coherency of the processors view of memory.

If bit[n] of the PMA2 register is set then the segment-n will be seen as uncacheable independent of the settings of PMA0 and PMA1.

***Note**: This register is ENDINIT protected*

PMA2 Address: 8108_H

Programmable memory access register-2 Reset value: $0000\ C000_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSD															
r															

Field	Bits	Type	Description
RES	31:16	r	Reserved
PSD	15:0	r	Peripheral space designator - Implementation defined

8.4.4 Program memory configuration registers (PCON0, PCON1, PCON2)

TriCore™ implementations may control and provide information on the status and configuration of the program cache and scratch memories through the program memory configuration registers. Three registers are architecturally defined for this purpose; PCON0, PCON1 and PCON2.

The contents of these registers (where implemented) is implementation dependent.

Implementations may ENDINIT protect these registers.

8.4.4.1 Program memory configuration register PCON0

PCON0 Address: $920C_H$

Program memory configuration register 0 Reset value: $\text{Implementation specific}_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Implementation specific															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Implementation specific															
-															

Field	Bits	Type	Description
Implementation specific	31:0	-	Implementation specific

8.4.4.2 Program memory configuration registers PCON1

PCON1 Address: 9204_H

Program memory configuration register 1 Reset value: $\text{Implementation specific}_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Implementation specific															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Implementation specific															
-															

Field	Bits	Type	Description
Implementation specific	31:0	-	Implementation specific

8.4.4.3 Program memory configuration registers PCON2

PCON2	Address:	9208_H
Program memory configuration register 2	Reset value:	Implementation specific$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Implementation specific															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Implementation specific															
-															

Field	Bits	Type	Description
Implementation specific	31:0	-	Implementation specific

8.4.5 Data memory configuration registers (DCON0, DCON1, DCON2)

TriCore™ implementations may control and provide information on the status and configuration of the data cache and scratch memories through the data memory configuration registers. Three registers are architecturally defined for this purpose; DCON0, DCON1 and DCON2.

The contents of these registers (where implemented) is implementation dependent.

Implementations may ENDINIT protect these registers.

8.4.5.1 Data memory configuration register DCON0

DCON0	Address:	9040_H
Data memory configuration register 0	Reset value:	Implementation specific$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Implementation specific															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Implementation specific															
-															

Field	Bits	Type	Description
Implementation specific	31:0	-	Implementation specific

8.4.5.2 Data memory configuration register DCON1

DCON1	Address:	9008_H
Data memory configuration register 1	Reset value:	Implementation specific$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Implementation specific															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Implementation specific															
-															

Field	Bits	Type	Description
Implementation specific	31:0	-	Implementation specific

8.4.5.3 Data memory configuration register DCON2

DCON2		Address:	9000_H
Data memory configuration register 2		Reset value:	Implementation specific$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Implementation specific															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Implementation specific															
-															

Field	Bits	Type	Description
Implementation specific	31:0	-	Implementation specific

9 Floating point unit (FPU)

This chapter describes the TriCore™ Floating Point Unit (FPU) architecture. The floating point extensions to the TriCore™ architecture are optional, and may not be present in every core of a system (See TriCore™ core configuration register (TCCON))

The optional single precision FPU (coprocessor 1) is an IEEE 754-2019 compatible floating-point unit to accompany the TriCore™ instruction set.

The optional double precision FPU (coprocessor 2) is an IEEE 754-2019 compliant floating-point unit to accompany the TriCore™ instruction set.

If no FPU is present, then FPU instructions will cause a UOPC (unimplemented opcode) trap.

***Note**: Double precision FPU is only available in conjunction with the single precision FPU.*

9.1 Functional overview

The FPU executes IEEE 754-2019 compatible floating-point arithmetic instructions and supports the following feature set:

- IEEE 754-2019 single precision and double precision formats are supported
- Floating-point add, subtract, multiply, MAC, and divide instructions
- Conversion to or from IEEE 754-2019 single precision format from or to TriCore™ signed and unsigned integers and 32-bit signed fractions (Q31 format)
- QSEED.F and QSEED.DF instructions used to obtain an approximate value intended for use in Newton-Raphson iterations to perform a square-root operation
- Comparison of two floating-point numbers
- All four IEEE 754-2019 rounding modes are implemented
- Asynchronous traps can be generated on selected IEEE 754-2019 exceptions on results
- Synchronous traps can be generated on selected IEEE 754-2019 exceptions on input operand

Restrictions

The FPU has the following restrictions and usage limitations:

- IEEE 754-2019 single precision subnormal numbers are not supported for arithmetic operations
- IEEE 754-2019 compliant remainder function cannot be implemented using FPU instructions because of the effects of multiple rounding when using a sequence of individually rounded instructions
- Using FPU MAC operations can give different results from using separate multiply and accumulate operations because the result is only rounded once at the end of a MAC
- Full compliance with the IEEE 754-2019 standard is not achieved using FPU instructions alone because subnormal numbers are not supported for single precision. To support full compliance requires additional software

9.2 IEEE 754-2019 compliance

9.2.1 IEEE 754-2019 half precision

9.2.1.1 IEEE 754-2019 half precision data format

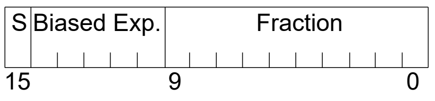


Figure 25 Half precision IEEE 754-2019 floating-point format

The half precision IEEE 754-2019 floating-point format has three sections: a sign bit, an 5-bit biased exponent, and a 10-bit fractional mantissa with an implied binary point before bit 9. For normal numbers the mantissa has an implied 1 immediately to the left of the binary point. The following table shows the different types of number representation in IEEE 754-2019 half precision format. In this table:

- s = bit [15]: sign bit
- e = bits [14:10]: biased exponent
- f = bits [9:0]: fractional part of mantissa

Table 14 IEEE 754-2019 half precision representation types

Condition	Represented value	Description
0 < e < 31	$(-1)^s \times 2^{(e-15)} \times 1.f$	Normal number
e == 0 AND f != 0	$(-1)^s \times 2^{(-14)} \times 0.f$	Subnormal number
e == 0 AND f == 0	$(-1)^s \times 0$	Signed zero
s == 0 AND e == 31 AND f == 0	$-\infty$	+ infinity
s == 1 AND e == 31 AND f == 0	$-\infty$	– infinity
e == 31 AND f != 0 AND f[9] == 0		Signaling NaN[1)]
e == 31 AND f != 0 AND f[9] == 1		Quiet NaN[1)]

1) IEEE 754-2019 does not define how to distinguish between signaling NaNs and quiet NaNs, but bit[9] has become the standard way of doing this.

***Note**: Both signed values of zero are always treated identically and never produce different results except different signed zeros.*

9.2.1.2 IEEE 754-2019 half precision NaNs

NaNs (Not a Number) are bit combinations within the IEEE 754-2019 standard that do not correspond to numbers. There are two types of NaNs: signaling and quiet. The half precision FPU defines signaling NaNs to have bit 9 = ‘0’, and quiet NaNs to have bit 9 = ‘1’.

When invalid double precision operations are performed (including operations with a signaling NaN operand), FI is asserted and a quiet NaN is produced as the floating-point result. The quiet NaN contains information about the origin of the invalid operation; see Table 20.

IEEE 754-2019 suggests that quiet NaNs should be propagated so that the result of an instruction receiving a quiet NaN as an operand (with no signaling NaN operands) should be that quiet NaN. The FPU does not propagate quiet NaNs in this way. The result of an operation that has one (or more) quiet NaN operands and no signaling NaN operands is always the quiet NaN $7FE0_H$.

9.2.1.3 Half precision restrictions

Half precision is only supported as a data interchange format and is limited to conversion functions to and from single precision (HPTOF and FTOHP).

9.2.2 IEEE 754-2019 single precision

9.2.2.1 IEEE 754-2019 single precision data format

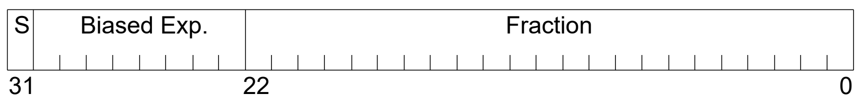


Figure 26 **Single precision IEEE 754-2019 floating-point format**

The single precision IEEE 754-2019 floating-point format has three sections: a sign bit, an 8-bit biased exponent, and a 23-bit fractional mantissa with an implied binary point before bit 22. For normal numbers the mantissa has an implied 1 immediately to the left of the binary point. The following table shows the different types of number representation in IEEE 754-2019 single precision format. In this table:

- s = bit [31]: sign bit
- e = bits [30:23]: biased exponent
- f = bits [22:0]: fractional part of mantissa

Table 15 **IEEE 754-2019 single precision representation types**

Condition	Represented value	Description
0 < e < 255	$(-1)^s \times 2^{(e-127)} \times 1.f$	Normal number
e == 0 AND f != 0	$(-1)^s \times 2^{(-126)} \times 0.f$	Subnormal number
e == 0 AND f == 0	$(-1)^s \times 0$	Signed zero
s == 0 AND e == 255 AND f == 0	$-\infty$	+ infinity
s == 1 AND e == 255 AND f == 0	$-\infty$	– infinity
e == 255 AND f != 0 AND f[22] == 0		Signaling NaN[1]
e == 255 AND f != 0 AND f[22] == 1		Quiet NaN[1]

1) IEEE 754-2019 defines bit[22] as the way to distinguish between signaling NaNs and quiet NaNs

***Note**:* *Both signed values of zero are always treated identically and never produce different results except different signed zeros.*

9.2.2.2 IEEE 754-2019 single precision NaNs

NaNs (Not a Number) are bit combinations within the IEEE 754-2019 standard that do not correspond to numbers. There are two types of NaNs: signaling and quiet. The single precision FPU defines signaling NaNs to have bit 22 = ‘0’, and quiet NaNs to have bit 22 = ‘1’.

When invalid operations are performed (including operations with a signaling NaN operand), FI is asserted and a quiet NaN is produced as the floating-point result. The quiet NaN contains information about the origin of the invalid operation; see Table 20.

IEEE 754-2019 suggests that quiet NaNs should be propagated so that the result of an instruction receiving a quiet NaN as an operand (with no signaling NaN operands) should be that quiet NaN. The FPU does not propagate quiet NaNs in this way. The result of an operation that has one (or more) quiet NaN operands and no signaling NaN operands is always the quiet NaN $7FC00000_H$.

9.2.2.3 Subnormal numbers

Subnormal numbers are not supported for single precision arithmetic operations. With the exception of the ABS.F, CMP.F, MAX.F, MIN.F, NEG.F, HPTOF and FTOHP instructions, single precision instructions replace subnormal operands with the appropriately signed zero before computation. Following computation, if a subnormal number would otherwise be the result, it is replaced with the appropriately signed zero.

Conceptually, the conventional order for making IEEE 754-2019 computations is:

1. Compute result to infinite precision
2. Round to IEEE 754-2019 format

This is replaced with:

1. Substitute signed zero for all subnormal operands
2. Compute result to infinite precision
3. Round to IEEE 754-2019 format
4. Substitute signed zero for all subnormal results

This procedure has a subtle effect on underflow; see Round to nearest: subnormals and zero substitution.

Subnormal numbers are supported by the CMP.F instruction which makes comparisons of subnormal numbers in addition to identifying subnormal operands.

Subnormal numbers are also supported by the ABS.F, MAX.F, MIN.F, NEG.F, HPTOF and FTOHP instructions.

9.2.2.4 Single precision underflow

Underflow occurs when the result of a single precision floating-point operation is too small to store in floating-point representation.

IEEE 754-2019 requires two conditions to occur before flagging underflow:

- The result must be 'tiny'
  - A result is 'tiny' if it is non-zero and its magnitude is < 2^{-126} (for single precision). IEEE 754-2019 allows this to be detected either before or after rounding
- AND there must be a loss of accuracy in the stored result

Loss of accuracy can be detected in two ways: either as a subnormalization loss, or an inexact result.

Subnormalization loss occurs when the result is calculated assuming an unbounded exponent, but is rounded to a normalized number using 23 fractional bits. If this rounded result must be subnormalized to fit into IEEE 754-2019 format and the resultant subnormalized number differs from the normalized result with unbounded exponent range, then a subnormalization loss occurs.

An inexact result is one where the infinitely precise result differs from the value stored.

The FPU determines tininess before rounding and inexact results to determine loss of accuracy.

In the case of the FPU, even if a subnormal result would produce no loss of accuracy, because it is replaced with a zero, accuracy is lost and underflow must be flagged.

Any tiny number that is detected must therefore result in a loss of accuracy since it will either be a subnormal that is replaced with zero or rounded up. Therefore underflow detection can be simplified to tiny number detection alone; that means any non-zero unrounded number whose magnitude is < 2^{-126}.

9.2.3 IEEE 754-2019 double precision

9.2.3.1 IEEE 754-2019 double precision data format

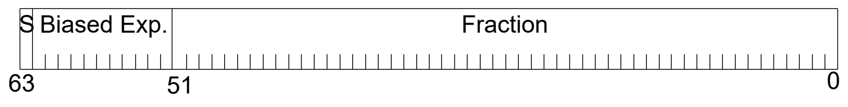


Figure 27 Double precision IEEE 754-2019 floating-point format

The double precision IEEE 754-2019 floating-point format has three sections: a sign bit, an 11-bit biased exponent, and a 52-bit fractional mantissa with an implied binary point before bit 51. For normal numbers the mantissa has an implied 1 immediately to the left of the binary point. The following table shows the different types of number representation in IEEE 754-2019 double precision format. In this table:

- s = bit [63]: sign bit
- e = bits [62:52]: biased exponent
- f = bits [51:0]: fractional part of mantissa

Table 16 IEEE 754-2019 double precision representation types

Condition	Represented value	Description
0 < e < 2047	$(-1)^s \times 2^{(e-1023)} \times 1.f$	Normal number
e == 0 AND f != 0	$(-1)^s \times 2^{(-1022)} \times 0.f$	Subnormal number

(table continues...)

Table 16 (continued) IEEE 754-2019 double precision representation types

Condition	Represented value	Description
e == 0 AND f == 0	$(-1)^s \times 0$	Signed zero
s == 0 AND e == 2047 AND f == 0	$-\infty$	+ infinity
s == 1 AND e == 2047 AND f == 0	$-\infty$	– infinity
e == 2047 AND f != 0 AND f[51] == 0		Signaling NaN[1)]
e == 2047 AND f != 0 AND f[51] == 1		Quiet NaN[1)]

1) IEEE 754-2019 defines bit[51] as the way to distinguish between signaling NaNs and quiet NaNs

Note: *Both signed values of zero are always treated identically and never produce different results except different signed zeros.*

9.2.3.2 IEEE 754-2019 double precision NaNs

NaNs (Not a Number) are bit combinations within the IEEE 754-2019 standard that do not correspond to numbers. There are two types of NaNs: signaling and quiet. The double precision FPU defines signaling NaNs to have bit 51 = ‘0’, and quiet NaNs to have bit 51 = ‘1’.

When invalid double precision operations are performed (including operations with a signaling NaN operand), FI is asserted and a quiet NaN is produced as the floating-point result. The quiet NaN contains information about the origin of the invalid operation; see Table 20.

IEEE 754-2019 suggests that quiet NaNs should be propagated so that the result of an instruction receiving a quiet NaN as an operand (with no signaling NaN operands) should be that quiet NaN. The FPU does not propagate quiet NaNs in this way. The result of an operation that has one (or more) quiet NaN operands and no signaling NaN operands is always the quiet NaN $7FF8000000000000_H$.

9.2.3.3 Double precision subnormal numbers

Double precision subnormal numbers are fully supported.

9.2.3.4 Double precision underflow

Underflow occurs when the result of a double precision floating-point operation is too small to store in floating-point representation.

IEEE 754-2019 requires two conditions to occur before flagging underflow:

- The result must be ‘tiny’
  - A result is ‘tiny’ if it is non-zero and its magnitude is < 2^{-1022} (for double precision). IEEE 754-2019 allows this to be detected either before or after rounding
- AND there must be a loss of accuracy in the stored result

Loss of accuracy can be detected in two ways: either as a subnormalization loss, or an inexact result.

Subnormalization loss occurs when the result is calculated assuming an unbounded exponent, but is rounded to a normalized number using 52 fractional bits. If this rounded result must be subnormalized to fit into IEEE 754-2019 format and the resultant subnormalized number differs from the normalized result with unbounded exponent range, then a subnormalization loss occurs.

An inexact result is one where the infinitely precise result differs from the value stored.

The FPU determines tininess before rounding and inexact results to determine loss of accuracy.

In the case of the FPU, when a subnormal result produces a loss of accuracy, the underflow must be flagged.

Any tiny number that results in a loss of accuracy results in the underflow flag.

9.2.4 Traps

IEEE 754-2019 allows optional provision for synchronous traps to occur when exception conditions occur. Under these circumstances the results returned by arithmetic operations may differ from IEEE 754-2019 requirements to allow intermediate results to be passed to the trap handling routines. These traps are provided to assist in debugging routines and operations.

FPU traps generated by input operand conditions are synchronous (CSE) while FPU traps generated by results are asynchronous (CAE) and therefore not IEEE 754-2019 compliant traps. Since IEEE 754-2019 traps are optional this does not cause any IEEE 754-2019 non compliance.

9.2.5 Software routines

Operations required for IEEE 754-2019 compliance, but not implemented in the FPU instruction set, are detailed in the following table.

Table 17 IEEE 754-2019 operations requiring software implementation

IEEE 754-2019 operation	Suggested implementation
Square root	Newton-Raphson using QSEED.F instruction for single precision or QSEED.DF for double precision results
Remainder	FPU instructions cannot be used to implement the remainder function because of the errors that can occur from multiple rounding. For reference, the IEEE method for calculating remainder is given below. Note that rounding must only occur on the conversion to integer, and for the final result. **Operands** rem: remainder x: dividend d: divisor **Single precision** rem = x - (d * (FTOIN(x/d) [1])) **Double precision** rem = x - (d * (DFTOL(x/d)[1]))
Round to integer in single precision floating-point format	ITOF(FTOI(x))
Round to long integer in double precision floating-point format	LTODF(DFTOL(x))
Conversion between binary and decimal	-
Conversion from long or unsigned long to single precision floating-point format	- ***Note**: The sequence long or unsigned long to double precision followed by double precision to single precision leads to multiple rounding which would not be IEEE compliant.*

1) Round to nearest.

9.3 Rounding

All four rounding modes specified in IEEE 754-2019 are supported. The rounding mode is selected using the RM field of the PSW (PSW[25:24]).

Table 18 Rounding mode definition (PSW.RM)

Rounding mode value	Mode
00 [1)]	Round to nearest
01	Round toward + ∞
10	Round toward - ∞
11	Round toward zero

1) Round to nearest is the default rounding mode.

IEEE 754-2019 defines the rounding modes in terms of representable results, in relation to the 'infinitely precise' result. The infinitely precise result is the mathematically exact result that would be computed by the operation, if the number of mantissa and exponent bits were unlimited.

- **Round to nearest with ties towards even** is defined as returning the representable value that is nearest to the infinitely precise result. This is the default rounding mode that should be selected when RTOS software initializes a task. See Round to nearest with ties towards even, for further information
- **Round toward + ∞** is defined as returning the representable value that is closest to and no less than the infinitely precise result
- **Round toward – ∞** is defined as returning the representable value that is closest to and no greater than the infinitely precise result
- **Round toward zero** is defined as returning the representable value that is closest to and no greater in magnitude than the infinitely precise result. It is equivalent to truncation

The rounding mode can be changed by the UPDFL (Update Flags) instruction.

Rounding is performed at the end of each relevant FPU instruction, followed by the replacement of all subnormal single precision results with the appropriately signed 0.

The result from the multiply part of a MAC instruction is not rounded before it is used in the addition in the FPU. Instead the whole MAC is calculated with infinite precision and rounded at the end of the add.

- The result from a MADD.F/MSUB.F instruction may differ from the result that would be obtained using the same operands in a MUL.F followed by an ADD.F/SUB.F
- The result from a MADD.DF/MSUB.DF instruction may differ from the result that would be obtained using the same operands in a MUL.DF followed by an ADD.DF/SUB.DF

Rounding mode restored

The rounding mode is restored on an RFE (return from exception) instruction or an RFM (return from monitor) instruction. The RET (Return From Call) instruction only restores the rounding mode when COMPAT.RM == 1.

- COMPAT.RM == 0 - PSW.RM not restored by RET
- COMPAT.RM == 1 - PSW.RM restored by RET (TC1.3 behavior)

9.3.1 Round to nearest with ties towards even

'Round to nearest' is defined as returning the representable value that is nearest to the infinitely precise result. If two representable values are equally close (that means the infinitely precise result is exactly half way between two representable values), then the one whose LSB (Least Significant Bit) is zero is returned. This is sometimes known as rounding to nearest even.

This is usually straight forward, but if the infinitely precise result is half way between two representable numbers with different exponents, the result with the larger exponent is always selected (the LSB of its mantissa is zero).

For example, if the infinitely precise result is:

$1.111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1000\ 0000\ 0000_B * 2^0$

This is half way between:

$1.0000\ 0000\ 0000\ 0000\ 0000\ 000_B * 2^1$

and:

$1.111\ 1111\ 1111\ 1111\ 1111\ 1111_B * 2^0$

The result with the larger exponent is returned.

9.3.2 Round to nearest: subnormals and zero substitution

Following computation, single precision results are first rounded to IEEE 754-2019 representable numbers and then the appropriately signed zero is substituted for any subnormal results that may have occurred. This produces some results that can seem counter intuitive.

Consider an infinitely precise result that has been computed and falls between the smallest representable positive IEEE 754-2019 normal number ($1.000 \ldots 000 * 2^{-126}$) and the largest representable positive IEEE 754-2019 subnormal number ($0.111 \ldots 111 * 2^{-126}$).

- If the infinitely precise result is nearer to the normal number, or halfway between the two, then the result must be rounded to the normal number
- If the infinitely precise result is nearer to the subnormal number, then the result is rounded to the subnormal value. Zero is then substituted for the subnormal result

The single precision FPU architecture does not produce subnormal results, however the concept of subnormal numbers is important to the FPU. It would be wrong to assume that the infinitely precise result should be rounded to the nearest FPU representable number, in this case ($+1.000 \ldots 000 * 2^{-126}$) or (0). Such an implementation would mean that all unrounded results between ($+1.000 \ldots 000 * 2^{-126}$) and ($+0.100 \ldots 000 * 2^{-126}$) would be rounded to the smallest representable positive IEEE 754-2019 normal number.

The double precision FPU architecture fully supports subnormal results.

9.3.3 Round Towards ± ∞: subnormals and zero substitution

Following computation single precision results are first rounded to IEEE 754-2019 representable numbers, then the appropriately signed zero is substituted for any subnormal results that may have occurred. See Subnormal numbers.

According to the IEEE 754-2019 definition of the rounding modes, when rounding towards +∞ (- ∞) the rounded result should not be less than (greater than) the infinitely precise result. However if a positive (negative) result would otherwise be rounded to a subnormal number, it is then substituted for a zero. Therefore the returned result of zero is less than (greater than) the infinitely precise result. The returned result appears to contradict the definition of these rounding modes in this case.

The double precision FPU architecture fully supports subnormal results.

9.4 Exceptions

The FPU implements all five IEEE 754-2019 exceptions (invalid operation, overflow, divide by zero, underflow, and inexact). When one of these exceptions occur the corresponding exception flag in the PSW is asserted.

Asynchronous traps

An asynchronous trap may optionally be taken when an overflow, underflow or inexact exception occurs. These traps might not be suitable for the purpose of resuming alternate exception handling, although software can always implement them through other means, however IEEE 754-2019 compliant traps are not implemented, see Asynchronous and synchronous traps.

Synchronous traps

A synchronous trap may optionally be taken when an invalid operation or a division by zero exception occurs. These implement traps which can support resuming alternate exception handling, however IEEE 754-2019 compliant traps are not implemented, see Asynchronous and synchronous traps.

IEEE 754-2019 exception flags

The IEEE 754-2019 exception flags are stored as part of the PSW register as shown in the following table. In accordance with IEEE 754-2019, each bit is sticky so that the FPU instructions in general assert these flags when an exception occurs and do not negate them when the exception does not occur. The UPDFL instruction can be used to clear the exception flags.

Table 19 FPU exception flags

ALU flag	FPU flag	FPU exception	PSW bit position
C	FS	Some exception	31
V	FI	Invalid operation	30
SV	FV	Overflow	29
AV	FZ	Divide by zero	28
SAV	FU	Underflow	27
-	FX	Inexact	26

Since the IEEE 754-2019 exception flags are sticky, it can be impossible to tell if an asynchronous exception occurred on the last instruction if it was asserted before the last instruction executed. An additional, non sticky, exception flag (FS) is therefore implemented to identify if the last FPU instruction caused an IEEE 754-2019 exception or not.

Note that the PSW bits used to store the exception flags are also used to store ALU flags as shown in the table above. When an ALU instruction updates these flags, the corresponding FPU exception flag is overwritten and lost.

The following conditions are true for all FPU operations asserting exception flags, with the exception of UPDFL.

- Any FPU operation can assert only one of the FI, FV, FZ or FU exception flags
- FX can be asserted by any operation so long as FI and FZ are negated
- When either FV or FU are asserted, FX is also asserted

FS - some exception

This bit is not sticky and is asserted or negated for all instructions that can cause IEEE 754-2019 exceptions to occur. If any of the IEEE 754-2019 exceptions (FI, FV, FZ, FU, FX) have occurred during that instruction, FS is also asserted.

***Note**: UPDFL can assert IEEE 754-2019 exceptions without asserting FS.*

FI - invalid operation

FI is asserted in four circumstances:

- When a signaling NaN (see IEEE 754-2019 single precision NaNs) is an operand for a single precision FPU instruction
- When a signaling NaN (see IEEE 754-2019 double precision NaNs) is an operand for a double precision FPU instruction
- For invalid operations such as QSEED.F or QSEED.DF of a negative number
- Conversions from floating-point to other floating-point formats where the rounded result is outside the range of the target

When an instruction that produces a floating-point result asserts FI as a result of a signaling NaN or invalid operation, the result is a quiet NaN.

The exception associated with invalid operation is synchronous (CSE).

Table 20 Invalid operations and their quiet NaN results (single-precision)

Invalid operation	Quiet NaN
Signaling NaN operand for arithmetic instructions [1)]	$7FC00000_H$ [2)]
Signaling NaN operand for CMP.F instruction	n.a.
ADD.F with + ∞ and - ∞ as operands	$7FC00001_H$
SUB.F with (+ ∞ and + ∞) or (- ∞ and - ∞) as operands	$7FC00001_H$
MADD.F if the result of the multiplication is ± ∞ and the addend is the oppositely signed ∞	$7FC00001_H$
MSUB.F if the result of the multiplication is ± ∞ and the minuend is the same signed ∞	$7FC00001_H$
MUL.F with 0 and ± ∞ as multiplicands	$7FC00002_H$
MADD.F with 0 and ± ∞ as multiplicands	$7FC00002_H$
MSUB.F with 0 and ± ∞ as multiplicands	$7FC00002_H$
QSEED.F with a negative operand [3)]	$7FC00004_H$
DIV.F with 0 as both operands [4)]	$7FC00008_H$
DIV.F with both operands being an ∞ of either sign	$7FC00008_H$
FTOI, FTOU or FTOQ31 with rounded result outside the range of the target format	n.a. [5)]
FTOIN, FTOIZ, FTOUZ or FTOQ31Z with rounded result outside the range of the target format	n.a.[5)]
FTOI, FTOU or FTOQ31 with the input operand a quiet NaN, a signalling NaN or ± ∞	n.a.[5)]
FTOIN, FTOIZ, FTOUZ or FTOQ31Z with the input operand a quiet NaN, a signalling NaN or ± ∞	n.a.[5)]

1) Also see the FPU operation syntax description in the Instruction Set.
2) The quiet NaN ($7FC00000_H$) is produced as the result of arithmetic operations that have any NaN as an operand. FI is only asserted when one of these NaNs is signaling. See IEEE 754-2019 single precision NaNs.
3) -0 is not negative, therefore QSEED.F of -0 is -∞
4) 0/0 is defined as being an invalid operation (FI) rather than a divide by zero (FZ).
5) The result is not in floating-point format and therefore cannot be a quiet NaN. Refer to the instruction description for what the result should be.

Table 21 Invalid operations and their quiet NaN results (double-precision)

Invalid operation	Quiet NaN
Signaling NaN operand for arithmetic instructions [1)]	$7FF8000000000000_H$ [2)]
Signaling NaN operand for CMP.DF instruction	n.a.

(table continues...)

Table 21 (continued) Invalid operations and their quiet NaN results (double-precision)

Invalid operation	Quiet NaN
ADD.DF with + ∞ and - ∞ as operands	$7FF8000000000001_H$
SUB.DF with (+ ∞ and + ∞) or (- ∞ and - ∞) as operands	$7FF8000000000001_H$
MADD.DF if the result of the multiplication is ± ∞ and the addend is the oppositely signed ∞	$7FF8000000000001_H$
MSUB.DF if the result of the multiplication is ± ∞ and the minuend is the same signed ∞	$7FF8000000000001_H$
MUL.DF with 0 and ± ∞ as multiplicands	$7FF8000000000002_H$
MADD.DF with 0 and ± ∞ as multiplicands	$7FF8000000000002_H$
MSUB.DF with 0 and ± ∞ as multiplicands	$7FF8000000000002_H$
QSEED.DF with a negative operand [3]	$7FF8000000000004_H$
DIV.DF with 0 as both operands [4]	$7FF8000000000008_H$
DIV.DF with both operands being an ∞ of either sign	$7FF8000000000008_H$
DFTOI, DFTOU, DFTOL or DFTOUL with rounded result outside the range of the target format	n.a. [5]
DFTOIN, DFTOIZ, DFTOUZ, DFTOLZ or DFTOULZ with rounded result outside the range of the target format	n.a.[5]
DFTOI, DFTOU, DFTOL or DFTOUL with the input operand a quiet NaN, a signaling NaN or ± ∞	n.a.[5]
DFTOIN, DFTOIZ, DFTOUZ, DFTOLZ or DFTOULZ with the input operand a quiet NaN, a signaling NaN or ± ∞	n.a.[5]

1) Also see the FPU operation syntax description in the instruction set.
2) The quiet NaN ($7FF8000000000000_H$) is produced as the result of arithmetic operations that have any NaN as an operand. FI is only asserted when one of these NaNs is signaling. See IEEE 754-2019 double precision NaNs.
3) -0 is not negative, therefore QSEED.F of -0 is -∞.
4) 0/0 is defined as being an invalid operation (FI) rather than a divide by zero (FZ).
5) The result is not in floating-point format and therefore cannot be a quiet NaN. Refer to the instruction description for what the result should be.

FV - overflow

For operations that return a floating-point result, the FV flag is set as stated in IEEE 754-2019; 'whenever the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result, were the exponent range unbounded'.

The result returned is determined by the rounding mode and the sign of the unrounded result:

- Round to nearest carries all overflows to infinity, with the sign of the unrounded result
- Round toward zero carries all overflows to the format's largest finite number with the sign of the unrounded result

- Round toward minus infinity carries positive overflows to the format's largest finite number, and carries negative overflows to minus infinity
- Round toward plus infinity carries negative overflows to the format's most negative finite number, and carries positive overflows to plus infinity

When overflow is flagged (FV asserted), the returned result can not be exactly equal to the unrounded result. Therefore whenever FV is asserted FX is also asserted.

The exception associated with overflow is asynchronous (CAE).

FZ - divide by zero

The FZ flag is set by DIV.F if the divisor operand is zero and the dividend operand is a finite non zero number. The result is an infinity with sign determined by the usual rules.

Note that:

- 0/0 is defined as an invalid operation, so FI is asserted rather than FZ
- All arithmetic with $\pm\infty$ as an operand is defined as being exact, except for invalid operations where FI is asserted. Therefore for $\pm\infty/\pm 0$ FZ is not asserted, the appropriately signed ∞ is returned as the result with no other exceptions occurring

The exception associated with divide by zero is synchronous (CSE).

FU - underflow

As discussed in Single precision underflow and Double precision underflow , underflow is detected and so FU is asserted, when the unrounded result is smaller in magnitude than the smallest representable normal number.

The instructions that can cause underflow are:

- Q31TOF
- ADD.F, DIV.F, MADD.F, MSUB.F, MUL.F, and SUB.F
- ADD.DF, DIV.DF, MADD.DF, MSUB.DF, MUL.DF, and SUB.DF

The return result for instructions flagging an underflow are complicated by the way that FPU treats subnormal numbers. This is described in detail in Subnormal numbers.

The exception associated with underflow is asynchronous (CAE).

FX - inexact

If the rounded result of an operation is not exactly equal to the unrounded result, then the FX flag is set.

The result delivered is the rounded result, unless either overflow (FV) or underflow (FU) has also occurred during this instruction, when the overflow or subnormalization return result rules are followed.

The exception associated with overflow is asynchronous (CAE).

Related information

PSW user status bits on page 35

9.5 Asynchronous and synchronous traps

The FPU can be configured such that a trap is signaled to the TriCore™ core when an FPU instruction causes an IEEE 754-2019 exception. The trap generated is either coprocessor Asynchronous Error (CAE), trap class 4, TIN 4 or a Co-Processor Synchronous Error (CSE) trap class 2 TIN 6.

Asynchronous trap

The FPU CAE trap is intended for monitoring purposes only and has no effect on either the exceptional instruction or any other instruction which may be executing within the FPU. The result returned by an exceptional instruction causing a CAE trap is identical to that which would be returned if no trap were taken. The CAE trap is signaled after instruction completion.

The specific exception conditions which cause FPU CAE traps to be generated are under software control. To enable the trap generation for a specific exception type the appropriate enable bit in the FPU_TRAP_CON register must be asserted (FVE, FUE or FXE). Any number of these enable bits may be set to allow traps to be taken if any of a range of exceptions occur. FX is a regularly occurring condition, care should be taken in enabling this trap.

When an instruction causes one of the enabled exceptions, information about the exceptional instruction including the instruction PC, opcode and source operands are captured in the FPU special function registers. At the same time the Trap Status flag (TST) is set within the FPU_TRAP_CON register, denoting that the contents of the FPU trap capture registers are valid. In addition, so long as FPU_TRAP_CON.TST remains set, further FPU CAE trap generation is inhibited. This avoids multiple traps being generated from the same root problem and the original information being lost. Once the trap handler has interrogated the FPU to determine the cause of the trap, the FPU_TRAP_CON.TST bit may be cleared to enable further traps.

The result of the exceptional instruction causing a trap is not stored in an FPU CSFR. The result will be available in the instruction’s destination register as long as it has not been overwritten before the asynchronous trap is taken.

Synchronous trap

The FPU CSE trap is intended to allow more straightforward support of 'resuming alternate exception handlers'. The CSE trap is signaled after instruction completion.

The specific exception conditions which cause FPU CSE traps to be generated are under software control. To enable the trap generation for a specific exception type the appropriate enable bit in the FPU_SYNC_TRAP_CON register must be asserted (FIE or FZE). Any number of these enable bits may be set to allow traps to be taken if any of a range of exceptions occur.

When an instruction causes one of the enabled exceptions, information about the exceptional instruction, including the opcode, is captured in the FPU special function registers. The trap being synchronous, the PC of the trapping instruction is available in A[11](RA). At the same time the Trap Status flag (TST) is set within the FPU_SYNC_TRAP_CON register, denoting that the contents of the FPU trap capture registers are valid. In addition, so long as FPU_SYNC_TRAP_CON.TST remains set, further FPU CSE trap generation is inhibited. This avoids multiple traps being generated from the same root problem and the original information being lost. Once the trap handler has interrogated the FPU to determine the cause of the trap, the FPU_SYNC_TRAP_CON.TST bit may be cleared to enable further traps.

The IEEE 754-2019 defined result of the exceptional instruction causing a CSE trap is not stored in an FPU CSFR. As the trap is synchronous the instruction will not modify the GPRs or PSW state, only the FPU CSFRs.

9.6 FPU CSFR Registers

The FPU CSFR registers are used to store the details of instructions causing traps.

9.6.1 FPU trap control register

FPU_TRAP_CON	Address:	A000$_H$
Trap control register	Reset value:	0000 0000$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES		**FV**	**RES**	**FU**	**FX**	**RES**				**FVE**	**RES**	**FUE**	**FXE**	**RES**	
-		rh	-	rh	rh	-				rw	-	rw	rw	-	

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES						**RM**		**RES**						**TCL**	**TST**
-						rh		-						w	rh

Field	Bits	Type	Description
RES	31:30	-	**Reserved**
FV	29	rh	Captured FV Asserted if the captured instruction asserted FV. Only valid when TST is asserted.
RES	28	-	Reserved
FU	27	rh	Captured FU Asserted if the captured instruction asserted FU. Only valid when TST is asserted.
FX	26	rh	Captured FX Asserted if the captured instruction asserted FX. Only valid when TST is asserted.
RES	25:22	-	**Reserved**
FVE	21	rw	FV trap enable When set, an instruction generating an FV exception will trigger a trap.
RES	20	-	Reserved
FUE	19	rw	FU trap enable When set, an instruction generating an FU exception will trigger a trap.
FXE	18	rw	FX trap enable When set, an instruction generating an FX exception will trigger a trap.
RES	17:10	-	**Reserved**

(table continues...)

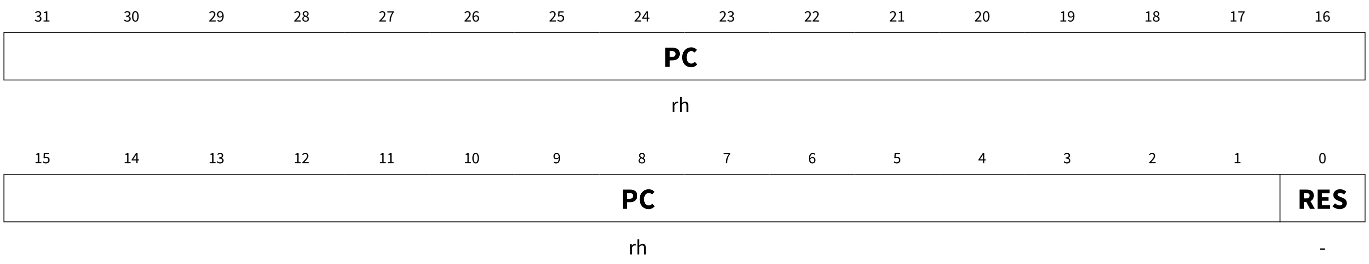
(continued)

Field	Bits	Type	Description
RM	9:8	rh	Captured rounding mode The rounding mode of the captured instruction. Only valid when TST is asserted. Note that this is the rounding mode supplied to the FPU for the exceptional instruction. UPDFL instructions may cause a trap and change the rounding mode. In this case the RM bits capture the input rounding mode.
RES	7:2	-	**Reserved**
TCL	1	w	Trap clear 1 : Clears the trapped instruction (TST will be negated). 0 : Does nothing. Read: always reads as 0.
TST	0	rh	Trap status 0 : No instruction captured: The next enabled exception will cause the exceptional instruction to be captured. 1 : Instruction captured: No further enabled exceptions will be captured until TST is cleared.

9.6.2 FPU trapping instruction program counter register

FPU_TRAP_PC Address: $A004_H$

Trapping instruction program counter Reset value: Implementation Specific



Field	Bits	Type	Description
PC	31:1	rh	Captured program counter The program counter of the captured instruction. Only valid when FPU_TRAP_CON.TST is asserted.
RES	0	-	Reserved

9.6.3 FPU trapping instruction opcode register

FPU_TRAP_OPC Address: $A008_H$

Trapping instruction opcode Reset value: Implementation Specific

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES												**DREG**			
-												rh			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES						**DP**	**FMT**	**OPC**							
-						rh	rh	rh							

Field	Bits	Type	Description
RES	31:20	-	**Reserved**
DREG	19:16	rh	Captured destination register The destination register of the captured instruction. 0_H : Data general purpose register 0. … F_H : Data general purpose register 15. Only valid when FPU_TRAP_CON.TST is asserted.
RES	15:10	-	**Reserved**
DP	9	rh	Captured instruction precision Single / Double precision identifier. Only valid when FPU_TRAP_CON.TST is asserted. 0_B : Single Precisions (including HPTOF and FTOHP) 1_B : Double Precision
FMT	8	rh	Captured instruction format The format of the captured instruction’s opcode. Only valid when FPU_TRAP_CON.TST is asserted. 0_B : RRR. 1_B : RR.
OPC	7:0	rh	Captured opcode The secondary opcode of the captured instruction. When FMT=0 only bits [3:0] are defined. OPC is valid only when FPU_TRAP_CON.TST is asserted.

9.6.4 FPU trapping instruction operand SRC1_L register

FPU_TRAP_SRC1_L		Address:	$A010_H$
Trapping instruction operand		Reset value:	Implementation Specific

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SRC1_L															
rh															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRC1_L															
rh															

Field	Bits	Type	Description
SRC1_L	31:0	rh	Captured SRC1 operand The SRC1[31:0] operand of the captured instruction. Only valid when FPU_TRAP_CON.TST is asserted.

9.6.5 FPU trapping instruction operand SRC1_U register

FPU_TRAP_OPC_SRC1_U		Address:	$A014_H$
Trapping instruction operand		Reset value:	Implementation $Specific_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SRC1_U															
rh															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRC1_U															
rh															

Field	Bits	Type	Description
SRC1_U	31:0	rh	Captured SRC1 operand The SRC1[63:32] operand of the captured instruction. For single precision SRC1, this register is set to 0. Only valid when FPU_TRAP_CON.TST is asserted.

9.6.6 FPU trapping instruction operand SRC2_L register

FPU_TRAP_SRC2_L		Address:	$A018_H$
Trapping instruction operand		Reset value:	Implementation Specific

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SRC2															
rh															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRC2															
rh															

Field	Bits	Type	Description
SRC2	31:0	rh	Captured SRC2 operand The SRC2[31:0] operand of the captured instruction. Only valid when FPU_TRAP_CON.TST is asserted.

9.6.7 FPU trapping instruction operand SRC2_U register

FPU_TRAP_OPC_SRC2_U Address: A01C$_H$
Trapping instruction operand Reset value: Implementation Specific$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SRC2_U															
rh															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRC2_U															
rh															

Field	Bits	Type	Description
SRC2_U	31:0	rh	Captured SRC2 operand The SRC2[63:32] operand of the captured instruction. For single precision SRC2, this register is set to 0. Only valid when FPU_TRAP_CON.TST is asserted.

9.6.8 FPU trapping instruction operand SRC3_L register

FPU_TRAP_SRC3_L Address: A020$_H$
Trapping instruction operand Reset value: Implementation Specific

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SRC3_L															
rh															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRC3_L															
rh															

Field	Bits	Type	Description
SRC3_L	31:0	rh	Captured SRC3 operand The SRC3[31:0] operand of the captured instruction. Only valid when FPU_TRAP_CON.TST is asserted.

9.6.9 FPU trapping instruction operand SRC3_U register

FPU_TRAP_OPC_SRC3_U Address: A024$_H$
Trapping instruction operand Reset value: Implementation Specific$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SRC3_U															
rh															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRC3_U															
rh															

Field	**Bits**	**Type**	**Description**
SRC3_U	31:0	rh	Captured SRC3 operand The SRC3[63:32] operand of the captured instruction. For single precision SRC3, this register is set to 0. Only valid when FPU_TRAP_CON.TST is asserted.

9.6.10 FPU synchronous trap control register

FPU_SYNC_TRAP_CON Address: $A030_H$

Synchronous trap control register Reset value: $0000\ 0000_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES	**FI**	**RES**	**FZ**	**RES**					**FIE**	**RES**	**FZE**	**RES**			
-	rh	-	rh	-					rw	-	rw	-			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES						**RM**		**RES**						**TCL**	**TST**
-						rh		-						w	rh

Field	**Bits**	**Type**	**Description**
RES	31	-	**Reserved**
FI	30	rh	Captured FI Asserted if the captured instruction asserted FI. Only valid when TST is asserted.
RES	29	-	Reserved
FZ	28	rh	Captured FZ Asserted if the captured instruction asserted FZ. Only valid when TST is asserted.
RES	27:23	-	**Reserved**
FIE	22	rw	FI trap enable When set, an instruction generating an FI exception will trigger a trap.
RES	21	-	Reserved
FZE	20	rw	FZ trap enable When set, an instruction generating an FZ exception will trigger a trap.

(table continues...)

(continued)

Field	Bits	Type	Description
RES	19:10	-	**Reserved**
RM	9:8	rh	Captured rounding mode The rounding mode of the captured instruction. Only valid when TST is asserted. Note that this is the rounding mode supplied to the FPU for the exceptional instruction. UPDFL instructions may cause a trap and change the rounding mode. In this case the RM bits capture the input rounding mode.
RES	7:2	-	**Reserved**
TCL	1	w	Trap clear 1 : Clears the trapped instruction (TST will be negated). 0 : Does nothing. Read: always reads as 0.
TST	0	rh	Trap status 0 : No instruction captured: The next enabled exception will cause the exceptional instruction to be captured. 1 : Instruction captured: No further enabled exceptions will be captured until TST is cleared.

9.6.11 FPU synchronous trapping instruction opcode register

FPU_SYNC_TRAP_OPC	Address:	$A034_H$
Synchronous trapping instruction opcode	Reset value:	Implementation Specific

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES												**DREG**			
-												rh			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES						**DP**	**FMT**	**OPC**							
-						rh	rh	rh							

Field	Bits	Type	Description
RES	31:20	-	**Reserved**
DREG	19:16	rh	Captured destination register The destination register of the captured instruction. 0_H : Data general purpose register 0. … F_H : Data general purpose register 15. Only valid when FPU_SYNC_TRAP_CON.TST is asserted.
RES	15:10	-	**Reserved**

(table continues...)

(continued)

Field	Bits	Type	Description
DP	9	rh	Captured instruction precision Single / Double precision indentifier. Only valid when FPU_SYNC_TRAP_CON.TST is asserted. 0_B : Single Precisions (including HPTOF and FTOHP) 1_B : Double Precision
FMT	8	rh	Captured instruction format The format of the captured instruction's opcode. Only valid when FPU_SYNC_TRAP_CON.TST is asserted. 0_B : RRR. 1_B : RR.
OPC	7:0	rh	Captured opcode The secondary opcode of the captured instruction. When FMT=0 only bits [3:0] are defined. OPC is valid only when FPU_SYNC_TRAP_CON.TST is asserted.

9.6.12 PSW user status bits (FPU view)

The eight most significant bits of the PSW are designated as user status bits. These bits may be set or cleared as execution side effects of user instructions, typically recording result status. Individual bits can also be used to condition the operation of float-point operations.

Table 22 PSW FPU usage of user status bits

ALU flag	Field	Bits	Type	Description
C	FS	31	rw	Some exception
V	FI	30	rw	Invalid operation
SV	FV	29	rw	Overflow
AV	FZ	28	rw	Divide by zero
SAV	FU	27	rw	Underflow
-	FX	26	rw	Inexact
-	RM	25:24	rw	Rounding mode

Related information

PSW user status bits on page 35

10 Memory protection system

The TriCore™ protection system provides the essential features to isolate errors. The system is unobtrusive, imposing little overhead and avoids non-deterministic run-time behavior.

The protection system incorporates hardware mechanisms that protect software-specified memory ranges from unauthorized read, write, or instruction fetch accesses.

The protection hardware can also facilitate application debugging.

Related information

HRHV memory protection unit on page 183

10.1 Memory protection subsystems

The following subsystems are involved with memory protection.

The trap system

A trap occurs as a result of an event such as a Non-Maskable Interrupt (NMI), an instruction exception or illegal access.

The TriCore™ architecture contains eight trap classes and these are further classified as synchronous or asynchronous, hardware or software.

For more information see Trap system.

The I/O privilege level

There are three I/O modes: User-0 mode, User-1 mode and Supervisor mode.

The User-1 mode allows application tasks to directly access non-critical system peripherals. This allows systems to be implemented efficiently, without the loss of security inherent in running in Supervisor mode. (The default behavior of User-1 mode may be overriden by the core control register).

For more information see Access privilege level control (I/O privilege).

Memory protection

Provides control over which regions of memory a task is allowed to access, and what types of access is permitted.

- Range based: The range-based memory protection system is designed for small and low cost applications to provide coarse-grained memory protection for systems that do not require virtual memory. This range-based system is detailed in this chapter
- Page based: For applications that require virtual memory, the optional Memory Management Unit (MMU) supports a familiar model that gives each memory page its own access permissions

Effective addresses

Effective addresses are translated into physical addresses using one of two translation mechanisms:

- Direct translation
- Page Table Entry (PTE) based translation (optional MMU only)

Memory protection for addresses that undergo direct address translation is enforced using the range-based memory protection system described in this chapter.

10.2 Range based memory protection

The range-based memory protection system is designed for deterministic applications to provide memory protection for systems that do not require virtual memory.

This section describes:

- Protection ranges
- Access permissions
- Protection sets

Protection ranges

A protection range is a continuous part of address space for which access permissions may be specified.

A protection range is defined by the lower boundary and the upper boundary. An address belongs to the range if:

- Lower boundary <= address < upper boundary

There are two groups of protection ranges:

- Data protection ranges specify data access permissions
- Code protection ranges specify instruction fetch permissions

The number of code and data protection ranges is implementation dependent, limited to a minimum of four and a maximum of 32 for each.

The granularity for lower and upper boundaries is 8-byte for data protection ranges and 32-byte for code protection ranges

The three least significant bits of the data protection upper and lower bound registers are not writable and always return zero.

The five least significant bits of the code protection upper and lower bound registers are not writable and always return zero.

Access permissions

Access permissions define the kind of access allowed to a protection range.

The available types are:

- Data read
- Data write
- Instruction fetch

Each access type can be separately permitted by setting the corresponding access flag.

Table 23 Access types

Access type	Flag name	Short name	Affected operation
Data read	Read enable	RE	Load
Data write	Write enable	WE	Store
Instruction fetch	Execution enable	XE	Instruction fetch

Protection sets

A complete set of access permissions defined for the whole address space used, is called a protection set.

Each protection set consists of:

- A selection of code protection ranges
- A selection of data protection ranges
- The access permissions defined for each range
- A selection of execute enabled code protection ranges
- A selection of write enabled data protection ranges
- A selection of read enabled data protection ranges

The protection set defines both data access permissions and instruction fetch permissions.

In a protection set each data protection range has associated read enable and write enable flags. Each code protection range has an associated execution enable flag.

The number of memory protection sets provided is specific to each TriCore™ implementation, limited to a minimum of two and a maximum of eight.

Having multiple protection sets allows for a rapid change of the whole set of access permissions when switching between User and Supervisor mode, or between different user tasks.

At any given time one of the sets is the current protection register set which determines the legality of memory accesses by the current task. The PSW.PRS field determines the current protection register set number.

10.2.1 Access permissions for intersecting memory ranges

The permission to access a memory location is the OR of the memory range permissions.

If one of the ranges allows it, the memory access is permitted. This means that when two ranges intersect, the intersecting regions will have the permission of the most permissive range.

For example:

- Range A is set for read/write permission
- Range B is set for read-only permission
- Therefore the intersecting region of A and B will be read/write

Nesting of ranges can be used to allow read/write access to a subrange of a larger range in which the current task is allowed read access.

10.2.2 Crossing protection boundaries

A memory access can straddle two regions defined by the protection system. The following figure shows a memory access to B (code or data) crossing the boundary of a permitted region and a 'not permitted' region of memory.

A code fetch to B that starts in a permitted region and extends into a non-permitted region triggers a MPX trap.

For a data access to B that starts in a permitted region and extends into a non permitted region it is implementation defined (not architecturally defined) as to whether or not a memory protection trap (MPR, MPW) is taken.

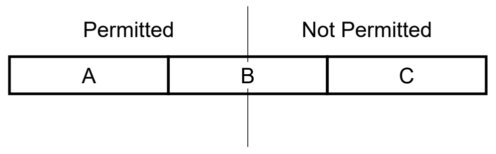


Figure 28 Protection boundaries

***Note**: To ensure deterministic behavior in all implementations of TriCore™, a region at least twice the size of the largest memory data accesses, minus one byte, should be left as a buffer between each data memory protection region. Some implementations may require a smaller buffer between regions, please refer to implementation specific documentation for details.*

10.3 Using the range based memory protection system

When the protection system is enabled, every memory access (read, write or execute) is checked for legality before the access is permitted. The permission is determined by all of the following:

- The state of the protection enable bit in the CORECON register (CORECON.PROTEN)
- The currently selected protection register set (PSW.PRS)

- The ranges selected in the protection register set
- The address ranges selected in the respective protection set fetch enable, read enable and write enable configuration registers

10.3.1 Protection enable bit

For the memory protection system to be active, the protection enable bit (CORECON.PROTEN) must be set to one (CORECON.PROTEN == 1).

If the memory protection system is disabled (CORECON.PROTEN == 0), then any access to any memory address is permitted.

10.3.2 Protection set selection

At all times, one of the protection sets is the current protection register set which determines the permission of memory accesses by the current task or Interrupt Service Routine (ISR).

The PSW.PRS field indicates the current protection register set number.

10.3.3 Address range

Instruction fetch addresses are checked against the currently selected code address ranges (in CPXE_x where x is PSW.PRS).

Data read addresses are checked against the currently selected data read address ranges (in DPRE_x where x is PSW.PRS).

Data write addresses are checked against the currently selected data write address ranges (in DPWE_x where x is PSW.PRS).

For an access to be permitted, there must be an address range that covers the address being accessed.

For instruction fetches, the PC value for the fetch is checked against the execute enabled selected code protection ranges of the current protection set. When a PC lies within any execute enabled range the access is permitted. When a PC does not lie any of the execute enabled ranges, then an MPX (Memory Protection Execute) trap is taken.

For load operations, the data address values are checked against the read enabled selected data protection ranges of the current protection set. When an address lies within any read enabled range the access is permitted. When an address does not lie within any of the read enabled ranges, then an MPR (Memory Protection Read) trap is taken.

For store operations, the data address values are checked against the write enabled selected data protection ranges of the current protection set. When an address lies within any write enabled range the access is permitted. When an address does not lie within any of the write enabled ranges, then an MPW (Memory Protection Write) trap is taken.

For load and store operations (that means those that have an atomic load and store sequence) operations, the data address values are checked against both the read and write enabled selected data protection ranges of the current protection set. When an address lies within any read enabled range and any write enabled range (it need not be the same range pair) the access is permitted. When an address does not lie both within any of the read enabled ranges and any of the write enabled range then it is not permitted. The access will take an MPR trap if there is no read enabled range, and will take an MPW trap if there is an enabled read range but no write enabled range.

The memory protection system does not differentiate between access permission levels. The data and code protection settings have the same effect, whether the permission level is currently set to Supervisor, User-1 or User-0 mode.

Supervisor mode does not automatically disable memory protection. The protection register set that is selected for Supervisor mode tasks (set-0) will normally be set up to allow write access to regions of memory that are protected from User mode access. In addition Supervisor mode tasks can execute instructions to change the protection of address regions, or to disable the protection system entirely. As Supervisor mode does not implicitly override memory protection it is possible for a Supervisor mode task to take a memory protection trap.

Saves or restores of contexts to the context save area (as distinct from loads and stores using LDLCX,LDUCX, STLCX, STUCX) do not require the permission of the memory protection system to proceed.

For a complete description of traps see Trap system.

10.3.4 Protection range register naming convention

Data protection range registers are named as follows:

- DPRn_L - Defines the lower address boundary for data range pair n
- DPRn_U - Defines the upper address boundary for data range pair n

Code protection range registers are names as follows:

- CPRn_L - Defines the lower address boundary for code range pair n
- CPRn_U - Defines the upper address boundary for code range pair n

***Note**: Range of x is implementation dependent and independent for code and data.*

10.3.5 Protection set enable register naming convention

The protection set enable registers are named as follows:

- CPXE_x - Defines the execute permission enabled code protection ranges for set-x
- DPRE_x - Defines the read permission enabled data protection ranges for set-x
- DPWE_x - Defines the write permission enabled data protection ranges for set-x

Within each of these registers range-n has permissions enabled if bit-n of the register is 1 else permission is disabled. As the number of code and data protection ranges is implementation dependent the number of bits in these registers is also implementation dependent.

10.4 Range based memory protection registers

10.4.1 Data protection range register lower bound

DPRn_L (n=0-31)	Address:	$(C000+n*8)_H$
Data protection range register x lower bound	Reset value:	Implementation $Specific_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
LOWBND															
rw															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LOWBND													**RES**		
rw													r		

Field	Bits	Type	Description
LOWBND	31:3	rw	DPRn_L lower boundary address
RES	2:0	r	Reserved The three least significant bits are not writeable and always return zero.

10.4.2 Data protection range register upper bound

DPRn_U (n=0-31) Address: $(C004+n*8)_H$

Data protection range register x upper bound Reset value: $\text{Implementation Specific}_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
UPPBND															
rw															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UPPBND													**RES**		
rw													r		

Field	Bits	Type	Description
UPPBND	31:3	rw	DPRn_U upper boundary address
RES	2:0	r	Reserved The three least significant bits are not writeable and always return zero.

10.4.3 Code protection range register lower bound

CPRn_L (n=0-31) Address: $(D000+n*8)_H$

Code protection range register x lower bound Reset value: $\text{Implementation Specific}_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
LOWBND															
rw															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LOWBND											**RES**				
rw											r				

Field	Bits	Type	Description
LOWBND	31:5	rw	CPRn_L lower boundary address

(table continues...)

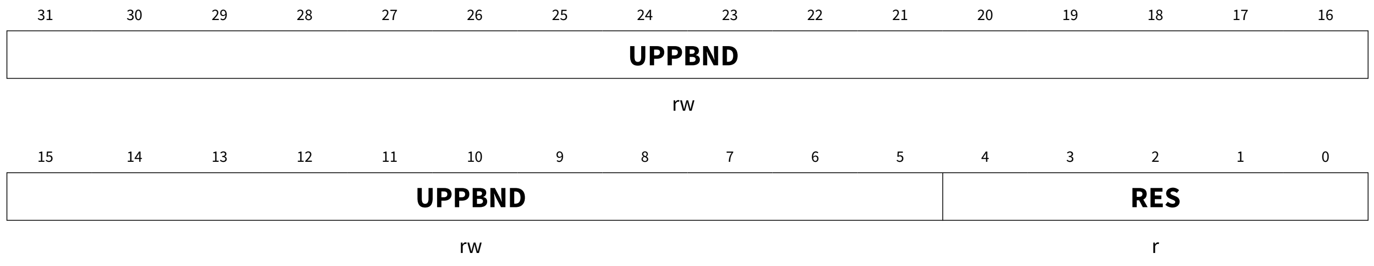
(continued)

Field	Bits	Type	Description
RES	4:0	r	Reserved The five least significant bits are not writeable and always returns zero.

10.4.4 Code protection range register upper bound

CPRn_U (n=0-31) Address: $(D004+n*8)_H$

Code protection range register x upper bound Reset value: Implementation $Specific_H$

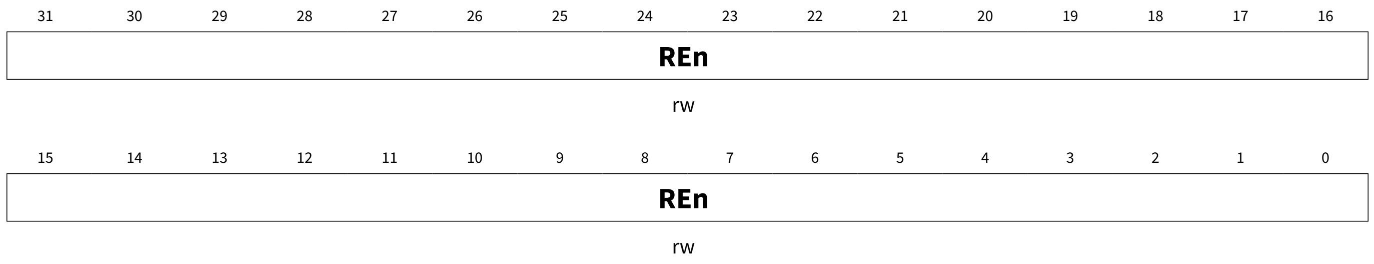


Field	Bits	Type	Description
UPPBND	31:5	rw	CPRn_U upper boundary address
RES	4:0	r	Reserved The five least significant bits are not writeable and always return zero.

10.4.5 Data protection read enable set configuration register

DPRE_x (x=0-7) Address: $(E010+x*4)_H$

Data protection read enable set configuration register x Reset value: Implementation $Specific_H$



Field	Bits	Type	Description
REn	31:0	rw	Data protection range read enable 0 : Data read accesses to data protection range[n] not permitted for set-x 1 : Data read accesses to data protection range[n] permitted for set-x ***Note**: The number of protection ranges is implementation dependent. Enable bits for unimplemented ranges are read only and return 0 when read.*

10.4.6 Data protection write enable set configuration register

DPWE_x (x=0-7) Address: (E020+x*4)$_H$

Data protection write enable set configuration register x Reset value: Implementation Specific$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
WEn															
rw															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
WEn															
rw															

Field	Bits	Type	Description
WEn	31:0	rw	Data protection range write enable 0 : Data write accesses to data protection range[n] not permitted for set-x 1 : Data write accesses to data protection range[n] permitted for set-x ***Note**: The number of protection ranges is implementation dependent. Enable bits for unimplemented ranges are read only and return 0 when read.*

10.4.7 Code protection execute enable set configuration register

CPXE_x (x=0-7) Address: (E000+x*4)$_H$

Code protection execute enable set configuration register x Reset value: Implementation Specific$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
XEn															
rw															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
XEn															
rw															

Field	Bits	Type	Description
XEn	31:0	rw	Code protection range execute enable 0 : Execute accesses to code protection range[n] not permitted for set-x 1 : Execute accesses to code protection range[n] permitted for set-x ***Note**: The number of protection ranges is implementation dependent. Enable bits for unimplemented ranges are read only and return 0 when read.*

11 Temporal protection system

The TriCore™ temporal protection system is used to guard against run-time over-run. The system primary mechanism is the temporal protection timers.

11.1 Temporal protection timers

The temporal protection timers system consists of three independent decrementing 32-bit counters, arranged to generate a Temporal Asynchronous Exception (TAE) trap (class 4, TIN 7), on decrement to zero.

The temporal protection system is enabled by setting the TPROTEN bit in the CORECON register.

A timer is activated by writing a non-zero value to the TPS_TIMERx register.

After activation, the timer will decrement by one on each CPU clock cycle.

The timer will continue to decrement until either the count value reaches zero, or the timer is de-activated by writing zero to the TPS_TIMERx register. The current timer value can be read from the TPS_TIMERx register.

On a count decrement from one to zero, the associated TEXPx bit in the TPS_CON register is set. The TEXPx bit is cleared by any write to the associated TPS_TIMERx register.

On setting any TEXPx bit in the TPS_CON register, the TPEND bit in the same register is set if TTRAP is clear. A transition of TPEND from 0 to 1 indicates that a TAE request is pending. When the TAE trap is taken, TTRAP is set to 1 which clears TPEND.

The TTRAP bit is cleared by any write to the TPS_CON register. However, attempting to clear the register while any TEXPx bit is set will cause the TPEND bit to be reset and a new TAE trap request is generated. This ensures that no time-out event is missed during the handling of another TAE trap.

11.2 Temporal protection system registers

11.2.1 TPS timer register

Definition of the Temporal Protection System Timer register.

TPS_TIMERx (x=0-2)	Address:	E404+x*4$_H$
TPS timer register x	Reset value:	Implementation specific$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Timer															
rwh															

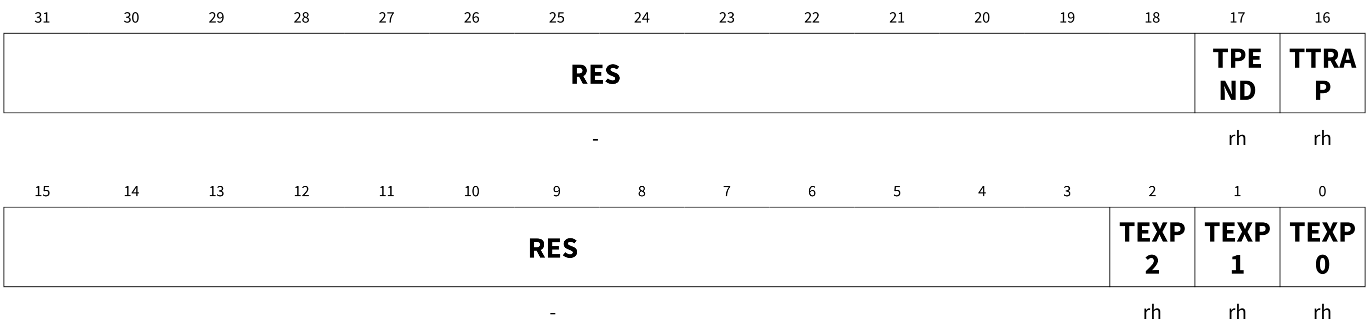
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Timer															
rwh															

Field	Bits	Type	Description
Timer	31:0	rwh	Temporal protection timer Writing zero de-activates the Timer. Writing a non-zero value starts the Timer. Any write clears the corresponding TPS_CON.TEXPx flag. Read returns the current value of Timer.

11.2.2 TPS control register

TPS_CON Address: E400$_H$

TPS control register Reset value: Implementation Specific$_H$



Field	**Bits**	**Type**	**Description**
RES	31:18	-	Reserved
TPEND	17	rh	TAE pending flag If set, indicates that a TAE trap has been requested but not yet taken. Any subsequent TAE traps are disabled. TPEND is automatically cleared when TTRAP is set.
TTRAP	16	rh	Temporal protection trap If set, indicates that a TAE trap has been taken. Any subsequent TAE traps requests are disabled. Any write to the register clears the flag and re-enables TAE traps.
RES	15:3	-	Reserved
TEXP2	2	rh	Timer2 expired flag Set when the corresponding timer expires. Cleared on any write to the TPS_TIMER2 register.
TEXP1	1	rh	Timer1 expired flag Set when the corresponding timer expires. Cleared on any write to the TPS_TIMER1 register.
TEXP0	0	rh	Timer0 expired flag Set when the corresponding timer expires. Cleared on any write to the TPS_TIMER0 register.

12 Core Performance Measurement and Analysis

Real-time measurement of core performance provides useful insights to system developers, architects, compiler developers, application developers, OS developers, and so on.

TriCore™ includes the ability to measure different performance aspects of the processor without any real-time effect on its execution. The performance measurement hardware is configured so that only a subset of performance measurements can be taken simultaneously.

The performance measurement block can be used to measure basic parameters such as:

- CPU clocks
- Instruction count
- Instruction cache hit/miss
- Data cache hit/miss (clean or dirty)

The actual parameters that may be measured are implementation specific.

The performance counters can be used in a free running manner, enabled to acquire aggregate information. Alternatively they can be used in conjunction with the debug event logic to control 'windows' of operation for an individual task, for example starting and stopping the counters dynamically to filter the measured information on some desired event.

Typical performance counter usage

The performance counters are controlled by the CCTRL CSFR register.

The performance counters can be enabled or disabled by writing the appropriate value to the counter enable CCTRL.CE bit.

Typically two parameters are always counted for base line measurement:

- The clock count
- The number of instructions issued

One of:

- Instruction cache hits
- Data cache hits

One of:

- Instruction cache misses
- Data cache clean misses

Additionally:

- Data cache dirty misses (cache write-back/eviction was required)

***Note**: Counters can only be written when they are disabled (that is not in 'counting mode'). Any attempt to write during counting mode will have no effect.*

***Note**: The counters are free running incrementors once enabled, and will roll over to zero after the maximum value is reached while setting the overflow field.*

The grouping of counter functions allows typical measurements to be clustered; that is data cache performance and instruction cache performance.

These can all be measured against the background statistics of clock cycles and instructions issued.

The start of counters is not precisely synchronized to any pipeline stage. For example, once the instruction counter is enabled to count, it starts counting all retiring instructions from that clock cycle onward. Similarly, once the instruction cache miss counter is started, it will count all the instruction cache misses from that clock cycle onward.

There are two ways to enable counters: Normal mode and Task mode (CCTRL.CM).

Normal (default mode) or task mode are configured by CCTRL.CM:

- Normal mode - The counters start counting as soon as they are enabled, and will keep counting until they are disabled
- Task mode - The counters will only count if the processor detected a debug event with the action to start the performance counters

Writing of the counters

Counters can be read any time, but they can only be written when they are not actively counting (that is when they are disabled). If the counters are disabled, then they are not considered to be in counting mode and so they can be written.

A counter is said to be in the counting mode if:

- The normal or task mode is selected AND
- The mode is active (normal mode is always active) AND
- The counter enable CE bit (in the counter control register - CCTRL) is enabled

Counter modes

The Counter Mode (CM) bit in the counter control CSFR (that is CCTRL.CM) determines the operating mode of all the counters.

In the normal mode of operation the counter increments on their respective triggers if the count enable bit in the CCTRL is set (CCTRL.CE). In task mode there is additional gating control from the debug unit which allows the data gathered in the performance counters to be filtered by some specific criteria, such as a single task for example.

Wrapping of the counters/sticky bit

The performance counters give the software some indication that the counters had wrapped (by use of a sticky bit). This helps to tell whether the counter has wrapped between two measured values.

- All performance counters are 31-bit counters with free wrapping operation
- Bit 31 of each register is sticky. It gets set when bits 30:0 wrap. It stays set until written by software

Virtualization

For a CPU supporting virtualization, the performance counters and control registers are replicated per hardware resource.

12.1 Performance counter registers

The performance counter registers are:

12.1.1 Counter control

CCTRL Address: $FC00_H$

Counter control Reset value: $0000\ 0000_H$

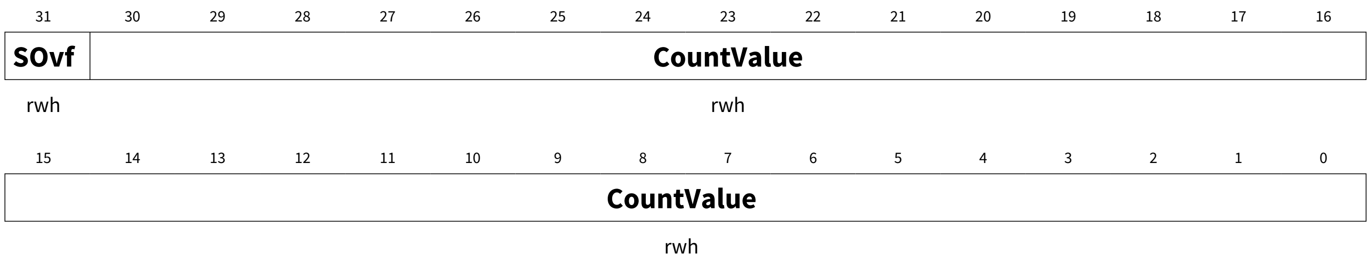
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0					**M3**			**M2**			**M1**			**CE**	**CM**
r					rw			rw			rw			rw	rw

Field	Bits	Type	Description
CM	0	rw	**Counter mode** 0_B Normal Mode. 1_B Task Mode.
CE	1	rw	**Count enable** 0_B Disable the counters: CCNT, ICNT, M1CNT, M2CNT, M3CNT. 1_B Enable the counters: CCNT, ICNT, M1CNT, M2CNT, M3CNT.
M1	4:2	rw	**M1CNT configuration**
M2	7:5	rw	**M2CNT configuration**
M3	10:8	rw	**M3CNT configuration**
0	31:11	r	**Reserved - RES**

12.1.2 CPU clock cycle count

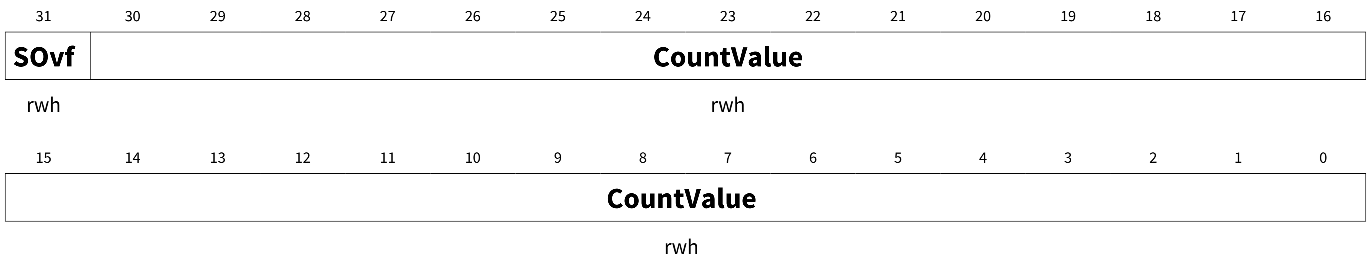
CCNT Address: $FC04_H$
CPU clock cycle count Reset value: $0000\ 0000_H$



Field	Bits	Type	Description
CountValue	30:0	rwh	**Count value** Current Count of the CPU Clock Cycles.
SOvf	31	rwh	**Sticky overflow bit** This bit is set by hardware when count value [30:0] wraps to 0 from $7FFF\_FFFF_H$. It can only be cleared by software.

12.1.3 Instruction count

ICNT Address: $FC08_H$
Instruction count Reset value: $0000\ 0000_H$



Field	Bits	Type	Description
CountValue	30:0	rwh	**Count value** Count of the Instructions Executed.
SOvf	31	rwh	**Sticky overflow bit** This bit is set by hardware when count value [30:0] wraps to 0 from 7FFF_FFFF$_H$. It can only be cleared by software.

12.1.4 Multi-count register 1

M1CNT Address: FC0C$_H$
Multi-Count Register 1 Reset value: 0000 0000$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SOvf	**CountValue**														
rwh	rwh														

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CountValue															
rwh															

Field	Bits	Type	Description
CountValue	30:0	rwh	**Count value** Count of the Selected Event.
SOvf	31	rwh	**Sticky overflow bit** This bit is set by hardware when count value [30:0] wraps to 0 from 7FFF_FFFF$_H$. It can only be cleared by software.

12.1.5 Multi-count register 2

M2CNT Address: FC10$_H$
Multi-count register 2 Reset value: 0000 0000$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SOvf	**CountValue**														
rwh	rwh														

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CountValue															
rwh															

Field	Bits	Type	Description
CountValue	30:0	rwh	**Count value** Count of the Selected Event.
SOvf	31	rwh	**Sticky overflow bit** This bit is set by hardware when count value [30:0] wraps to 0 from 7FFF_FFFF$_H$. It can only be cleared by software.

12.1.6 Multi-count register 3

M3CNT	Address:	FC14$_H$
Multi-count register 3	Reset value:	0000 0000$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SOvf	**CountValue**														
rwh	rwh														

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CountValue															
rwh															

Field	Bits	Type	Description
CountValue	30:0	rwh	**Count value** Count of the Selected Event.
SOvf	31	rwh	**Sticky overflow bit** This bit is set by hardware when count value [30:0] wraps to 0 from 7FFF_FFFF$_H$. It can only be cleared by software.

13 Core debug controller

The TriCore™ debug functionality is an interface of architecture, implementation and software tools. Users are advised that mechanisms may differ in subsequent architecture generations.

The core debug controller is designed to support real-time systems that require non-intrusive debugging. Most of the architectural state in the CPU Core and Core on-chip memories can be accessed through the system address map.

Access to the core debug is typically provided through the On-Chip Debug Support (OCDS) of the system containing the CPU.

Core debug controller features

Core debug controller features are aimed predominantly at the software development environment. It offers real-time run control and internal visibility of resources such as data and memories. Features include:

- Real-time run control (halt and restart the CPU)
- Access and update internal registers and core local memory
- Setting breakpoints and watchpoints with complex trigger conditions

Enabling the core debug controller

To enable the core debug controller, the system containing the core must set the Debug Enable bit (DE) in the Debug Status Register (DBGSR). The core debug controller is disabled when DBGSR.DE == 0, and enabled when DBGSR.DE == 1. How the DBGSR.DE bit is controlled and how the core debug controller is enabled or disabled, is system dependent. When the core debug controller is enabled, the core is said to be in debug mode.

13.1 Run control features

Real-time run control functions are accessed and controlled by address mapped reads and writes, typically by the OCDS or by any other bus master that has the appropriate authorization. The core debug controller provides hardware hooks into the core allowing the detection of debug events which result in debug actions.

Four signals are provided by the core debug controller for communication with the OCDS:

- Core break-in
  - An indication from the OCDS to the Core of a condition of interest
- Core break-out
  - An indication from the core to the OCDS of a condition of interest
- Core suspend-in
  - An indication from the OCDS to the Core to enter Halt mode
- Core suspend-out
  - An indication from the core to the OCDS of the state of the Debug Status Register (DBGSR) SUSOUT field (DBGSR.SUSOUT). This signal can be controlled by writing to the debug status register, whereas the core break-out signal can not

Features

- Single-step support in hardware
- Debug events that can cause a debug action:
  - Assertion of the external core break-In signal to the core
  - Execution of the DEBUG instruction
  - Execution of the MTCR (Move To Core Register) or the MFCR (Move From Core Register) instruction

  - Execution of the MTDCR (Move To Double Core Register) or the MFDCR (Move From Double Core Register) instruction
  - Events raised by the trigger event unit (see Trigger event unit)
- Debug actions can be one or more of the following:
  - Update debug status register
  - Indicate event on core break-out signal and/or core suspend-out signal
  - Halt CPU execution
  - Take breakpoint trap
  - Raise breakpoint interrupt
  - Control performance counters
- Real-time features:
  - Read and write of core memory and register while the core is running, with minimum intrusion (may steal cycles)
  - The service of high priority interrupt routines by use of the breakpoint Interrupt debug action

***Note**: The reading and writing of other system memory while the CPU is running can be intrusive, depending on the number of cycles that are required to perform the operation. When this happens, cycle stealing occurs.*

The programming of debug events and debug actions can occur while the CPU is running with little or no intrusion. The detection of debug events has no effect on real-time execution.

13.2 Debug events

When the core debug controller is enabled, a debug event can be generated by:

- Core break-in signal
  - See External debug event
- Execution of a debug instruction
  - See Debug instruction
- Execution of the MTCR , MFCR, MTDCR and MFDCR instruction
  - See MTCR, MFCR, MTDCR and MFDCR instructions
- A hardware event generation unit
  - See Trigger event unit

***Note**: All the debug events share the same debug action configuration (DBGACT)*

13.2.1 External debug event

An external debug event is not correlated in any way to the instruction flow, but it provides the ability to stop and gain control of the CPU without having to reset. It may take several clocks for the debug event to be recognized by the CPU if it is currently executing a multi-cycle, non-cancellable instruction (such as a context save and restore for example).

The debug action taken on the assertion of the core break-in signal is specified in the DBGACT (Debug Action) register (see Debug action Configuration). The action must be enabled in EXEVT (External Event) register (see External Event Register).

13.2.2 Debug instruction

TriCore™ supports a user mode debug instruction which can generate a debug event when the core debug controller is enabled. When the core debug controller is disabled it is treated as a NOP (No Operation). Both 16-bit and 32-bit forms of the debug instruction are provided. This feature facilitates software debug, which allows a jump to a monitor program and provides a relatively inexpensive software instrumentation and interrogation mechanism.

The debug action taken on the debug event is specified in the DBGACT (Debug Action) register (see Debug action Configuration). The action must be enabled in SWEVT (Software Debug Event) register (See Software Debug Event).

13.2.3 MTCR, MFCR, MTDCR and MFDCR instructions

A debug event is raised when an instruction is used to read or modify a non-debug Core Special Function Register (CSFR). This is achieved by executing one of the following instructions: MTCR (Move To Core Register), MTDCR (Move To Double Core Register), MFCR (Move From Core Register) or MFDCR (Move From Double Core Register). This gives the debug software the ability to monitor, detect and modify changes to CSFRs. A debug event is not raised when a MTCR, MTDCR, MFCR or MFDCR is performed to a register in the range $F000_H$ to $FDFF_H$. This range contains all dedicated debug SFRs (Special Function Registers):

- Debug Status Register (Debug Status Register)
- Core Register Access Event Register (Core Register Access Event)
- Software Debug Event Register (Software Debug Event)
- External Event Register (External Event Register)
- Trigger Event Register (TRnEVT) (Trigger Event n)
- Trigger Address Register (TRnADR) (Trigger Address n)

- Debug Monitor Start Register (Debug Monitor Start Address)
- Debug Context Pointer Register (Debug Context Save Area Pointer)
- Debug Trap Control Register (Debug Trap Control Register)
- Accumulated Trigger Information Register (Trigger Accumulator)
- Debug Action Register (Debug action Configuration)
- Debug Configuration Register (Debug Configuration Register)

Additional counter registers

- Counter Control Register - Counter control
- CPU Clock Count Register - CPU clock cycle count
- Instruction Count Register - Instruction count
- Multi-Count Register 1 - Multi-count register 1
- Multi-Count Register 2 - Multi-count register 2
- Multi-Count Register 3 - Multi-count register 3

Additional trace tegisters

- Trace Configuration Register (Trace Configuration Register)
- Trace Limiter Register (Trace Bandwidth Limiter)
- Trace Filtering Register (Trace Filter)

The debug action taken when the debug event is raised is specified in the DBGACT (Debug Action) register (see Debug action Configuration). The action must be enabled in CREVT (see Core Register Access Event) register. Configuring the debug controller or accessing performance counters will not cause a debug event.

13.2.4 Trigger event unit

The trigger event unit is responsible for generating debug events when a programmable set of debug triggers are active. Debug triggers are either:

- Code addresses
- Data accesses

These debug triggers provide the inputs to a programmable block of logic which produces debug events as its output (see Trigger Event n).

The debug action taken when the debug event is raised, is specified in the debug action register (DBGACT). See Debug action Configuration for the register definition.

13.3 Debug triggers

Each debug trigger consists of a trigger address register (TRnADR) and an associate trigger event register (TRnEVT). Pairs of debug trigger addresses are used to define address ranges.

The core debug controller can generate the following types of debug triggers:

- Execution of an instruction at a specific address
- Execution of an instruction within a range of addresses
- Loading a value from a specific address
- Loading a value from within a range of addresses
- Storing a value to a specific address
- Storing a value to within a range of addresses

The number of available debug triggers is implementation dependent.

13.3.1 Combining debug triggers

Pairs of odd and even trigger address registers may be combined to define address ranges. A trigger will be generated for an address in the range.

- Even address register <= address < odd address register

A pair of registers is defined as a range pair, by setting the RNG bit in the even EVT trigger of the pair.

When the RNG bit of the even EVT trigger is set, all settings for the range are taken from the even EVT register and the odd EVT register is ignored.

- Range0 defined by TR0ADR and TR1ADR, enabled by TR0EVT.RNG
- Range1 defined by TR2ADR and TR3ADR, enabled by TR2EVT.RNG
- Range2 defined by TR4ADR and TR5ADR, enabled by TR4EVT.RNG
- Range3 defined by TR6ADR and TR7ADR, enabled by TR6EVT.RNG

***Note**: The RNG bit of 'odd' numbered Trigger Event registers (TR1EVT, TR3EVT, and so on) is always reserved.*

13.3.2 Accumulated debug trigger information

To further aid debug the TRIG_ACC register is provided. This register contains the accumulated state of the debug triggers since the register was last cleared. Whenever a trigger is activated - whether or not it leads to a debug event - it is recorded in the TRIG_ACC register. (for range comparisons only the lower trigger activation is recorded).

For example, if TRIG_ACC.T[n] is set, then trigger-n has activated since the TRIG_ACC register was last cleared.

The TRIG_ACC register is read only and is cleared by any read, all writes are ignored.

13.3.3 Debug support for virtualization

If virtualization is present and enabled, the debug system can be configured to either debug the whole CPU or be restricted to specific virtual machines through the Debug Configuration Register register.

When virtualization is not present or disabled, any of the VM specific bit-field are reserved and will read as 0. This applies to Debug Status Register and Debug Configuration Register .

***Note**: When running in an virtualized environment, a MTCR or MTDCR to any debug register will be restricted to the hypervisor.*

13.4 Debug actions

When a debug event occurs, one or more of the following debug actions are taken depending upon the programming of the relevant event register:

- Update Debug Status Register (DBGSR)
- Indicate on core break-out signal
- Indicate on core suspend-out signal
- Halt
- Breakpoint trap
- Breakpoint trap
- Suspend Out
- Performance counter start/stop
- None
- Disabled
- Suspend in halt

13.4.1 Update Debug Status Register (DBGSR)

When a debug event occurs, the EVTSRC (Event Source), PEVT (Posted Event) and SUSOUT (Current State of Suspend Signal) fields of the Debug Status Register (DBGSR) are always updated.

SUSOUT is only updated if the DBGACT.SUSOUT field is set, and otherwise retains its value.

If virtualization is present and enabled, the EVTVMN will also be updated with the machine number associated with the event except when the event source is external. Since an external event (EVTSRC == EXEVT) is independent of the software execution, the EVTVMN will be set to 0 for external events.

13.4.2 Indicate on core break-out signal

A debug event can indicate to the OCDS that the event has occurred. Note that it is implementation dependent whether or not this signal is connected to an external pin.

13.4.3 Indicate on core suspend-out signal

The DBGSR.SUSOUT field is updated if the DBGACT.SUSOUT.

Modification of the DBGSR.SUSOUT bit will be reflected in the core suspend-out signal.

When a debug event causes a breakpoint interrupt to be posted, DBGSR.SUSOUT and the core suspend-out signal remain unchanged.

13.4.4 Halt

The debug action halt causes the halt mode to be entered. Halt mode performs a cancel of:

- All instructions after and including the instruction that caused the breakpoint if Break Before Make (BBM) is set
- All instructions after the instruction that caused the breakpoint if BBM is clear

Once these instructions have been canceled, the CPU enters halt mode where no more instructions are fetched or executed. Halt mode is entered when the DBGSR.HALT bit-field is set to 01_B. On entering halt mode the DBGSR.EVTSRC bit-field is updated.

Once in halt mode, the external debug system is used to interrogate the target through the mapping of the architectural state into the FPI address space.

While halted, the CPU does not respond to any interrupts and only resumes execution once the debug status register HALT bit is clear (DBGSR.HALT). The bit is cleared by writing 10_B to the HALT field.

It is also possible to enter halt by writing the DBGSR.HALT field. This is treated as external event and will result in the DBGSR fields being updated accordingly.

The DBGSR.HALT field is cleared by reset and the CPU will resume normal operation.

13.4.5 Breakpoint trap

The breakpoint trap enters a debug monitor routine without using any software task resource. It relies upon the following emulator resources:

- A debug monitor routine which is executed commencing at the address defined in the DMS (Debug monitor start address) register
- A 4-word area of RAM is available at the address defined in the DCX (Debug context save area pointer) register. This is used to store the critical state during the debug monitor entry sequence

When a breakpoint trap is taken, the following actions are performed:

- Write PSW to DCX + 4_H
- Write PCXI to DCX + 0_H
- Write A[10] to DCX + 8_H
- Write A[11] to DCX + C_H
- A[11] = PC
- Write A[10] with the contents of ISP if PSW.IS == 0
- PCXI.PIE = ICR.IE
- PCXI.PCPN = ICR.CCPN
- PC = DMS
- PSW.PRS = 0_H
- PSW.IO = 2_H
- PSW.GW = 0_H
- PSW.IS = 1_H
- PSW.CDE = 0_H
- PSW.CDC = 0000000_B
- ICR.IE = 0_H
- DBGTCR.DTA = 1_H

The corresponding return sequence is provided through the privileged instruction RFM (Return From Monitor).

This provides an automated route into the debug monitor routine which does not use any software task resource. The RFM (return from monitor) instruction is then used to return control to the original task. The RFM instruction is a NOP (no operation) when not in debug mode (that is DBGSR.DE == 0).

***Note**: The generation of breakpoint traps on the load or store address of any CSA access caused by a trap or interrupt is inhibited.*

Emulator space

To enable the debug monitor to operate without requiring the modification of the current memory protection settings, the following protection modifications are applied in debug mode:

- The 16-MByte region containing the DMS pointer (Base address == {DMS[31:24],000000_H}] will have MPX traps disabled for instruction fetches in debug mode
- The 16-MByte region containing the DCX pointer (Base address == {DCX[31:24],000000_H}] will have MPR and MPW traps disabled for load and store operations in debug mode

These two memory regions are referred to as emulator space. This behavior may be conditionally enabled by setting the emulator space enable bit in the COMPAT register (COMPAT.ESEN).

The cacheability of emulator space depends on the memory attributes assigned to the segments in which they reside, by the PMA registers.

Multiple breakpoint traps

On taking a breakpoint trap TriCore™ saves a debug context (PCX, PSW, A[10], A[11]) at the location indicated by the DCX register. At the end of the debug trap handler an RFM instruction is used to restore this state.

The DCX location is only able to store a single debug context. Problems therefore arise if multiple breakpoint traps are triggered. Only the state saved by the final breakpoint trap is retained, all state from the previous breakpoint traps is lost.

To prevent this situation occurring the breakpoint trap entry sequence sets the Debug Trap Active (DTA) bit in the Debug Trap Control Register (DBGTCR). This bit is used to inhibit further breakpoint traps.

The DTA bit is cleared on an RFM instruction and set on a breakpoint trap (It may also be set and cleared by MTCR).

A breakpoint trap may only be taken in the condition DTA == 0. Taking a breakpoint trap sets the DTA bit to one. Further breakpoint traps are therefore disabled until such time as the breakpoint trap handler clears the DTA bit or until the breakpoint trap handler terminates with a RFM.

After an application reset the DTA bit is set to one. The register must therefore be cleared before a debug trap may be taken.

13.4.6 Breakpoint interrupt

One of the possible debug actions to be taken on a debug event, is to raise a breakpoint interrupt. The interrupt priority is programmable and is defined in the control register associated with the breakpoint interrupt.

The architecture allows a debug event to raise one of four breakpoint interrupts, each of which can have its own interrupt priority. The number of breakpoint interrupts is implementation dependent.

The breakpoint interrupt allows a flexible debug environment to be defined which is capable of satisfying many of the requirements for efficient debugging of a real-time system. For example, the execution of safety critical code can be preserved while the debugger is active.

Breakpoint interrupts can be used to provide the conventional debug model available in traditional micro-controllers, where a breakpoint stops the processor, by simply assigning the highest interrupt priority level to the debug monitor routine or by ensuring interrupts are disabled in the debug monitor routine. It also provides the flexibility for critical interrupts to be programmed with a higher priority than the debug monitor routine. The advantages of this are:

- The debug monitor routine can be interrupted in an identical manner to any other interrupt by a higher level interrupt. This allows the CPU to service critical interrupts while the debug monitor routine is running
- Any debug events posted in a critical routine are postponed until the CPU priority drops below that of the debug monitor routine

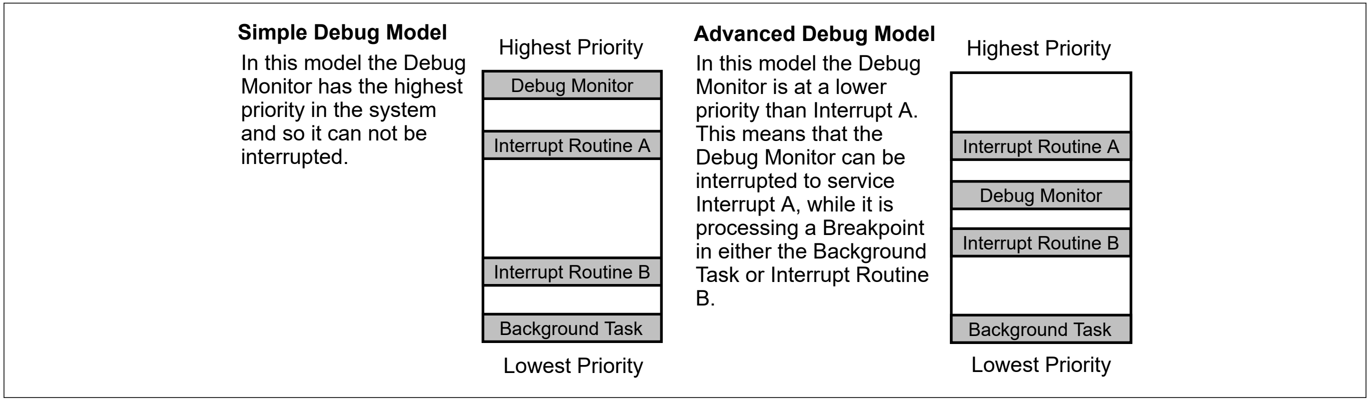

Simple Debug Model
In this model the Debug Monitor has the highest priority in the system and so it can not be interrupted.
Highest Priority
Debug Monitor
Interrupt Routine A
Interrupt Routine B
Background Task
Lowest Priority
Advanced Debug Model
In this model the Debug Monitor is at a lower priority than Interrupt A. This means that the Debug Monitor can be interrupted to service Interrupt A, while it is processing a Breakpoint in either the Background Task or Interrupt Routine B.
Highest Priority
Interrupt Routine A
Debug Monitor
Interrupt Routine B
Background Task
Lowest Priority

Figure 29 **Debug monitor routine - Simple and advanced models**

Posted breakpoint interrupts

The situation needs to be considered where a breakpoint interrupt targeted at the CPU is at an interrupt priority level below the current CPU priority. In the advanced model in the previous figure for example, if a breakpoint interrupt is set in interrupt routine 'A' it is a problem, because the debug monitor routine is programmed to be at a lower priority than the current task.

This scenario is indicated by posting a breakpoint interrupt at the interrupt level associated with the breakpoint. Therefore, when the CPU interrupt priority level falls below that of the debug monitor routine, the debug monitor routine is entered. In order to indicate to the monitor routine that the breakpoint was postponed, the Posted Event bit (PEVT) in the debug status register is set when the breakpoint interrupt is posted. It is the responsibility of the breakpoint interrupt handler to check this bit in the debug status register and to subsequently clear that bit if necessary.

***Note**:* *DBGSR.SUSOUT is not updated when a breakpoint interrupt is posted. DBGSR.EVTSRC is always updated regardless of whether or not a breakpoint interrupt is posted.*

Interrupts to other targets

As well as being targeted at the CPU, a breakpoint interrupt can be targeted at other cores in the system.

13.4.7 Suspend Out

The suspend out signal can be asserted when a debug event occurs, or when the suspend-in signal is set, and must be cleared by software or by removal of the Suspend In signal.

13.4.8 Performance counter start/stop

When the performance counter is operating in task mode, the counters are started and stopped by debug actions. All event registers allow the counters to either be started or stopped.

The trigger event registers also allow the mode to be toggled to active (start) or inactive (stop). This allows a single RTE to be used to control the performance counter, in certain applications.

13.4.9 None

No action is implemented through the EVTA field of the DBGACT register, however the suspend out signal, performance count and DBGSR register updates still occur as normal for an event.

13.4.10 Disabled

The event is disabled and no actions occur: the suspend out signal, performance counter control and DBGSR register ignore the event.

13.4.11 Suspend in halt

When the Suspend In signal is asserted, halt mode is always entered so long as debug is enabled. The CPU remains in halt mode so long as suspend in is asserted. When suspend in is negated, the CPU is released from halt.

This facility is implemented so that in a multi core system, several cores can be halted and released from halt simultaneously.

13.5 Priority of Debug Events

It is possible for multiple trigger points to be activated simultaneously. The trigger associated with the oldest instruction in the pipeline is dealt with first. In addition, simultaneous Trigger points associated with the same point in the pipeline are prioritized from highest to lowest as:

- Assertion of External Input (asynchronous)
- Programmable bank triggers on PC
  - When multiple triggers are active, 0 has the highest priority and 7 the lowest
- MTDCR/MFDCR Instruction
- MTCR/MFCR Instruction
- Debug Instruction
- Programmable triggers on Address
  - When multiple triggers are active, 0 has the highest priority and 7 the lowest

13.6 Call tracing

The tracing of subroutine calls in a TriCore™ system is performed using the PSW based call depth counter and the CDO trap handler.

The sequence followed for call tracing is as follows:

1. The PSW based call depth counter is set so as to generate a CDO trap on every subroutine call. (PSW.CDC = 1111110_B)
2. The call depth counting system is enabled. (PSW.CDE = 1)
3. When the next CALL is attempted, a CDO trap will be taken instead of the subroutine call
4. The CDO trap handler then performs the required trace function
5. The CDO trap handler clears the PSW.CDE bit of the trapping context in memory
6. The CDO trap handler executes a return from exception (RFE). This restores the trapping context from memory, this time with the call depth tracing disabled. (PSW.CDE = 0)
7. The original CALL is executed. As the call depth tracing system is now disabled (PSW.CDE=0) the subroutine call will be successful

- Whenever the PSW is saved by a CALL instruction the CDE bit is forced to "1"
- The state of the PSW.CDE bit at the start of a subroutine is "1"

In a call tracing sequence the PSW.CDE bit has a "one-shot" operation, being disabled for a single subroutine call after being cleared by the CDO trap.

For more information, please refer to the CALL instruction in the instruction set volume of this manual (volume 2).

13.7 The debug control registers

The debug status register (DBGSR) contains information about the current status of the core debug controller hardware in the CPU core:

- A bit to indicate whether the debug is enabled
- The source of the last debug event

Each source of a debug event has an associated register which defines the debug actions to be taken when the debug event is raised. These registers may contain extra information about the criteria that must be met for the debug event to be raised, such as the combination of debug triggers for example.

13.8 Debug control registers - summary

Core debug controller registers.

Table 24 Debug control registers summary

Register	Description	Offset address
DBGSR	Debug status register	$FD00_H$
EXEVT	External event register	$FD08_H$
CREVT	Core register access event register	$FD0C_H$
SWEVT	Software debug event register	$FD10_H$
DBGACT	Debug action register	$FD14_H$
TRIG_ACC	Trigger accumulator register	$FD30_H$
DMS	Debug monitor start address register	$FD40_H$
DCX	Debug context save area pointer register	$FD44_H$
DBGTCR	Debug trap control register	$FD48_H$
DBGCFG	Debug configuration register	$FD4C_H$
TR0EVT	Trigger event 0 configuration register	$F000_H$
TR0ADR	Trigger event 0 address register	$F004_H$
TR1EVT	Trigger event 1 configuration register	$F008_H$
TR1ADR	Trigger event 1 address register	$F00C_H$
TR2EVT	Trigger event 2 configuration register	$F010_H$
TR2ADR	Trigger event 2 address register	$F014_H$
TR3EVT	Trigger event 3 configuration register	$F018_H$
TR3ADR	Trigger event 3 address register	$F01C_H$
TR4EVT	Trigger event 4 configuration register	$F020_H$
TR4ADR	Trigger event 4 address register	$F024_H$
TR5EVT	Trigger event 5 configuration register	$F028_H$
TR5ADR	Trigger event 5 address register	$F02C_H$
TR6EVT	Trigger event 6 configuration register	$F030_H$
TR6ADR	Trigger event 6 address register	$F034_H$
TR7EVT	Trigger event 7 configuration register	$F038_H$
TR7ADR	Trigger event 7 address register	$F03C_H$

13.9 Debug control registers

13.9.1 Debug Status Register

Reset for Boot Execute is $0000\ 0000_H$
Reset for Boot Halt is $0000\ 0002_H$

For configurations without Virtualization, or if Virtualization is disabled, the fields [18:16] are reserved and will read as 0

DBGSR	Address:	FD00$_H$
Debug Status Register	Reset value:	0000 0000$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0													**EVTVMN**		
r													rh		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0			**EVTSRC**					**PEVT**	**0**		**SUSOUT**	**SUSIN**	**HALT**		**DE**
r			rh					rwh	r		rwh	rh	rwh		rh

Field	Bits	Type	Description
DE	0	rh	**Debug enable** Determines whether the CDC is enabled or not. 0$_B$ The CDC is disabled. 1$_B$ The CDC is enabled.
HALT	2:1	rwh	**CPU Halt request, status field** HALT can be set or cleared by software.HALT[0] is the actual Halt bit. HALT[1] is a mask bit to specify whether or not HALT[0] is to be updated on a software write. HALT[1] is always read as 0. HALT[1] must be set to 1 in order to update HALT[0] by software (R: read; W: write). 00$_B$ R: CPU running. W: HALT[0] unchanged. 01$_B$ R: CPU halted. W: HALT[0] unchanged. 10$_B$ R: Not Applicable. W: reset HALT[0]. 11$_B$ R: Not Applicable. W: If DBGSR.DE == 1 (The CDC is enabled), set HALT[0]. If DBGSR.DE == 0 (The CDC is not enabled), HALT[0] is left unchanged.
SUSIN	3	rh	**Suspend-in Halt** State of the Suspend-In signal. 0$_B$ The Suspend-In signal is negated. The CPU is not in Halt Mode, (except when the Halt mechanism is set following a Debug Event or a write to DBGSR.HALT). 1$_B$ The Suspend-In signal is asserted. The CPU is in Halt Mode.
SUSOUT	4	rwh	**Current state of the core suspend-out signal** 0$_B$ Core suspend-out inactive. 1$_B$ Core suspend-out active.
PEVT	7	rwh	**Posted event** 0$_B$ No posted event. 1$_B$ Posted event.

(table continues...)

(continued)

Field	Bits	Type	Description
EVTSRC	12:8	rh	**Event source** 00_H EXEVT 01_H CREVT 02_H SWEVT 10_H 16 + n TRnEVT (n = 0, 7) … Other = Reserved.
EVTVMN	18:16	rh	**Event virtual machine number** The number of the VM running when the event occurred unless it was an external event, in which case forced to zero.
0	6:5,15:13, 31:19	r	**Reserved - RES**

13.9.2 External Event Register

EXEVT Address: $FD08_H$
External Event Register Reset value: $0000\ 0000_H$

31–16
0
r

15–8	7–6	5–4	3	2–1	0
0	CNT	0	BBM	0	EN
r	rw	r	rw	r	rw

Field	Bits	Type	Description
EN	0	rw	**External event enabled** External Event enable. If enabled, the settings defined in DGBACT will determine if the external event is active or not. 0_B Disabled. 1_B Event enabled according to the settings defined in DBGACT.
BBM	3	rw	**Break before Make (BBM) or break after make (BAM) Selection** 0_B Break after make (BAM). 1_B Break before make (BBM).
CNT	7:6	rw	**Counter** When this event occurs adjust the control of the performance counters in task mode as follows: 00_B No change. 01_B Start the performance counters. 10_B Stop the performance counters. 11_B Toggle the performance counter control (i.e. start it if it is currently stopped, stop it if it is currently running).
0	2:1,5:4,31 :8	r	**Reserved - RES**

13.9.3 Core Register Access Event

CREVT	Address:	FD0C$_H$
Core Register Access Event	Reset value:	0000 0000$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								**CNT**		**0**		**BBM**	**0**		**EN**
r								rw		r		rw	r		rw

Field	Bits	Type	Description
EN	0	rw	**Event enable** Event enable. If enabled, the settings defined in DGBACT will determine if the event is active or not. 0_B Disabled. 1_B Event enabled according to the settings defined in DBGACT.
BBM	3	rw	**Break before make (BBM) or break after make (BAM) Selection** 0_B Break after make (BAM). 1_B Break before make (BBM).
CNT	7:6	rw	**Counter** When this event occurs adjust the control of the performance counters in task mode as follows: 00_B No change. 01_B Start the performance counters. 10_B Stop the performance counters. 11_B Toggle the performance counter control (i.e. start it if it is currently stopped, stop it if it is currently running).
0	2:1,5:4,31:8	r	**Reserved - RES**

13.9.4 Software Debug Event

SWEVT	Address:	FD10$_H$
Software Debug Event	Reset value:	0000 0000$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								**CNT**		**0**		**BBM**	**0**		**EN**
r								rw		r		rw	r		rw

Field	Bits	Type	Description
EN	0	rw	**Event enable** Event enable. If enabled, the settings defined in DGBACT will determine if the event is active or not. 0_B Disabled. 1_B Event enabled according to the settings defined in DBGACT.
BBM	3	rw	**Break before make (BBM) or break after make (BAM) Selection** 0_B Break after make (BAM). 1_B Break before make (BBM).
CNT	7:6	rw	**Counter** When this event occurs adjust the control of the performance counters in task mode as follows: 00_B No change. 01_B Start the performance counters. 10_B Stop the performance counters. 11_B Toggle the performance counter control (i.e. start it if it is currently stopped, stop it if it is currently running).
0	2:1,5:4,31:8	r	**Reserved - RES**

13.9.5 Debug action Configuration

Debug action associated with the enabled debug event (CREVT, SWEVT, TRnEVT and EXEVT)

DBGACT	Address:	$FD14_H$
Debug action Configuration	Reset value:	$0000\ 0000_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0											**SUSP**	**BOD**	**EVTA**		
r											rw	rw	rw		

Field	Bits	Type	Description
EVTA	2:0	rw	**Event Associated** Debug Action associated with the Debug Event: 000_B BOD=0: Disabled.BOD=1: Disabled. 001_B BOD=0: Pulse BRKOUT Signal.BOD=1: None. 010_B BOD=0: Halt and pulse BRKOUT Signal.BOD=1: Halt. 011_B BOD=0: Breakpoint trap and pulse BRKOUT Signal.BOD=1: Breakpoint trap. 100_B BOD=0: Breakpoint interrupt 0 and pulse BRKOUT Signal.BOD=1: Breakpoint interrupt 0. 101_B BOD=0: If implemented, breakpoint interrupt 1 and pulse BRKOUT Signal.BOD=1: If implemented, breakpoint interrupt 1. Note: If not implemented, None. 110_B BOD=0: If implemented, breakpoint interrupt 2 and pulse BRKOUT Signal.BOD=1: If implemented, breakpoint interrupt 2. Note: If not implemented, None. 111_B BOD=0: If implemented, breakpoint interrupt 3 and pulse BRKOUT Signal.BOD=1: If implemented, breakpoint interrupt 3. Note: If not implemented, None.
BOD	3	rw	**Breakout Disable** 0_B BRKOUT signal asserted according to the action specified in the EVTA field. 1_B BRKOUT signal not asserted. This takes priority over any assertion generated by the EVTA field.
SUSP	4	rw	**CDC Suspend-Out Signal State** Value to be assigned to the CDC suspend-out signal when the event is raised.
0	31:5	r	**Reserved - RES**

13.9.6 Trigger Event n

TRnEVT (n=0-7) Address: $(F000+n*8)_H$
Trigger Event n Reset value: $0000\ 0000_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0			ALD	AST	0										
r			rw	rw	r										

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0		RNG	TYP	0				CNT		0		BBM	0		EN
r		rw	rw	r				rw		r		rw	r		rw

Field	Bits	Type	Description
EN	0	rw	**Trigger enable** Trigger enable. If enabled, the settings defined in DGBACT will determine if the trigger is active or not.Once an even numbered comparator has been set to range, the EN settings of its associated upper neighbour will be ignored. 0_B Disabled. 1_B Event enabled according to the settings defined in DBGACT.
BBM	3	rw	**Break before make (BBM) or break after make (BAM) Selection** Code triggers BBM or BAM selection. ***Note**: Data access and data/code combination access triggers can only create BAM Debug Events. When these triggers occur, TRnEVT.BBM is ignored.* 0_B Triggers Break After Make (BAM). 1_B Triggers Break Before Make (BBM). (will default to BAM for data or combined triggers regardless of this setting)
CNT	7:6	rw	**Counter** When this event occurs adjust the control of the performance counters in task mode as follows: 00_B No change. 01_B Start the performance counters. 10_B Stop the performance counters. 11_B Toggle the performance counter control (i.e. start it if it is currently stopped, stop it if it is currently running).
TYP	12	rw	**Input selection** 0_B Address 1_B PC
RNG	13	rw	**Compare type** Once an even numbered comparator has been set to range, the EVTR settings of its associated upper neighbour will be ignored. 0_B Equality 1_B Range
AST	27	rw	**Address store** Used in conjunction with TYP=0
ALD	28	rw	**Address load** Used in conjunction with TYP=0
0	2:1,5:4,11:8,26:14,31:29	r	**Reserved - RES**

13.9.7 Trigger Address n

TRnADR (n=0-7)	Address:	$(F004+n*8)_H$
Trigger Address n	Reset value:	$0000\ 0000_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ADDR															
rw															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDR															
rw															

Field	Bits	Type	Description
ADDR	31:0	rw	**Comparison address** ***Note**: For PC comparison, bit[0] is always zero.*

13.9.8 Trigger Accumulator

***Note**: This register is cleared by any read operation, write operations are ignored.*

TRIG_ACC Address: $FD30_H$
Trigger Accumulator Reset value: $0000\ 0000_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								**T7**	**T6**	**T5**	**T4**	**T3**	**T2**	**T1**	**T0**
r								rh	rh	rh	rh	rh	rh	rh	rh

Field	Bits	Type	Description
T0	0	rh	**Trigger-0** active since last cleared
T1	1	rh	**Trigger-1** active since last cleared
T2	2	rh	**Trigger-2** active since last cleared
T3	3	rh	**Trigger-3** active since last cleared
T4	4	rh	**Trigger-4** active since last cleared
T5	5	rh	**Trigger-5** active since last cleared
T6	6	rh	**Trigger-6** active since last cleared
T7	7	rh	**Trigger-7** active since last cleared
0	31:8	r	**Reserved - RES**

13.9.9 Debug Monitor Start Address

The DMS reset value is {A0000$_H$,001$_B$,CORE_ID,000000$_B$}.

DMS	Address:	FD40$_H$
Debug Monitor Start Address s	Reset value:	A000 0XX0$_H$

31–16
DMSValue
rw

15–1	0
DMSValue	**0**
rw	r

Field	Bits	Type	Description
DMSValue	31:1	rw	**Debug monitor start address** The address at which monitor code execution begins when a breakpoint trap is taken.
0	0	r	**Reserved - RES**

13.9.10 Debug Context Save Area Pointer

The reset value of the DCX register is {A0000$_H$,010$_B$,CORE_ID,000000$_B$}.

DCX	Address:	FD44$_H$
Debug Context Save Area Pointer	Reset value:	A000 0XX0$_H$

31–16
DCXValue
rw

15–6	5–0
DCXValue	**0**
rw	r

Field	Bits	Type	Description
DCXValue	31:6	rw	**Debug context save area pointer** Address where the debug context is stored following a breakpoint trap.
0	5:0	r	**Reserved - RES**

13.9.11 Debug Trap Control Register

DBGTCR	Address:	FD48$_H$
Debug Trap Control Register	Reset value:	0000 0001$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0															**DTA**
r															rwh

Field	Bits	Type	Description
DTA	0	rwh	**Debug trap active bit** A breakpoint trap may only be taken in the condition DTA == 0. Taking a breakpoint trap sets the DTA bit to one. Further breakpoint traps are therefore disabled until such time as the breakpoint trap handler clears the DTA bit or until the breakpoint trap handler terminates with a RFM. 0_B No breakpoint trap is active. 1_B A breakpoint Trap is active
0	31:1	r	**Reserved - RES**

13.9.12 Debug Configuration Register

The TC and TCP bits enable that a tool can reliably configure this register before or after it is configured by SW. The use case is that SW configures it for Trace Based Measurement (TBM) but a debug tool must still have all options. The simple rule is: SW will write the register with TC and TCP set 0, a tool will always write with both bits set 1

For configurations without Virtualization, or if Virtualization is disabled, the fields [23:16] becomes read only and will read as 0

DBGCFG Address: $FD4C_H$

Debug Configuration Register Reset value: $0000\ 0000_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0								**VM7**	**VM6**	**VM5**	**VM4**	**VM3**	**VM2**	**VM1**	**VM0**
r								rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EN	**0**													**TCP**	**TC**
rw	r													w	rw

Field	Bits	Type	Description
TC	0	rw	**Tool control** 0_B Control by SW. The register is updated with the written value. Shall be used by SW. 1_B Control by Tool. Only if TCP is 1 the register is updated with the written value. Shall be used by a tool.

(table continues...)

(continued)

Field	Bits	Type	Description
TCP	1	w	**Tool control protection** 0_B TC is not changed. Shall be used by SW. 1_B TC is set to the written value. The new TC value is already effective for this write. Shall be used by a tool.
EN	15	rw	**Debug enable** DBGSR.DE must be 1 to use the core debug functionality. 0_B Disabled. 1_B Enabled.
VMn (n=0-7)	n+16	rw	**Debug enable for virtual machine n** For CPUs with virtualization present and enabled, both EN and VM{n} bits must be set to enable debugging. 0_B Debug for VMn is blocked. 1_B Debug for VMn is enabled if debugging is enabled (EN == 1).
0	14:2,31:24	r	**Reserved - RES**

13.10 Core tracing configuration

In order to allow for different tools or even software to operate without interference trace and debug can be configured independently. For instance the a software task may want to configure trace based measurements without interfering with a debugger connection.

Trace Filtering

In order to reduce the amount of operations sent to the trace system, the CPU trace can be restricted to specific event types.

- Relative addressing ST/LD instructions
- Absolute addressing ST/LD instructions
- CSA context save/restore events
- Use of specific base address register ST/LD instructions

Trace Bandwidth Limiter

To better match the CPU bandwidth with that of the trace system, the CPU implements a virtual FIFO to limit trace output. The limiter is disabled after reset and can be configured to stall CPU operation when a high rate of events are being traced. When the limiter is enabled and the virtual FIFO reaches its limit the CPU execution is stalled affecting real-time behavior.

13.11 Trace configuration registers

The trace configuration registers are:

13.11.1 Trace Configuration Register

The TC and TCP bits enable that a tool can reliably configure this register before or after it is configured by SW. The use case is that SW configures it for Trace Based Measurement (TBM) but a debug tool must still have all options. The simple rule is: SW will write the register with TC and TCP set 0, a tool will always write with both bits set 1

For configurations without Virtualization, or if Virtualization is disabled, the fields [23:16] are reserved and will read as 0

TRCCFG Address: $FD50_H$

Trace Configuration Register Reset value: $0000\ 0000_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0								**VM7**	**VM6**	**VM5**	**VM4**	**VM3**	**VM2**	**VM1**	**VM0**
r								rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EN	**0**													**TCP**	**TC**
rw	r													w	rw

Field	Bits	Type	Description
TC	0	rw	**Tool control** 0_B Control by SW. The register is updated with the written value. Shall be used by SW. 1_B Control by Tool. Only if TCP is 1 the register is updated with the written value. Shall be used by a tool.
TCP	1	w	**Tool control protection** 0_B TC is not changed. Shall be used by SW. 1_B TC is set to the written value. The new TC value is already effective for this write. Shall be used by a tool.
EN	15	rw	**Trace enable** For CPUs with virtualization and Virtualization is enabled, both EN and VMn bits must be set to enable tracing. 0_B Disabled. 1_B Enabled.
VMn (n=0-7)	n+16	rw	**Trace enable for virtual machine n** These bits only exist in CPUs with Virtualization and Virtualization is enabled. 0_B Trace for VMn is blocked. 1_B Trace {program, data} for VMn is output if tracing is enabled (EN == 1).
0	14:2,31:24	r	**Reserved - RES**

13.11.2 Trace Filter

The TC and TCP bits enable that a tool can reliably configure this register before or after it is configured by SW. The use case is that SW configures it for Trace Based Measurement (TBM) but a debug tool must still have all options. The simple rule is: SW will write the register with TC and TCP set 0, a tool will always write with both bits set 1

SLF, AA and CSA filters are applied independently.

TRCFILT Address: $FD54_H$

Trace Filter Reset value: $0000\ 0000_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
A15	**A14**	**A13**	**A12**	**A11**	**A10**	**A9**	**A8**	**A7**	**A6**	**A5**	**A4**	**A3**	**A2**	**A1**	**A0**
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0							**CSA**	**AA**	**0**	**SLF**		**0**		**TCP**	**TC**
r							rw	rw	r	rw		r		w	rw

Field	Bits	Type	Description
TC	0	rw	**Tool control** 0_B Control by SW. The register is updated with the written value. Shall be used by SW. 1_B Control by Tool. Only if TCP is 1 the register is updated with the written value. Shall be used by a tool
TCP	1	w	**Tool control protection** 0_B TC is not changed. Shall be used by SW 1_B TC is set to the written value. The new TC value is already effective for this write. Shall be used by a tool
SLF	5:4	rw	**Store/Load filter for relative addressing ST/LD instructions** SLF, AA, and CSA filters are applied independently. 00_B All relative addressing ST/LD instructions pass this filter 01_B Only ST/LD instructions with short offset off10 pass this filter 10_B Only ST/LD instructions which use A[n] when TRCFLT.An==0 pass this filter 11_B Only ST/LD instructions with either off10 or A[n] when TRCFLT.An==0 pass this filter
AA	7	rw	**Filter for absolute addressing ST/LD instructions** Only taken into account if SLF is 10_B or 11_B 0_B ST/LD instructions with absolute addressing pass this filter 1_B ST/LD instructions with absolute addressing are blocked
CSA	8	rw	**Filter for context save area** Value to be assigned to the CDC suspend-out signal when the event is raised. 0_B CSA context save and restore pass this filter. 1_B CSA context save and restore trace is blocked
An (n=0-15)	n+16	rw	**SLF trace Filter for CPU address register An** Only taken into account if SLF is 10_B or 11_B 0_B ST/LD instructions using A[n] passes this filter 1_B ST/LD instructions using A[n] are blocked
0	3:2,6,15:9	r	**Reserved - RES**

13.11.3 Trace Bandwidth Limiter

The TC and TCP bits enable that a tool can reliably configure this register before or after it is configured by software. The use case is that software configures it for Trace Based Measurement (TBM) but a debug tool must still have all options. The simple rule is: software will write the register with TC and TCP set 0, a tool will always write with both bits set 1

TRCLIM Address: $FD58_H$

Trace Bandwidth Limiter Reset value: $0000\ 0000_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0					**DEPTH**			**STALL**				**0**		**TCP**	**TC**
r					rw			rw				r		w	rw

Field	Bits	Type	Description
TC	0	rw	**Tool control** 0_B Control by SW. The register is updated with the written value. Shall be used by SW. 1_B Control by Tool. Only if TCP is 1 the register is updated with the written value. Shall be used by a tool.
TCP	1	w	**Tool control protection** 0_B TC is not changed. Shall be used by SW. 1_B TC is set to the written value. The new TC value is already effective for this write. Shall be used by a tool.
STALL	7:4	rw	**Number of CPU write back stall cycles** 0_H The bandwidth limiter is disabled. 1_H Stall the write buffer by 1 cycle. .... F_H Stall the write buffer by 15 cycles.
DEPTH	10:8	rw	**Depth of the simulated FIFO** The simulated FIFO depth is DEPTH+1. Once the simulated FIFO depth is reached, the CPU write buffer will be stalled and thus no new trace generated for the STALL number of CPU clock cycles.
0	3:2,31:11	r	**Reserved - RES** Read as 0.

14 Real-time virtualization extension

This chapter describes the optional TriCore™ real-time virtualization extension. Support for this optional extension can be independent for each TC1.8 CPU in a system. Even if the real-time virtualization extension is implemented in a CPU, software has the option of disabling the extension. When this extension is disabled, the CPU provides a compatible architecture to a CPU without the extension implemented.

14.1 Functional overview

The virtualization extension provides the following features:

- Instruction extensions for handling virtual machine (VM) management
- Support for a hypervisor, one real-time VM and up to six other virtual machines
- Replication of architectural state to support multiple resident VMs
- Low overhead switching between resident VMs and hypervisor using TriCore™'s CSA mechanism
- Separate set of hypervisor traps with its own vector table
- Second level MPU to ensure freedom from interference between guest VMs
- Interrupt mechanism enhanced to provide interrupt delivery to specific VMs

14.2 Definitions

TriCore™ virtualization extensions facilitate better encapsulation of the resources allocated to software partitions, improving both device utilization and application separation without compromising real-time performance. The extension supports a hypervisor, one real-time virtual machine, and up to 6 other virtual machines.

Hardware resources

A hardware resource represents a collection of architectural and software state dedicated to a given software partition.

Virtual machine

Encapsulated software partition or partitions running in a virtualized system. The software partition can be as simple as a single task or a complete RTOS. Virtual machines are numbered VM0 to VM7.

Guest virtual machine

VM1 to VM7 are hosted by the hypervisor (VM0).

Hypervisor

The hypervisor (HV) or virtual machine monitor is a combination of architectural state and software dedicated to creating, running and controlling virtual machines.

***Note**: VM0 is used as an alias for the hypervisor.*

Architectural state

The TriCore™ 1.8 state is composed of three main groups:

- Guest VM state
  - Read-only state
    - Primarily identification registers, and hardware features

    - CSA managed state
        - This state includes the registers saved and restored using the CSA mechanism. This includes GPR's, PC, PSW, and PCXI
    - Supervisor state
        - This state includes all registers used by the software, under the control of the RTOS
    - MPU state
        - A collection of data and code address ranges with specific protection attributes organized as protection sets
- Hypervisor state
    - The set of resources for the exclusive use of the hypervisor
- System state
    - State required by external components of the system (primarily debug and trace)

14.3 Virtualization register overview

The virtualization extension provides three hardware resources, each providing a common view of the architectural state to the virtual machine associated with the resource. With the exception of the hypervisor exclusive state which is only accessible to the hypervisor, all other states are accessible through the same CSFR indices in all hardware resources.

Naming convention

Three sets of dedicated hardware resources are provided to support virtualization; HRHV, HRA, and HRB.

The mapping of virtual machines to hardware resources is as follows:

- HRHV – Hypervisor hardware resource (VM0)
- HRA – Real time virtual machine hardware resource (VM1)
- HRB – Other virtual machine hardware resource (VM2-7)

If only two virtual machines (VM1 and VM2) are used, the state for both hosted machines is resident in HRA and HRB and the hypervisor (VM0) software switches between HRA and HRB. If two or more virtual machines (from VM2 to VM7) are to be used then the hypervisor (VM0) software swaps the virtual machine state between HRB and memory when required. If virtualization is not enabled then the HRA resource is used, and so by definition none of the Hypervisor state is accessible.

Hardware resource register replication

Each hardware resource has dedicated sets of CSFRs to provide the hosted VM state, with the following exceptions:

- Read only state is not replicated
- CSA managed state is not replicated

The following state is only available in HRHV

- Hypervisor registers are not replicated
- System/shared state

The Interrupt Control Register is replicated per VM in the HRHV state, but the respective ICR (VM1_ICR) is accessible in HRA, and VMn_ICR (n = 2..7) is accessible in HRB when VMn is hosted.

Each CSFR register is identified by the prefix HRHV_, HRA_ or HRB_. For example, FCX is replicated and renamed as HRHV_FCX, HRA_FCX, HRB_FCX.

- Software running in HRHV has access to all registers
- Software running in HRA has access to HRA registers only
- Software running in HRB has access to HRB registers only

Hypervisor register access

Access to CSFR state is performed from the local core through the MTCR or MFCR instructions, and the MTDCR or MFDCR instructions. These instructions use a 16-bit constant (Const[15:0]) for their access address.

For MTDCR and MFDCR operations const[2] should always be "0" else the operation is accessing a misaligned location and will lead to an OPD trap.

The hypervisor can target any CSFR across all hardware resources. Offset bits const[1:0] are used as a selector as shown in the following table.

Table 25 CSFR access hardware resource selection rules from the local core

Currently active hardware resource	Const[1:0] == 00_B	Const[1:0] == 01_B	Const[1:0] == 10_B	Const[1:0] == 11_B
HRHV	Access to own CSFR register set	Access to HRA CSFR register set	Access to HRB CSFR register set	Misaligned CSFR access, OPD trap
HRA		Misaligned CSFR access, OPD Trap		
HRB				

A MTCR or MFCR operation in VM1-VM7 that targets an undefined CSFR location or a MTDCR or MFDCR operation in VM1-VM7 that targets an undefined double CSFR register location will result in a HV trap (class 4 HVCSFR) if virtualization is configured and enabled. In all other cases an OPD trap is raised.

To support the Hypervisor in restoring the complete state of a non-resident VM, the operation of MTCR and MTDCR executed in the Hypervisor context will override hardware writeable fields (normally read only for the application) when the target CSFR belongs to either HRA or HRB.

Hypervisor state

The virtualization extension defines additional core registers for the control and configuration of the virtualization function. These CSFRs are exclusive to the hypervisor hardware resource and can only be accessed as CSFRs by software running in HRHV.

14.3.1 Virtualization control register 0 (VCON0)

The virtualization extension can be enabled or disabled by writing the bit VCON0.VEN. This bit is "write once" in that the value is locked following a write until the next reset, subsequent writes will be ignored and have no effect.

If VCON0.VEN==0 then HVCALL and all other methods of switching to the hypervisor are disabled.

VCON0		Address:	$B000_H$
Virtualization control register 0		Reset value:	Implementation Specific$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES														**NMI**	**EN**
-														rw	rw

Field	Bits	Type	Description
RES	31:2	-	Reserved

(table continues...)

(continued)

Field	Bits	Type	Description
NMI	1	rw	NMI handling 0_B - NMI handled by the currently running VM. 1_B - NMI handled by the hypervisor
EN	0	rw	Enable 0_B - Virtualization disabled 1_B - Virtualization enabled

14.3.2 Virtualization control register 1 (VCON1)

VCON1 contains the number of the currently running virtual machine. This status register is updated from VCON2.VMN on execution of the return from hypervisor (RFH) instruction or by a switch to the Hypervisor from some other virtual machine.

VCON1 Address: $B004_H$

Virtualization configuration register 1 Reset value: Implementation $specific_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES													**CVMN**		
-													rh		

Field	Bits	Type	Description
RES	31:3	-	Reserved
CVMN	2:0	rh	Currently running virtual machine number

14.3.3 Virtualization configuration register 2 (VCON2)

VCON2 is used by the return from hypervisor (RFH) instruction to change the execution from VM0 (hypervisor) to a target virtual machine selected by the hypervisor. It provides both the target virtual machine number and the protection register set to be used for the level 2 MPU. VCON2.VMN is also used by the CACHE[A/I].[I/W/WI] instructions to selectively operate on the data cache entries only encached by VM[VCON2.VMN].

VCON2 Address: $B008_H$

Virtualization configuration register 2 Reset value: Implementation $specific_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES					**L2_PRS**			**RES**					**VMN**		
-					rw			-					rw		

Field	Bits	Type	Description
RES	31:11	-	Reserved
L2_PRS	10:8	rw	Level 2 MPU PRS Selection of the protection register set for the level 2 MPU.
RES	7:3	-	Reserved
VMN	2:0	rw	VMN Target virtual machine number. Used by certain CACHE instructions and by the RFH instruction.

14.3.4 Base hypervisor cector table pointer (BHV)

The BHV contains the base address of the hypervisor trap vector table. When a hypervisor trap occurs, the entry address into the hypervisor trap vector table is generated from the hypervisor trap class number (HVTCN) of that trap, left-shifted by 5-bit and then ORd with the contents of the BHV register. The left-shift of the HVTCN results in a spacing of 8 words (32 bytes) between the individual entries in the vector table.

***Note**: This register is ENDINIT protected.*

BHV		Address:	B010$_H$
Base hypervisor vector table pointer		Reset value:	Implementation specific$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BHV															
rw															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BHV															**RES**
rw															-

Field	Bits	Type	Description
BHV	31:1	rw	Base address of the hypervisor trap vector table The address in the BHV register must be aligned to an even byte address (half-word address). Also, due to the simple ORing of the left-shifted trap identification number and the contents of the BHV register, the alignment of the base address of the vector table must be to a power of two boundary. There are eight different hypervisor trap classes, resulting in Trap Classes from 0 to 7. The contents of BHV should therefore be set to at least a 256-byte boundary (8 Trap Classes * 8-word spacing).
RES	0	-	Reserved

14.3.5 Virtual machine interrupt control register (VMn_ICR)

The VMn_ICR registers allow the CPU to evaluate whether an incoming interrupt request can be serviced by a virtual machine (irrespective of its current execution status).

VMn_ICR (n=0-7)		Address:	(B100+n*4)$_H$
Virtual machine n interrupt control register		Reset value:	00000000$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES								**PIPN**							
-								rh							

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IE	**RES**							**CCPN**							
rwh	-							rwh							

Field	**Bits**	**Type**	**Description**
RES	31:24	-	Reserved
PIPN	23:16	rh	Pending interrupt priority number A read-only bit-field that indicates the priority number of the pending service request for a specific VM. VMn_ICR.PIPN is set to 0 when no request is pending, and at the beginning of each new arbitration process. ... 00_H - No valid pending request. 01_H - Request pending, lowest priority. FF_H - Request pending, highest priority.
IE	15	rwh	Interrupt enable bit The interrupt enable bit enables this VM's service request system. VMn_ICR.IE is automatically updated by hardware on entry and exit of an Interrupt Service Routine (ISR). VMn_ICR.IE is cleared to 0 when an interrupt is taken, and is restored to the previous value when the ISR executes an RFE instruction to terminate itself. VMn_ICR.IE can also be updated through the execution of the ENABLE, DISABLE, RESTORE, MTCR, MTDCR and BISR instructions. 0_B - Interrupt system is disabled for this VM 1_B - Interrupt system is enabled for this VM
RES	14:8	-	Reserved
CCPN	7:0	rwh	Virtual machine n current priority number The Current CPU Priority Number (CCPN) bit-field indicates the current priority level of the Virtual Machine. For the currently running virtual machine, it is automatically updated by hardware on entry or exit of Interrupt Service Routines (ISRs) and through the execution of a BISR instruction. CCPN can also be updated through an MTCR instruction.

14.3.6 Virtual machine pre-emption threshold (VMn_PETHRESH)

VMn_PETHRESH can be used to control the interrupt pre-emption of each virtual machine. If the incoming PIPN is above the pre-defined threshold of the target virtual machine then the interrupt may lead to a VM switch (VMn_ICR.IE == 1 and PIPN > VMn_ICR.CCPN checks are also gating the switch)

VMn_PETHRESH (n=0-7)	Address:	$(B200+n*4)_H$
Virtual machine n pre-emption threshold	Reset value:	00000000_H

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES															
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES								**PE_THRESH**							
-								rw							

Field	Bits	Type	Description
RES	31:8	-	Reserved
PE_THRESH	7:0	rw	Pre-emption threshold

14.4 Virtual machine handling

The virtualization extension builds upon task model for low-overhead task switching and function calling that is fundamental to TriCore™ architecture . The extension expands this model to another level of abstraction allowing the hypervisor layer to manage virtual machines efficiently while providing application separation.

Related information

Tasks and functions on page 46

14.4.1 Hypervisor and virtual machine switching operation

The architecture switches between the hypervisor and the target virtual machine, or between a virtual machine and the hypervisor, when one of the events or instructions listed in Table 26 occurs. These events will force a switch of the hardware resource. When one of these events or instructions is encountered, either the upper context of the currently running virtual machine is saved, or the upper context of the target virtual machine is restored. The lower context is saved explicitly though instructions. In Table 26, 'Save' is a store through the Free CSA List Head Pointer register of the hypervisor hardware resource (HRHV_FCX) after the next value for the HRHV_FCX is read from the Link Word. 'Restore' is the converse of 'Save'.

Events that do not require a switch of the hardware resource are described in Task switching operation.

Table 26 Context related events and instructions

Event/Instruction	Context operation	Complement instruction	Context operation
Guest VM trap to hypervisor (trap delegated to hypervisor when guest VM is running)	Save upper	RFH - Return from hypervisor	Restore upper
Guest VM direct interrupt to hypervisor (interrupt for VM0 when the guest VM is running)	Save upper	RFH - Return from hypervisor	Restore upper
VM interrupt trap (HVINT) to hypervisor (interrupt for guest VMy when guest VM[x != y] is currently running)	Save upper	RFH - Return from hypervisor	Restore upper

Related information

Task switching operation on page 48

14.4.2 Context save areas (CSAs) and context lists

The upper and lower contexts are saved in context save areas (CSAs). Unused CSAs are linked together In the Free Context List (FCX). CSAs that contain saved upper or lower contexts are linked together in the Previous Context List (PCX).

To provide independence between the different running virtual machines contexts, the FCX is replicated per hardware resource (HR).

The virtualization extension only supports one running thread, meaning that only one virtual machine (0-7) is running at any given time. For this reason, there is only one previous context list register.

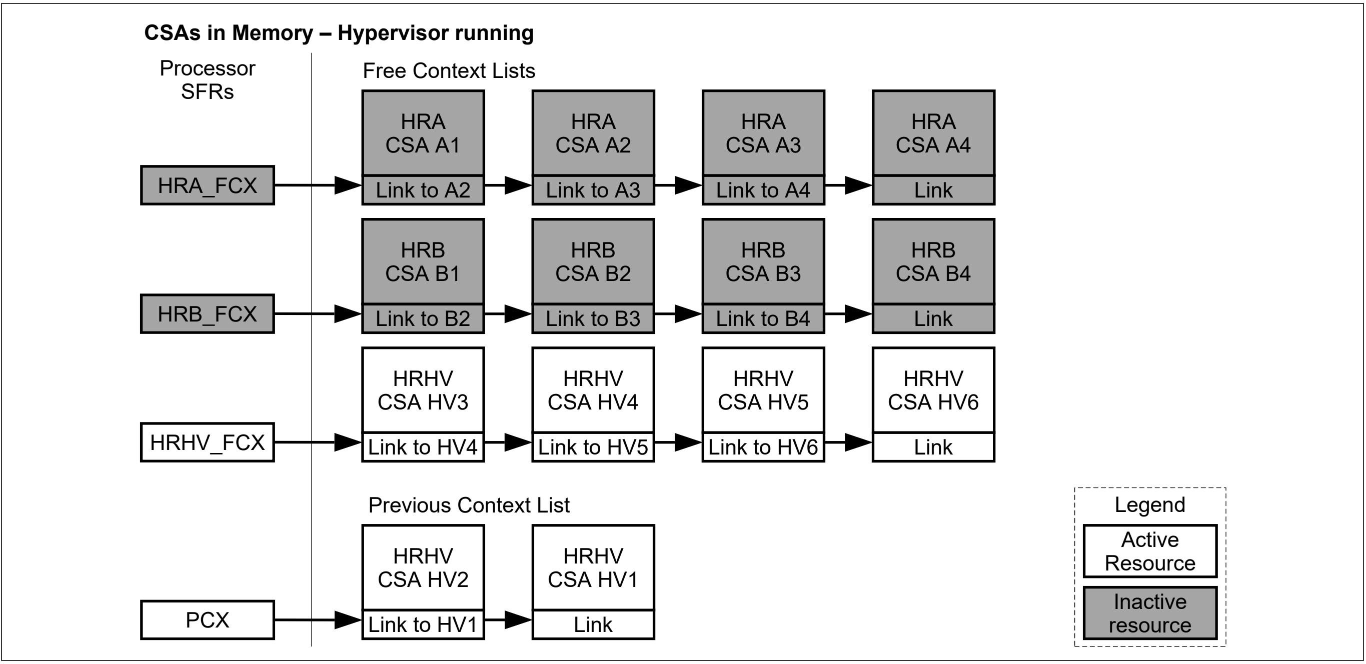


Figure 30 **CSAs in context lists**

14.4.3 Context switch with hypervisor interrupts and hypervisor traps

Hypervisor intervention is required in the following cases:

- When a guest VM traps to the hypervisor
- For direct interrupts to the hypervisor (when running a guest VM)
- For guest VM interrupts to a non-running guest VM

When one of these events occur and a guest VM is currently running, the processor saves the upper context of the currently running virtual machine in memory, using the hypervisor context save area accessed through HRHV_FCX. Execution of the current VM (and usage of HRA or HRB) then suspends and the CPU starts execution of the interrupt or trap handler; using the hypervisor state in the hardware resource HRHV.

When a guest VM exception is taken, the Stack Pointer is loaded with the current contents of the HRHV_ISP (Interrupt Stack Pointer). The PSW.IS bit is then set to one (1) to indicate execution from the interrupt stack.

To support interrupts to all virtual machines, the architecture maintains an ICR per virtual machine (VMn_ICR).

For hypervisor exceptions, the existing PCPN and PIE values in the PCXI of the currently running VM are saved in the CSA for the upper context, and the existing IE and CCPN values in the ICR are copied to the PCXI.PIE and PCXI.PCPN fields.

On an exception handled by the hypervisor, the upper context of the currently running virtual machine context is saved by hardware using the HRHV_FCX as an explicit part of the exception entry sequence. The hardware resource is switched to HRHV. Typical hypervisor intervention will lead to the switching of the guest VM. To preserve independence, other non-replicated state is required to be saved. The handler must execute BISR (Begin Interrupt Service Routine) or SVLCX (Save Lower Context) instruction to save the lower context registers that were not saved. The handler must also manage the PCXI, and select the target guest VM in VCON2.VMN.

Once the hypervisor has completed its intervention, the saved lower context is reloaded if necessary and execution of the interrupted virtual machine is resumed (RFH). The RFH will restore the context of the interrupted virtual machine and switch the hardware resource according to VCON2.VMN.

For small interrupt and trap handlers that can execute entirely within this set of registers saved on the exception, and that are only executing some operations on behalf of a guest VM, no further context saving is needed. The handler can execute immediately and return.

14.4.4 Considerations for starting a virtual machine

In order to switch execution context from the hypervisor, running in HRHV, to a virtual machine, running in either HRA or HRB, the hypervisor software is required to execute a return from hypervisor instruction (RFH). This instruction operates in a similar way to a return from exception (RFE) with the addition of switching hardware resources.

There are several conditions required for the RFH to complete successfully:

1. The previous context list (PCX) must point to a valid CSA location containing state related to targeted virtual machine
2. A[11] must point to the start PC of the target VM
3. The level 2 MPU must be configured to allow both access to the virtual machine code and data (level 2 MPU is active when HRHV is not running and cannot be disabled)
    a. HRHV MPU code and data registers must be configured
    b. VCON2.L2_PRS must be set to select the protection set allocated to the virtual machine
4. Response to NMI must be configured in VCON0.NMI
5. Proceed with the RFH instruction

14.4.5 Hypervisor context examples

This section provides examples of context save operations and of context restore operations related to interactions between virtual machines and hypervisor.

14.4.5.1 Transition from guest virtual machine to hypervisor

This example assumes:

- Hardware resource A is currently running (VM1)
- A switch to the hypervisor is requested

The following figure shows the free context lists for both HRA and HRHV as well as the previous context list. Virtual machine 1 is currently running and HRA is selected. The free context list (HRA_FCX) contains two free CSAs (A3 and A4), and the previous context list (PCX) contains two CSAs (A2 and A1) both relating the VM1 state. The free context list of the hypervisor contains three free CSAs (HV3, HV4 and HV5).

Prior to saving the context, the CPU switches from HRA to HRHV. When the context save operation is performed, the first CSA in the hypervisor free context list (CSA HV3) is pulled off and is placed on the front of the previous context list.

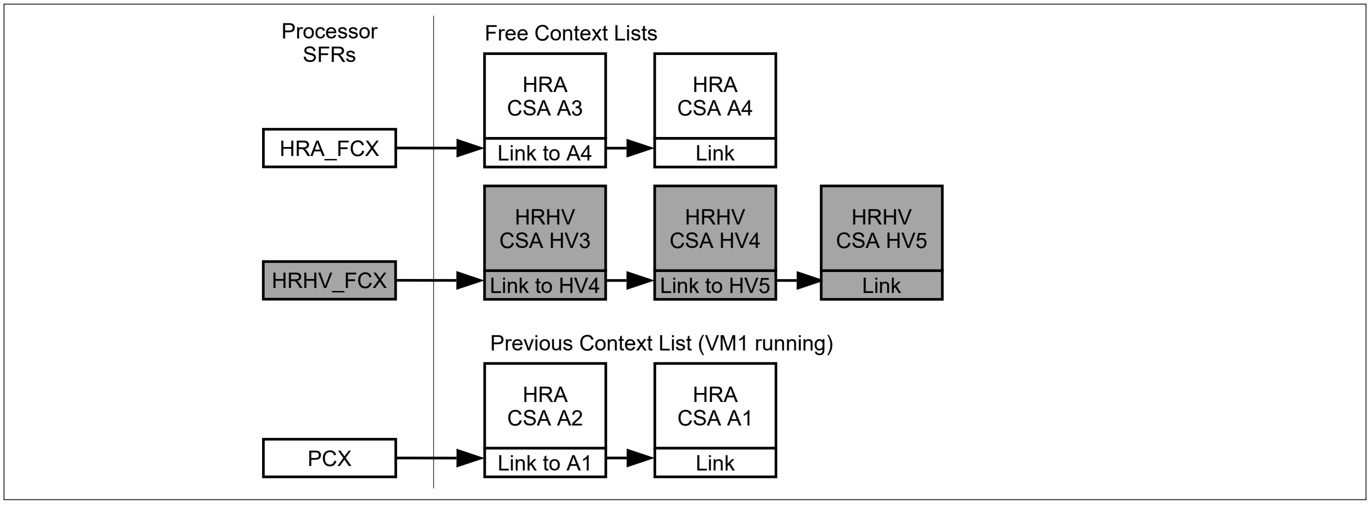


Figure 31 HRA (VM1) to HRHV (VM0) switch – Prior to context save

The following figure shows the steps taken during the context save operation. The numbers in the figure correspond to the steps listed after the figure.

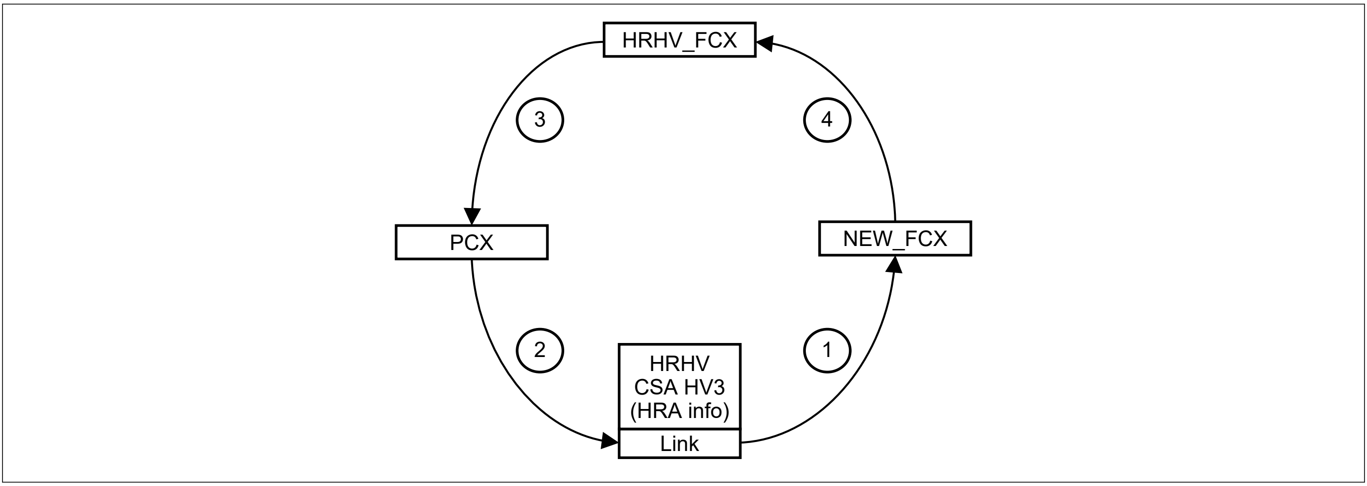


Figure 32 CSA and processor SFR updates on switch from HRA to HRHV

1. The contents of the Link Word in CSA HV3 are loaded into the NEW_FCX. The NEW_FCX now points to CSA HV4. The NEW_FCX is an internal buffer and is not accessible by the software

2. The contents of PCX are written into the Link Word CSA HV3. The Link Word of CSA HV3 now points to CSA A2

3. The contents of HRHV_FCX are written into the PCX. The PCX now points to CSA HV3, which is at the front of the Previous Context List

4. The NEW_FCX is loaded into the HRHV_FCX

The processor SFRs and CSAs look as shown in the following figure. The VM1 context to be saved in now written into the rest of CSA HV3. This CSA contains HRA state.

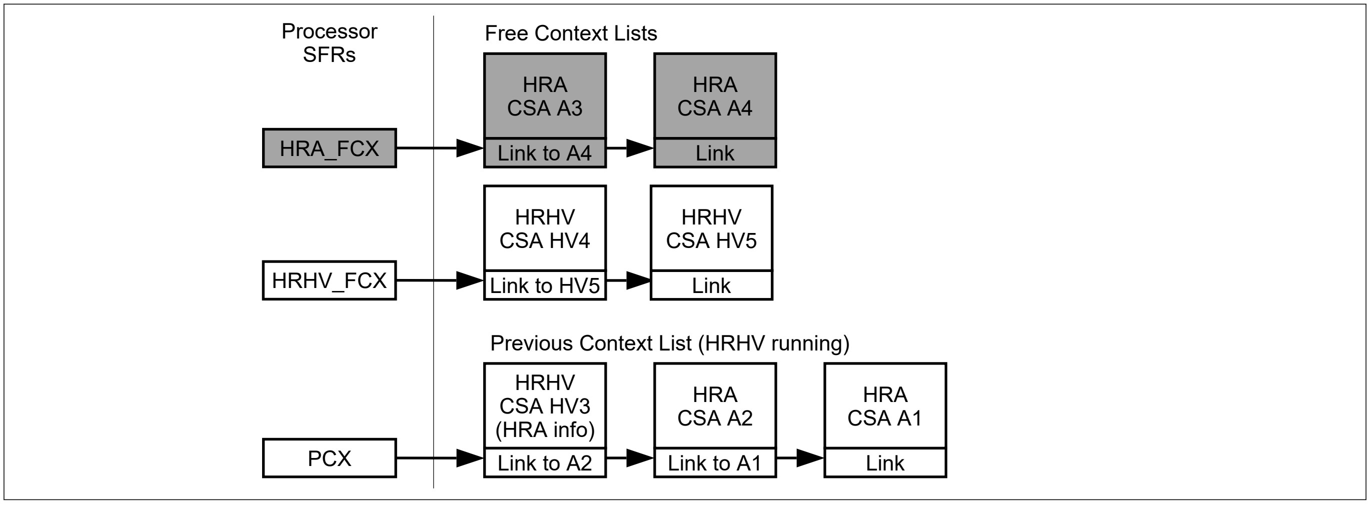


Figure 33 **HRA (VM1) to HRHV (VM0) switch – After the context save**

***Note**: Hypervisor routines that require more registers must execute the BISR (Begin Interrupt Service Routine) or SVLCX (Save Lower Context) instruction to save the lower context registers that were not saved as part of the VM switch.*

14.4.5.2 Transition from hypervisor to guest virtual machine

This example assumes:

- Hardware resource HV is currently running (hypervisor)
- A return to VM1 is requested (RFH instruction)
- PCX points to a CSA containing upper context information

The following figure shows the previous context list (PCX) with three CSAs (HV3, A2 and A1) and the free context lists for both HRA (containing two CSAs A3 and A4) and HRHV (containing two CSAs HV4 and HV5). The hypervisor is currently running and HRHV is selected.

The HRHV_FCX points to CSA HV4, the first available CSA in the free context list. PCX points to CSA HV3, the most recently saved CSA in the previous context list.

The Link Word CSA HV3 points to CSA A2; the Link Work of CSA A2 points to CSA A1; the Link word of CSA HV4 points to CSA HV5.

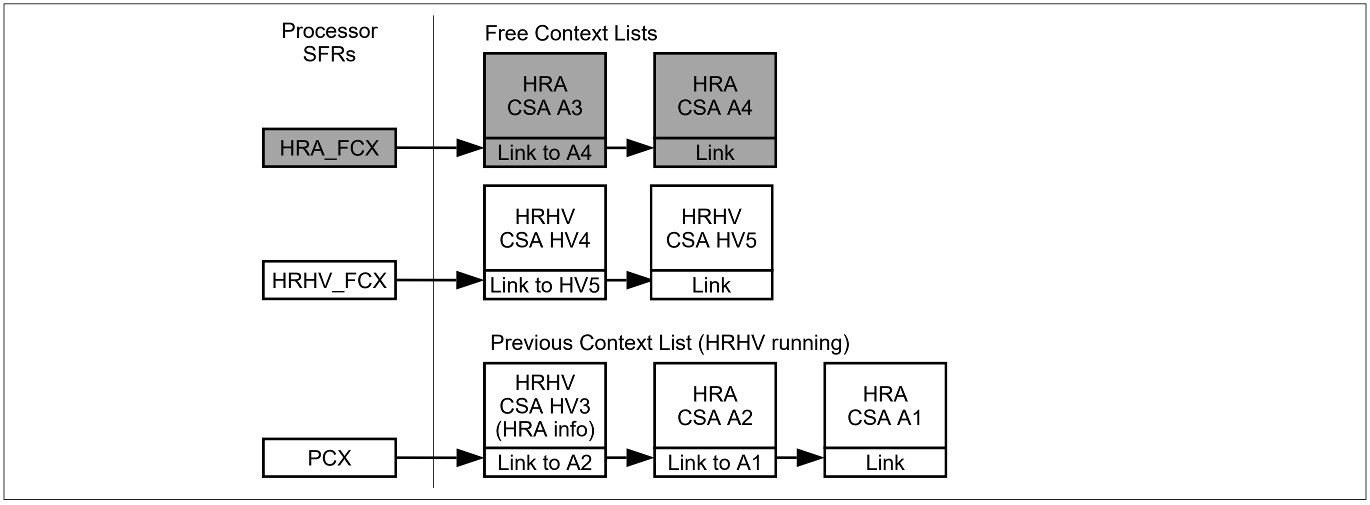


Figure 34 **HRHV (VM0) to HRA (VM1) switch – Prior to context restore**

When the context restore operation is performed, the fist CSA in the previous context list (CSA HV3) is pulled off and is placed on the front of the free context list.

The following figure shows the steps taken during the context restore operation. The numbers in the figure correspond to the following steps.

1. The contents of the Link Word in CSA HV3 are loaded into the NEW_PCX. The NEW_PCX now points to CSA A2. The NEW_PCX is an internal buffer and is not accessible by the software

2. The contents of the HRHV_FCX are written into the Link Word of CSA HV3. The Link Word of CSA HV3 now points to CSA HV4

3. The contents of the PCX are written into the HRHV_FCX. The HVRV_FCX now points to CSA HV3, which is at the front of the free context list

4. The NEW_PCX is loaded into the PCX

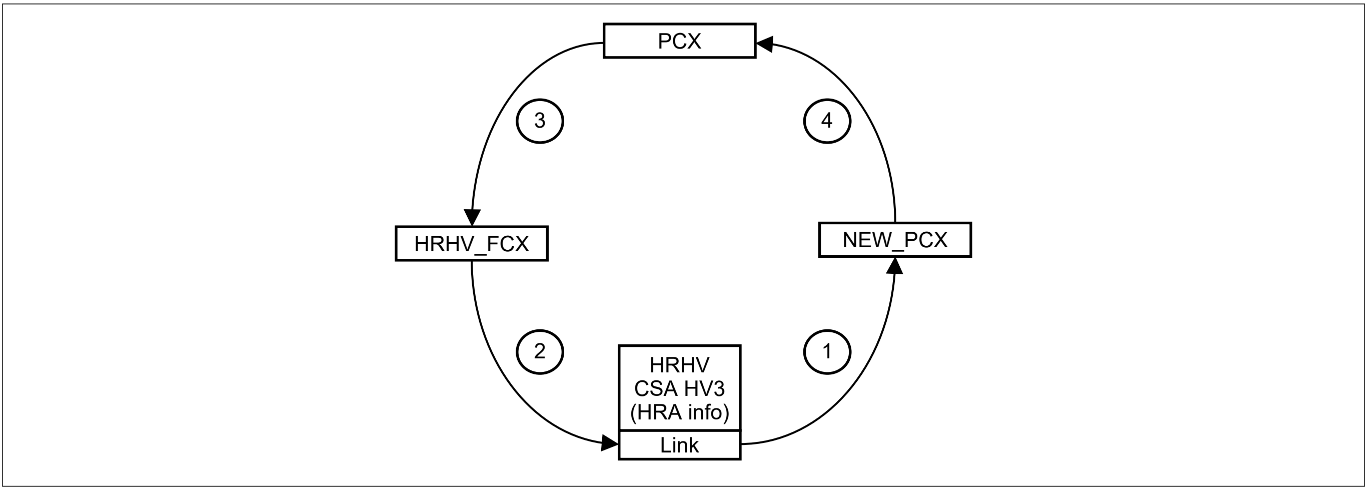


Figure 35 **CSA and processor SFR updates on switch from HRHV to HRA**

Once the context is restored, the hardware resources are switch from HRHV to HRA. The processor SFRs and CSAs now look as shown in the following figure. The restored context is then written into the upper context registers.

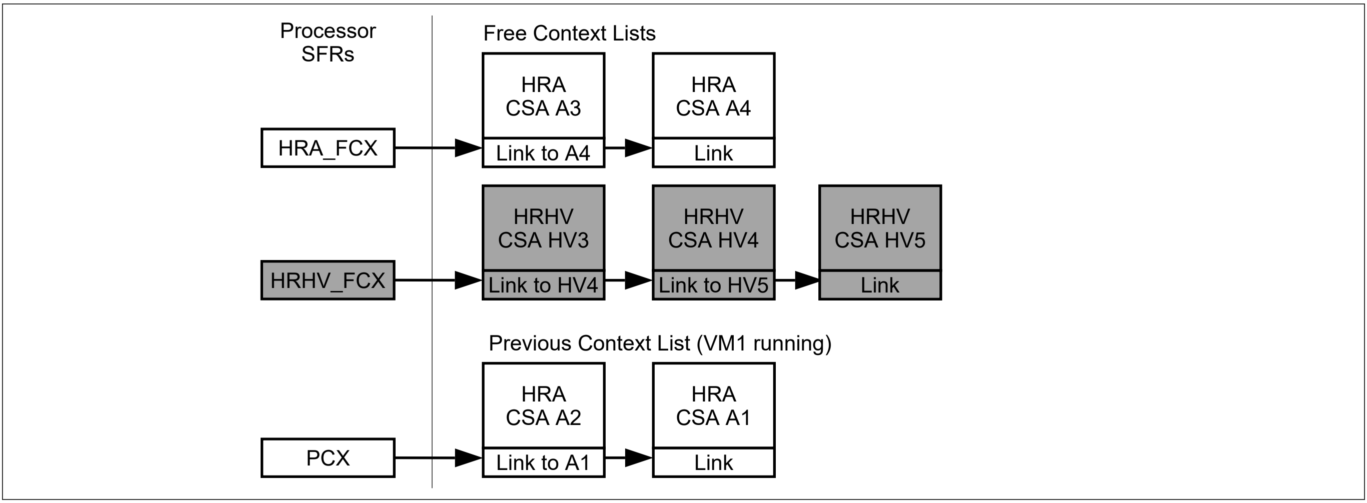


Figure 36 **HRHV (VM0) to HRA (VM1) switch – After the context save**

14.4.5.3 Switching execution from guest virtual machine in HRA to guest virtual machine in HRB

This example assumes:

- Hardware resource A is currently running (VM1)
- A switch to the hypervisor is requested
- The hypervisor decides to switch to hardware resource B (VM2)
- VM2 is already resident in HRB

The following figure shows the free context lists for HRA, HRB and HRHV as well as the previous context list. Virtual machine 1 is currently running and HRA is selected. The free context list (HRA_FCX) contains two free CSAs (A3 and A4), and the previous context list (PCX) contains two CSAs (A2 and A1) both relating to the state of VM1. The free context list of HRB(VM2) contains three free CSAs (B2, B3 and B4). The free context list of the hypervisor contains three free CSAs (HV3, HV4 and HV5).

Prior to saving the context, the CPU switches from HRA to HRHV. When the context save operation is performed, the first CSA in the hypervisor free context list (CSA HV3) is pulled off and is placed on the front of the previous context list.

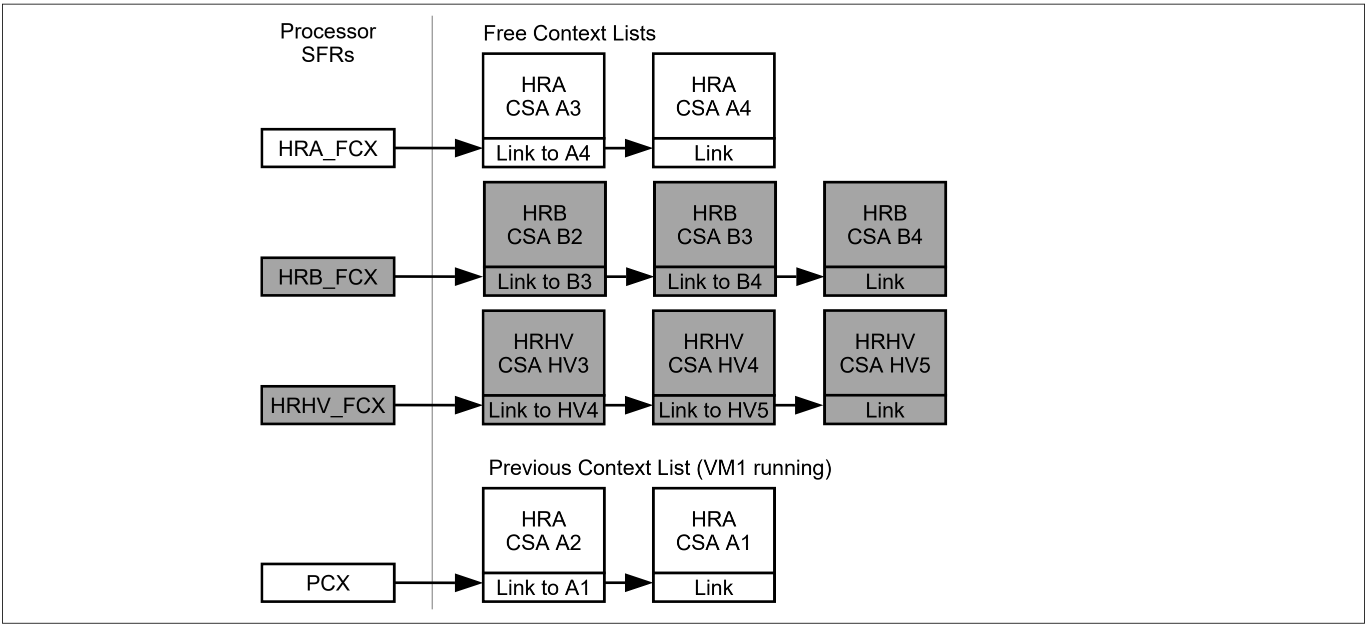


Figure 37 **HRA (VM1) to HRHV (VM0) switch – Prior to context save**

The processor SFR's and CSAs look as shown in the following figure. The VM1 context to be saved in now written into the rest of CSA HV3. This CSA contains HRA state.

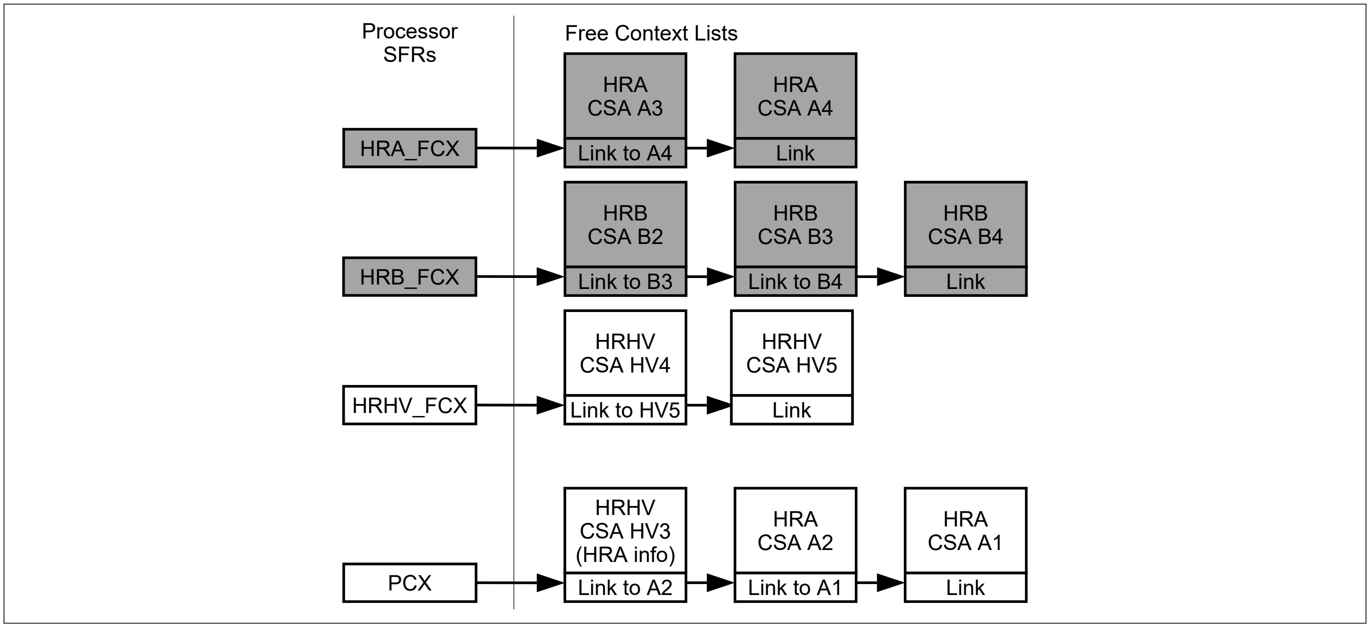


Figure 38 **HRA (VM1) to HRHV (VM0) switch – After the context save**

In order to ensure isolation between the state of the different VMs, the hypervisor software is required to:

- Maintain separate PCX for the VMs already started
- Maintain the lower context registers when switching or swapping VMs

The following figure shows the SFR's and CSAs as well as the necessary steps to ensure isolation.

1. PCX points to CSAs containing HRA state and cannot be used to restore HRB. The contents of PCX are first saved to a variable under the control of the hypervisor software
2. PCX is loaded with a previously saved value that points to a CSA containing HRB state

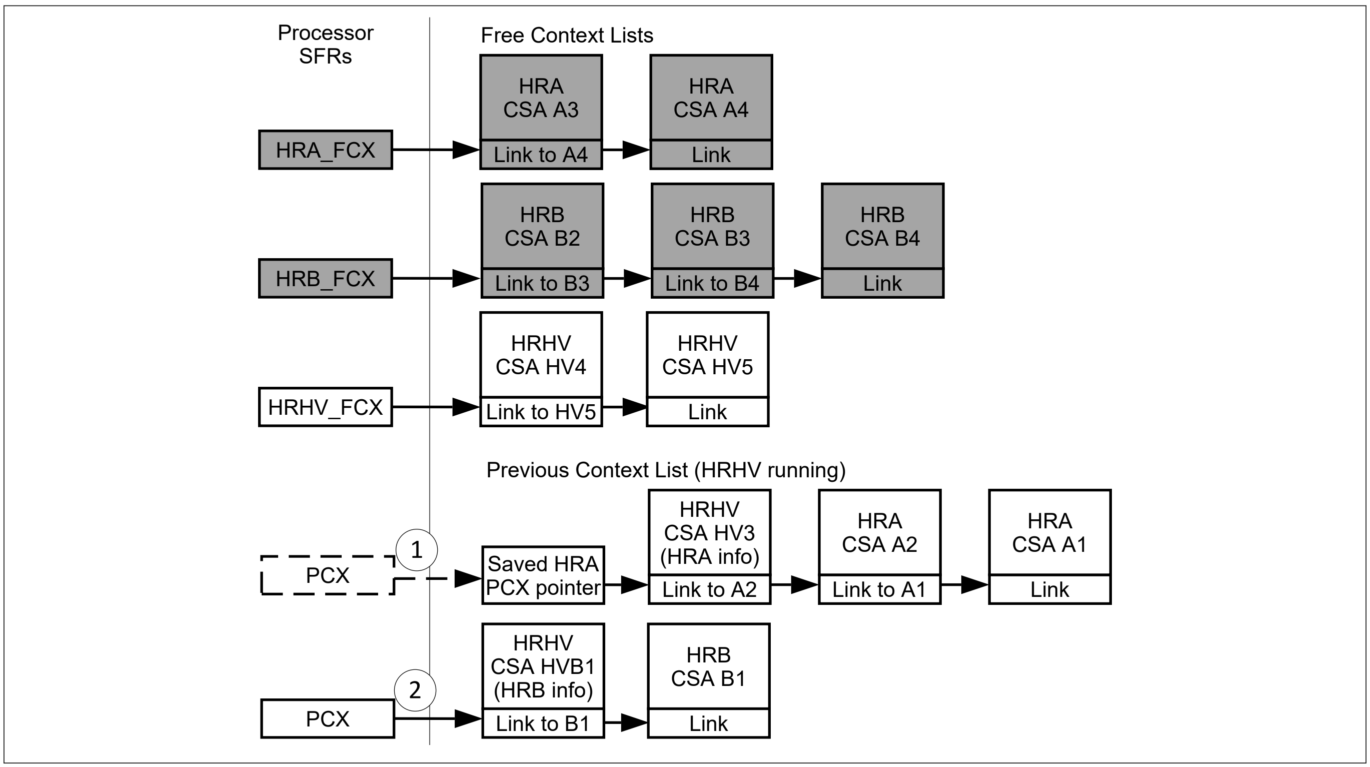


Figure 39 **HRHV (VM0) – PCX handling**

The hypervisor software can now update VCON2.VMN to select VM2, restore the lower context and proceed with the execution of the return from hypervisor (RFH) instruction.

1. The contents of the Link Word in CSA HVB1 are loaded into the NEW_PCX. The NEW_PCX now points to CSA B1. The NEW_PCX is an internal buffer and is not accessible by the software
2. The contents of the HRHV_FCX are written into the Link Word of CSA HVB1. The Link Word of CSA HVB1 now points to CSA HV4
3. The contents of the PCX are written into the HRHV_FCX. The HVRV_FCX now points to CSA HVB1, which is at the front of the free context list
4. The NEW_PCX is loaded into the PCX

Once the upper context is restored, the hardware resources are switch from HRHV to HRB. The processor SFRs and CSAs now look as shown in Figure Figure 40. The restored context is then written into the upper context registers.

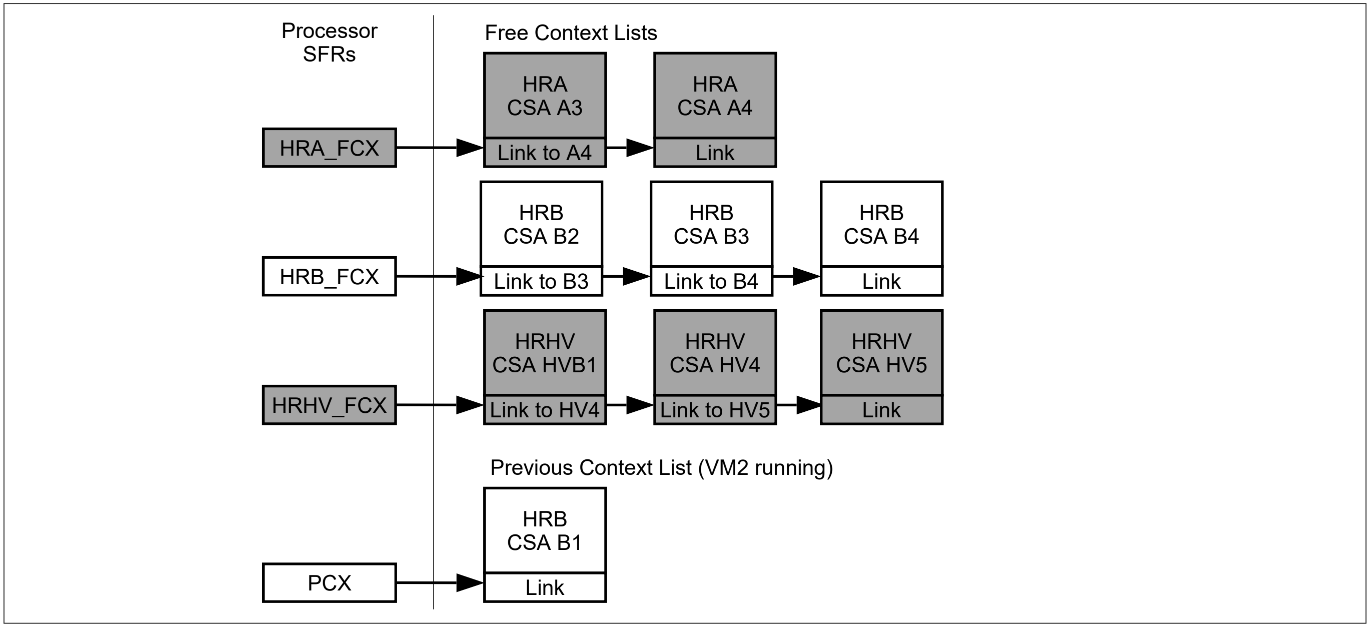


Figure 40 **HRHV (VM0) to HRB (VM2) switch – After the context restore**

Hardware resource B is now selected and virtual machine 2 is now running.

14.4.5.4 Swapping execution from guest virtual machine x in HRB to guest virtual machine y in HRB

This example assumes:

- Hardware resource B is currently running (VM2)
- A switch to the hypervisor is requested
- The hypervisor software swaps VM2 and VM3 within hardware resource B
- VM3 had already been running previously and is currently saved in memory

The following figure shows the free context lists for HRB and HRHV as well as the previous context list. Virtual machine 2 is currently running and HRB is selected. The free context list (HRB_FCX) contains two free CSAs (B3 and B4), and the previous context list (PCX) contains two CSAs (B2 and B1) both relating to the sated of VM2. The free context list of the hypervisor contains three free CSAs (HV3, HV4 and HV5).

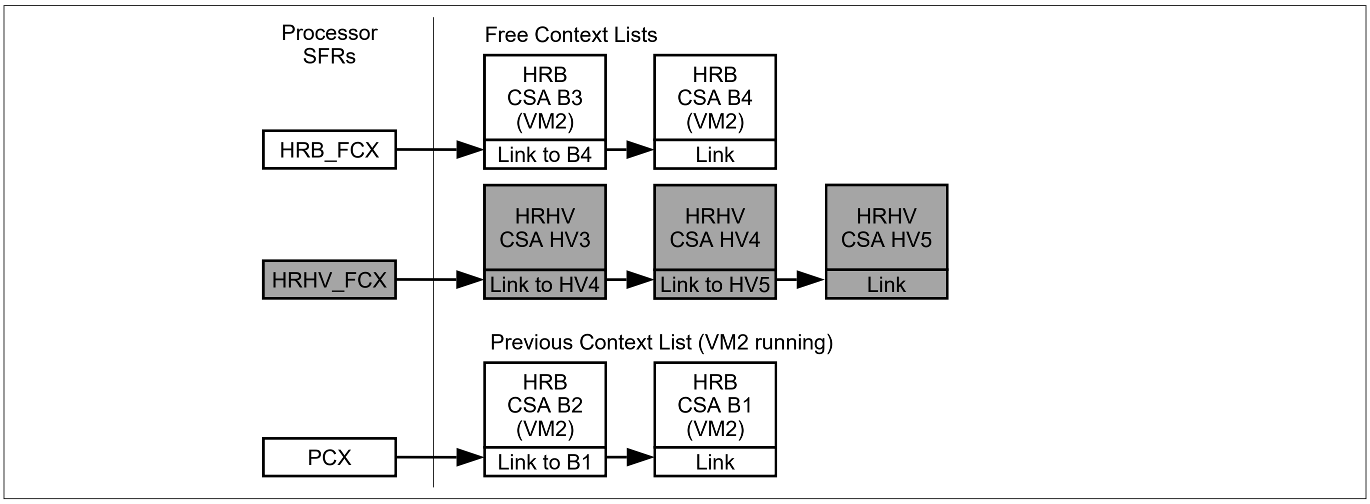


Figure 41 **HRB (VM2) to HRHV (VM0) switch – Prior to context save**

The processor SFR's and CSAs look as shown in the following figure. The VM2 context to be saved in now written into the rest of CSA HV3. This CSA contains HRB (VM2) state.

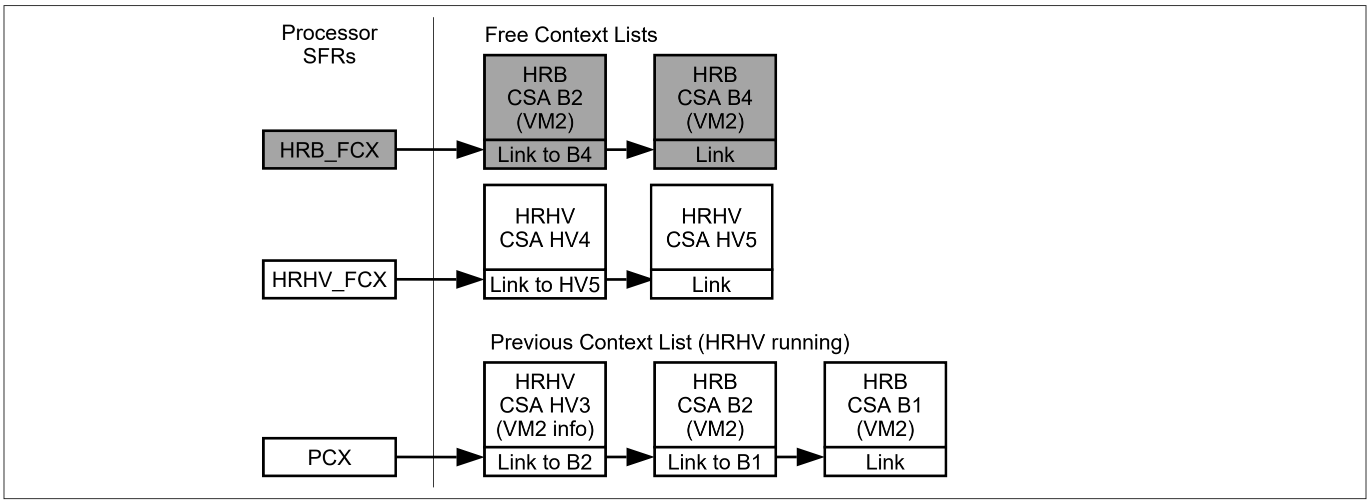


Figure 42 **HRB (VM2) to HRHV (VM0) switch – After the context save**

In order to ensure isolation between the state of the different VMs, the hypervisor software is required to:

- Maintain separate PCX for the VMs already started
- Maintain the lower context registers when switching or swapping VMs
- Maintain HRB MPU state

The following figure shows the SFR's and CSAs as well as the necessary steps to ensure isolation.

1. PCX points to CSAs containing VM2 state and cannot be used to restore VM3. The contents of PCX are first saved to a variable under the control of the hypervisor software

2. PCX is loaded with a previously saved value that points to a CSA containing VM3 state

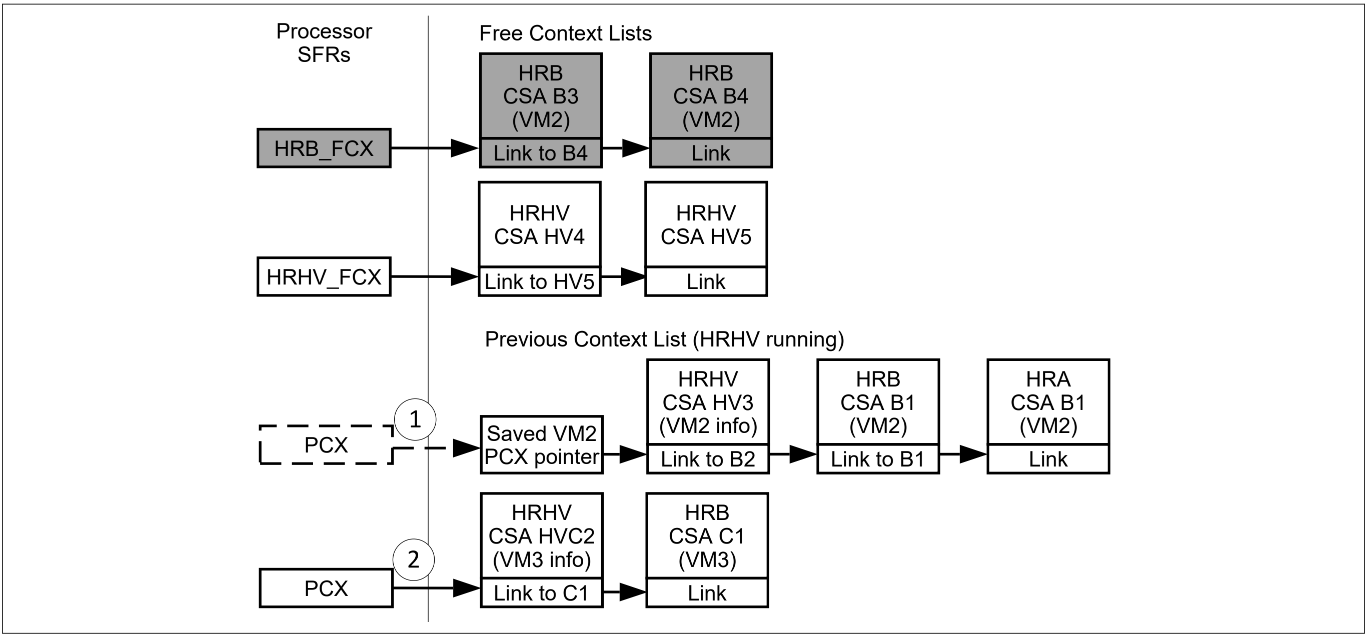


Figure 43 **HRHV(VM0) - PCX handling - Virtual machine swap**

The hypervisor software can now update VCON2.VMN to select VM3, restore the lower context and proceed with the execution of the return from hypervisor (RFH) instruction.

1. The contents of the Link Word in CSA HVC2 are loaded into the NEW_PCX. The NEW_PCX now points to CSA C1. The NEW_PCX is and internal buffer and is not accessible by the software
2. The contents of the HRHV_FCX are written into the Link Word of CSA HVC2. The Link Word of CSA HVC2 now points to CSA HV4
3. The contents of the PCX are written into the HRHV_FCX. The HVRV_FCX now points to CSA HVC2, which is at the front of the free context list
4. The NEW_PCX is loaded into the PCX

Once the upper context is restored, the hardware resources are switch from HRHV to HRB. The processor SFRs and CSAs now look as shown in the following figure. The restored context is then written into the upper context registers.

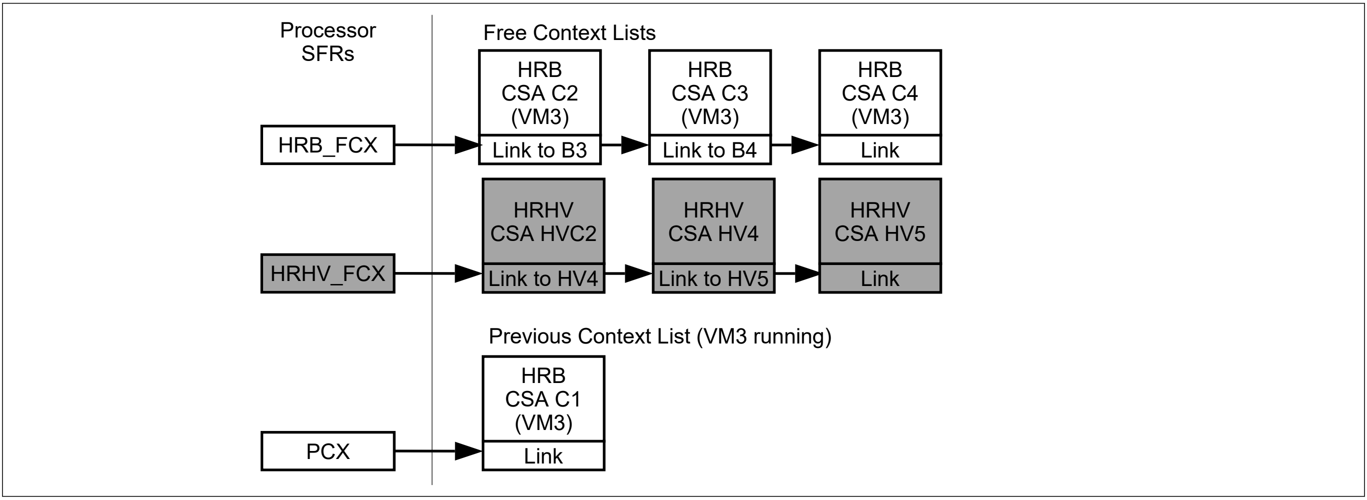


Figure 44 **HRHV (VM0) to HRB (VM3) switch – After the context restore**

Guest virtual machine 3 is now hosted in Hardware resource B and is running.

14.5 Disabling virtualization

For compatibility with earlier systems, where the use of virtualization extension is not required, TC1.8 can be configured to have virtualization disabled after start-up.

When virtualization is not required, the application software may disable the virtualization functionality by writing VCON0.VEN = 0 and then perform a RFH to VM1. Execution of a RFH prior to the first VCON0.VEN write is an illegal operation from an architectural viewpoint leading to undefined results.

***Note**: Virtualization disabling will be effective after the RFH is executed.*

When VCON0.VEN == 0 an RFH will perform a switch to VM1 with level 2 MPU protection disabled regardless of the values of VCON2.VMN and L2_PRS. Virtualization is only considered to be disabled once the RFH has completed. The disabling of the virtualization feature requires the RFH instruction to be executed without exceptions being triggered. A valid PCXI pointing to a correctly initialized CSA entry is a prerequisite for the instruction to complete properly.

External accesses targeting the HRB and HRHV CSFR ranges, when virtualization is disabled or not present will result in an error. All CSFRs not contained in HRA, for example VMn_INT, become undefined and should not be accessed.

14.6 Virtualization: Interrupt system

In a system supporting virtualization in a TriCore™ CPU, the IR will be enhanced to allow interrupts to target specific VMs on that CPU.

On receipt of an interrupt request from an interrupt source the SRPN is used by the IR to prioritize between multiple concurrent interrupt requests for a VM on that CPU. The SRPN of the winning request per VM is supplied to the CPU as a Pending Interrupt Priority Number (PIPN).

The CPU determines whether to accept a requested interrupt for a VM by comparing the PIPN against not only the VM's Current CPU Priority Number (CCPN), but additional state in HRHV (described later).

When the CPU determines the targeted VM can take the requested interrupt, if the interrupt is for the currently running VM it is directly taken and the CPU signals the IR that it has been accepted so that the IR can clear that request for the VM.

If the CPU determines the targeted VM can take the requested interrupt, and it is not the currently running VM, then it will initiate a switch to the hypervisor either as a trap to manage the targeted VM to run and accept the interrupt, or as a cross-VM interrupt to the hypervisor (VM0).

If a trap is required to manage the execution of the targeted VM, the interrupt is only taken when the VM has started execution. At this point the IR is notified that the interrupt has been accepted, allowing the IR to clear the request for the VM.

If the interrupt is for VM0, the CPU signals the IR that it has been accepted so that the IR can clear that request for VM0.

Interrupt control registers

Each virtual machine has a separate ICR (VMn_ICR) register and consequently distinct PIPN, IE, and CCPN values. Therefore, the interrupt state of each VM is independent of the other VMs.

Interrupt vector table

Interrupt Service Routines are associated with interrupts at a particular priority by way of the Interrupt Vector Table. The Interrupt Vector Table is an array of Interrupt Service Routine (ISR) entry points. The Interrupt Vector Table is stored in code memory.

To support independent functionality, each hardware resource is provided with its own base interrupt vector pointer HRx_BIV.

Interrupt pre-emption

A running guest virtual machine may be pre-empted on receipt of an interrupt to another VM only where the incoming PIPN is above a pre-defined threshold value for the target virtual machine.

Each virtual machine has an 8-bit pre-emption threshold register (VMn_PE_THRESH). An incoming interrupt to VMn when VMn is not currently running is taken if the PIPN is greater than the threshold value.

The hypervisor may be pre-empted only if hypervisor interrupts are enabled (VM0_ICR.IE == 1). This allows the hypervisor to execute uninterrupted by other VMs irrespective of their threshold values if its interrupts are disabled. The hypervisor can only be pre-empted when it chooses to re-enable interrupts, by a higher priority interrupt for the hypervisor itself. The hypervisor cannot be pre-empted by an interrupt for a guest virtual machine.

An incoming interrupt to VMn will be taken if:

```
if (VCON1.CVMN == n) then {
    // Interrupt for the currently running VM
    (VMn_ICR.PIPN > VMn_ICR.CCPN) AND VMn_ICR.IE
}
else if (n == 0) then {
    // Interrupt for hypervisor from guest VM
    (VM0_ICR.PIPN > VM0_PE_THRESH) AND (VM0_ICR.PIPN > VM0_ICR.CCPN) AND VM0_ICR.IE
}
else if (VCON1.CVMN != 0) then {
    // Interrupt for other guest VM from a guest VM
    (VMn_ICR.PIPN > VMn_PE_THRESH) AND (VMn_ICR.PIPN > VMn_ICR.CCPN) AND VMn_ICR.IE
}
else {
    // Interrupt for other guest VM from Hypervisor
    // Hypervisor cannot be pre-empted
    // Software intervention required
}
```

If the pre-emption threshold of a VM is 255 then no pre-emption is possible and only cooperative, or time sliced scheduling of that VM by the hypervisor will occur.

If the pre-emption threshold of a VM is 0 then all interrupts to that machine would have the potential to cause pre-emption and provide the best real-time behavior.

The VMn_PE_THRESH registers are only accessible by the hypervisor. In all cases of pre-emption by an interrupt to another VM, a VM switch or swap will be performed by the hypervisor.

Interrupts to VMn where VMn is currently running do not use threshold comparison and so require the interrupt to satisfy the condition to only satisfy the standard interrupt acceptance conditions, that is the VM's ICR.IE is 1 and the VM's ICR.CCPN < PIPN.

CPU operation on an interrupt request

Several conditions could block the CPU from immediately responding to the interrupt request presented to it. These are:

- The target VM interrupts are disabled (VMn_ICR.IE == 0)
- The target VM threshold is not met (VMn_ICR.PIPN <= VMn_PE_THRESH)
- The VM priority (VMn_ICR.CCPN), is equal to or higher than the Pending Interrupt Priority Number (PIPN) for the target VMn
- The CPU is in the process of entering an interrupt or trap service routine
- The CPU is operating on non-interruptible trap services
- The CPU is executing a multi-cycle instruction

- The CPU is executing an instruction which modifies any of the ICR registers
- The hypervisor is currently running (VCON1.CVMN==0) and the interrupt request is for a guest VM

The CPU responds to the interrupt request only when these conditions are no longer true.

Related information

Interrupt system on page 58

14.6.1 Interrupt use cases

The PIPN and VMN are presented to the CPU and compared against the CCPN and IE value for that machine in the standard way. In addition a pre-emption check will be performed if the requested VM is not currently running.

Interrupt for the currently running VM

The CPU checks the state of the interrupt enable bit VMn_ICR.IE, and compares the current guest VM priority number VMn_ICR.CCPN against the PIPN. The guest VM can be interrupted only if VMn_ICR.IE == 1 and PIPN is greater than VMn_ICR.CCPN. If this is true the VM can enter the service routine. The PIPN is used to determine the interrupt vector table entry point using the current HRHV_BIV (VM0), HRA_BIV (VM1) or HRB_BIV(VM2-7).

Guest VM direct interrupt to the hypervisor

The CPU checks the state of the hypervisor interrupt enable bit VM0_ICR.IE, compares the hypervisor priority number VM0_ICR.CCPN against the PIPN, and checks that the PIPN is greater than the hypervisor pre-emption threshold VM0_PETHRESH. The hypervisor can only be interrupted if VM0_ICR.IE == 1, the PIPN is greater than VM0_ICR.CCPN and the PIPN is greater than VM0_PETHRESH. If this is true the CPU will transition to HR0 and enter the service routine. The PIPN is used to determine the interrupt vector table entry point using HRHV_BIV.

Guest VMy interrupt while running guest VMx (x != y)

The CPU checks the state of the guest VMy interrupt enable bit VMy_ICR.IE, compares the guest VMy priority number VMy_ICR.CCPN against the PIPN, and compares the PIPN against VMy pre-emption threshold VMy_PETHRESH. The guest VMy can only be interrupted if VMy_ICR.IE == 1, the PIPN is greater than VMy_ICR.CCPN and the PIPN is greater than VMy_PETHRESH. If this is true the CPU will transition to HRHV (HVINT) to allow the hypervisor software to switch or swap virtual machines. If the guest VMy is resident in the alternate guest HR then a guest VM switch is sufficient. If the guest VMy is not resident in the alternate HR then the hypervisor software should swap the contents of the HR to memory and restore VMy. The guest VM switch is completed with the execution of an RFH (with VCON2.VMN == y). The guest VMy can enter the service routine. The PIPN is used to determine the interrupt vector table entry point using either HRA_BIV (interrupts for VM1) or HRB_BIV (interrupts for VM2-7).

***Note**:* *If the hypervisor software decides to return to a different guest VM, or if the content of VMy_ICR is updated by the hypervisor software will cause the interrupt arbitration to be restarted.*

Guest VMx interrupt while running the hypervisor

The hypervisor cannot be pre-empted by interrupts targeting a guest VM. It is the responsibility of the hypervisor software to monitor the VMx_ICR registers for pending interrupts if pre-emption to guest VM is required.

***Note**:* *Alternatively,*

14.6.2 Hypervisor interrupt entry sequence

HR switch if HRHV not currently active

- The currently executing VM is set to 0, VCON1.CVMN is set to 0
- Switch the global address registers to use HRHV set
- Switch the current CSFR set to the HRHV set

Once in the HRHV context

- The upper context of the current VM is saved to the CSA location defined by HRHV_FCX
- The return address in A[11] is updated with the current PC
- D15 is updated
  - If the interrupt entry required a switch to HRHV, D15[31] is set
  - If the interrupt is for a non-running VM, D15[2:0] captures the target VM
  - If the interrupt entry did not require a switch to HRHV, D15[31:0] is cleared
- The A[10] stack pointer is set to the interrupt stack pointer (HRHV_ISP). The stack pointer bit is then set for using the interrupt stack: PSW.IS = 1
- The current IO mode is set to Supervisor mode, which means all permissions are enabled: PSW.IO = 10_B
- The current protection register set is set to 0
- The call depth counting is enabled and the counter (CDC) is cleared, and the call depth limit is set for 64: PSW.CDC = 0000000_B
- PSW safety bit is set to value defined in the HRHV_CORECON register: PSW.S = HRHV_CORECON.IS
- Write permission to global registers A[0], A[1], A[8], A[9] is disabled: PSW.GW = 0
- PSW user status bits are set to 0: PSW.USB = 00000000_B
- The current HRHV_ICR.CCPN and HRHV_ICR.IE are saved into the PCXI
- The PXCI.UL bit is set to 1
- The interrupt system is disabled for this CPU: HRHV_ICR.IE = 0
- The ISR is then fetched from either BHV in the case of an interrupt for a different VM, or directly from BIV, the interrupt for the hypervisor

An hypervisor interrupt service routine is entered with the interrupt system disabled for this CPU and the current CPU priority (CCPN) set to the priority (PIPN) of the interrupt being serviced. It is up to the hypervisor to enable the interrupt system again and optionally modify the priority number CCPN to implement interrupt priority levels or handle special cases.

An virtual machine interrupt service routine is entered with the interrupt system disabled for the virtual machine only. Interrupts for the hypervisor or for non-running VM's can still be taken if the PIPN is either than the pre-emption threshold and PIPN of the target VM.

The interrupt system can be enabled with the ENABLE instruction. ENABLE sets ICR.IE = 1 (interrupt system enabled) for the currently running VM or hypervisor. The BISR (Begin Interrupt Service Routine) instruction also enables the interrupt system, sets the ICR.CCPN to a new value, and saves the lower context of the interrupted task. The interrupt enable bit (ICR.IE) and current CPU priority number (ICR.CCPN) of the currently running VM or hypervisor can also be modified with the MTCR (Move To Core Register) or MTDCR (Move To Dual Core Register) instructions. The hypervisor may also configure the VMn_ICR and pre-emption thresholds (VMn_PE_THRESH) for all the VMs

The ENABLE, BISR, and DISABLE (disable interrupts) instructions are all executed such that the CPU is blocked from taking interrupt requests until the instruction is completely finished. This avoids pipeline side effects and eliminates the need for an ISYNC (synchronize instruction stream) following these instructions. MTCR and MTDCR are exceptions and must be followed by an ISYNC instruction.

14.6.3 Exiting an interrupt service routine (ISR)

When an ISR exits with an RFE (return from exception) instruction the execution will continue in current VM even if another VM was running prior to the interruption.

When an ISR exits with an RFH (return from hypervisor) instruction the execution will return to the VM pointed to by VCON2.VMN.

14.6.4 Interrupts for the hypervisor

Interrupts for the hypervisor are always taken using HRHV_BIV as a base vector. When the hypervisor is already running (HRHV resource is selected), it behaves like a standard interrupt sequence. When the hypervisor in not currently running, a switch to HRHV is required before the hardware can execute the required interrupt service routine.

To ensure that the same ISR can be used in both instances, to allow software to determine whether an RFE or RFH should be used to terminate the service routine, bit D15[31] is set whenever the entry requires a switch to the hypervisor. A simple conditional jump can be used to select the operation required to complete the trap handler. Bits D15[30:0] are cleared by the interrupt entry sequence.

```
    // return from ISR
    JZ.T D15, 31, rfe_target
    RFH
rfe_target:
    RFE
```

14.7 Hypervisor trap system

A hypervisor trap occurs because of an event that cannot be handled in the currently running guest virtual machine such as an operation exception requiring hypervisor intervention, a level-two memory protection exception, or because of an interrupt presented for a non-running VM. This section describes the different hypervisor traps that can occur and the TriCore™ architecture hypervisor trap handling mechanism.

Related information

Trap system on page 66

14.7.1 Hypervisor trap types

The TriCore™ virtualization feature specifies eight general classes for hypervisor traps (three are reserved). Each class has its own hypervisor trap handler, accessed through a trap vector of 32 bytes per entry, indexed by the hardware-defined hypervisor trap class number. Within each class, specific traps are distinguished by a Hypervisor Trap Identification Number (HVTIN) that is loaded by hardware into register D[15] before the first instruction of the hypervisor trap handler is executed. The hypervisor trap handler must test and branch on the value in D[15] to reach the sub-handler for a specific HVTIN.

Hypervisor traps can be further classified as synchronous or asynchronous, and as hardware or software generated. These are explained after the following table which lists the hypervisor trap classes, summarizing and classifying the pre-defined set of specific traps within each class.

In the following table: HVTIN = Hypervisor trap identification Number, Synch. = Synchronous, Asynch. = Asynchronous, HW = Hardware, SW = Software.

Table 27 Supported hypervisor traps

HVTIN	Name	Synch. / Asynch.	HW / SW	Definition
Class 0 — Hypervisor call [1]				
	HVCALL	Synch.	SW	Hypervisor Call
Class 1 — Hypervisor Interrupt Trap [2]				
	HVINT	Asynch.	HW	Trap to HV for guest Interrupt
Class 2 — Level 2 data memory protection trap				
0	L2MPR	Synch./ Asynch.[3]	HW	Level 2 Memory Protection Read
1	L2MPW	Synch./ Asynch.[3]	HW	Level 2 Memory Protection Write
Class 3 — Level 2 code memory protection trap				
0	L2MPX	Synch.	HW	Level 2 Memory Protection Execute
Class 4 — HV CSFR Access support [4]				
	HVCSFR	Synch.	SW	HV CSFR Access Support
Class 5 — Reserved				
Class 6 — Reserved				
Class 7 — Reserved				

1) For the hypervisor call trap, the HVTIN is taken from the immediate constant specified in the HVCALL instruction. The range of values that can be specified is 0 to 255, inclusive
2) For hypervisor interrupt trap, the HVTIN indicates the VMN targeted by the pending interrupt request
3) The trap is always synchronous to the event that triggered it. In the case of a guest VM CSA check, the event may be asynchronous (for example interrupt for guest VM, NMI taken in guest VM, or CAE)
4) The HVTIN will capture the 16-bit register offset used in the MTCR/MFCR/MTDCR/MFDCR instruction. Bit 31 is set to 1 in case of MTCR/MTDCR, and to 0 otherwise

14.7.2 Hypervisor trap handling

The actions taken on hypervisor traps by the hypervisor trap handling mechanisms are similar from those taken on traps with the exception of the change in execution environment for the guest VM in HRA or HRB to HRHV.

Hypervisor trap vector format

The trap handler vectors are stored in code memory in the trap vector table. The BHV register specifies the base address of the hypervisor trap vector table. The vectors are made up of a number of short code blocks, evenly spaced by eight words.
If a trap handler is very short it may fit entirely within the eight words available in the vector code block. If it does not fit the vector code block then it should contain some initial instructions, followed by a jump to the rest of the handler.

Accessing the hypervisor trap vector table

When a hypervisor trap occurs, a trap identifier is generated by hardware. The trap identifier has two components:

• The Hypervisor trap class number (HVTCN) used to index into the trap vector table

• The Hypervisor trap identification number (HVTIN) which is loaded into the data register D[15]

The Hypervisor trap class number is left shifted by five bits and ORd with the address in the BHV register to generate the entry address of the hypervisor trap handler.

Return address (RA)

The return address for the guest VM is saved in the return address register A[11].

For a synchronous trap, the return address is the PC of the instruction that caused the trap with the exception of the HVCALL trap. On a HVCALL trap, triggered by the HVCALL instruction, the return address points to the instruction immediately following HVCALL.

For an asynchronous trap, the return address is that of the instruction that would have been executed next, if the asynchronous trap had not been taken.

Hypervisor trap vector table

The entry-points of all hypervisor trap service routines are stored in memory in the hypervisor trap vector table. The BHV register specifies the base address of the hypervisor trap vector table in memory. The hypervisor trap vector Table can be located in any available code memory. The BHV register can be modified using the MTCR instruction during the initialization phase of the system, (the BHV register is ENDINIT protected and under the control of the hypervisor). This arrangement makes it possible to have multiple trap vector tables and switch between them by changing the contents of the BHV register.

When a hypervisor trap event occurs, a trap identifier is generated by the hardware detecting the event. The trap identifier is made up of a Hypervisor trap class number (HVTCN) and a Hypervisor trap identification number (HVTIN).

The HVTCN is left-shifted by five bits and ORd with the address in the BHV register to form the entry address of the TSR. Because of this operation, it is recommended that bits [7:5] of register BHV are set to 0 (see the following figure). Note that bit 0 of the BHV register is always 0 and can not be written to (instructions have to be aligned on even byte boundaries).

Left-shifting the HVTCN by 5 bits creates entries into the hypervisor trap vector table which are evenly spaced 8 words apart. If a trap handler (TSR) is very short, it may fit entirely within the eight words available in the hypervisor trap vector table entry. Otherwise, the code at the entry point must ultimately cause a jump to the rest of the TSR residing elsewhere in memory.

Unlike the interrupt vector table, entries in the hypervisor trap vector table cannot be spanned.

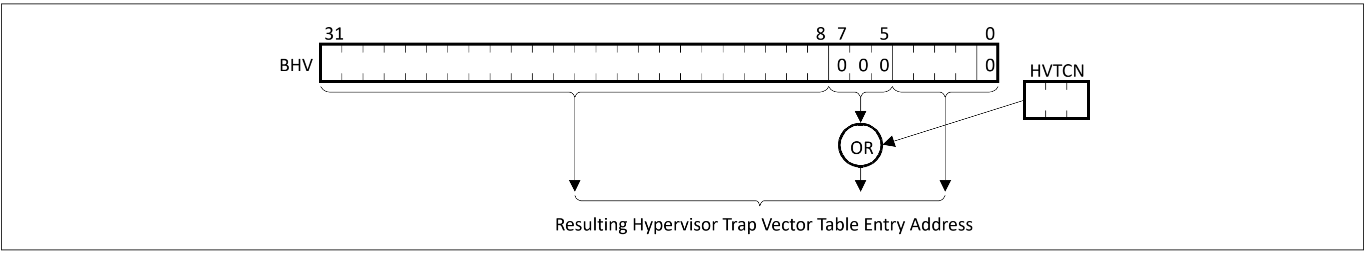


Figure 45 **Hypervisor trap vector table entry address calculation**

Entering an hypervisor trap handler

- The executing VM is set to 0 (VCON1.CVMN is set to 0)
- The global address registers are switched to use the HRHV instances
- The CSFR set is switched to the HRHV instances
- The upper context of the guest VM is saved to the CSA location defined by HRHV_FCX
- The return address in A[11] is updated
- The HVTIN is loaded into D[15]
- The stack pointer in A[10] is set to the Interrupt Stack Pointer (HRHV_ISP). The stack pointer bit is set for using the interrupt stack: PSW.IS = 1
- The current IO mode is set to Supervisor mode

- The current protection register set is set to 0
- The call depth is cleared, and the call depth limit is set for 64: PSW.CDC = 0000000_B
- Call depth counter is enabled, PSW.CDE = 1
- PSW safety bit is set to value defined in the HRHV_CORECON register. PSW.S = HRHV_CORECON.TS
- Write permission to global registers A[0], A[1], A[8], A[9] is disabled: PSW.GW = 0
- PSW user status bits are set to 0: PSW.USB = 00000000_B
- The interrupt system is disabled for this CPU: ICR.IE = 0. The 'old' HRHV_ICR.IE and HRHV_ICR.CCPN are saved in PCXI.PIE and PCXI.PCPN respectively. HRHV_ICR.CCPN remains unchanged
- The hypervisor trap vector table is accessed to fetch the first instruction of the hypervisor trap handler

Although hypervisor traps leave the ICR.CCPN unchanged, their handlers still begin execution with interrupts disabled. They can therefore perform critical initial operations without interruptions, unless they specifically re-enable interrupts.

When the guest VM upper context cannot be saved while attempting to take an hypervisor trap, the hypervisor trap is converted to a non-recoverable FCU trap taken within the hypervisor context using HRHV_BTV.

14.7.3 Hypervisor trap descriptions

The following sub-sections describe the hypervisor trap classes and specific traps listed in hypervisor trap types.

Hypervisor call

HVCALL - Hypervisor call (HVTIN = 8-bit unsigned immediate constant in HVCALL)

The HVCALL trap is raised immediately after the execution of the HVCALL instruction, to initiate a call requiring hypervisor support. The TIN that is loaded into D[15] when the trap is taken is specified as an 8-bit unsigned immediate constant in the HVCALL instruction. The return address points to the instruction immediately following the HVCALL.

Hypervisor interrupt trap

HVINT – Trap to HV for guest Interrupt (HVTIN = target VM)

The HVINT trap is raised when an interrupt request for a non-running guest VM has cleared all the conditions to be taken. The target VM number is loaded into D[15] to allow the hypervisor trap handler to update hardware resource state if required and then switch to the relevant hardware resource where the interrupt will be taken.

Level 2 data memory protection

L2MPR – Level 2 data memory protection read (TIN 0)

The L2MPR trap is generated by the level 2 memory protection system (HRHV MPU resources are used as level 2 MPU to a guest VM) when the guest VM's effective address of a load, or atomic read and write sequence instruction or the load part of a context operation does not lie within any range with read permissions enabled.

L2MPW – Level 2 data memory protection write (TIN 1)

The L2MPW trap is generated by the level 2 memory protection system (HRHV MPU resources are used as level 2 MPU to a guest VM) when the guest VM's effective address of a store, or atomic read and write sequence instruction or the store part of a context operation software, does not lie within any range with write permissions enabled.

Level 2 code memory protection

L2MPX – Level 2 code memory protection execute (TIN 0)

The L2MPX trap is generated by the level 2 memory protection system (HRHV MPU resources are sued as level 2 MPU to a guest VM) when the guest VM's PC does not lie within any range with execute permissions enabled.

HV CSFR access support

HVCSFR – Access to CSFR requiring hypervisor support
The HVCSFR trap is generated when a guest VM executes a MTCR/MFCR/MTDCR/MFDCR targeting a core register under the control of the hypervisor. The list of registers affected is implementation specific.

14.7.4 Exception priorities (virtualization present and enabled)

The priority order between an asynchronous trap, a synchronous trap, and an interrupt from the software architecture model, is as follows:

1. Asynchronous trap except HVINT (highest priority)
2. Synchronous trap and hypervisor traps
3. Hypervisor interrupt (HVINT)
4. Interrupt (lowest priority)

The following trap rules must also be considered:

1. The older the instruction in the instruction sequence which caused the trap, the higher the priority of the trap. All potential traps with lower priorities are void
2. Attempting to save a context with an empty free context list (FCX = 0) results in a FCU (Free Context List Underflow) trap. This trap takes priority over all other exceptions
3. When the same instruction causes several synchronous traps anywhere in the pipeline, priorities follow those shown in the table below

Table 28 Synchronous Trap and Hypervisor Trap Priorities

Priority	Type of trap or hypervisor trap	Switch to HRHV context
Instruction fetch traps		
1	Breakpoint trap or halt - BBM (Trigger on PC)	No
2	VAF-P[1)]	No
3	VAP-P[1)]	No
4	MPX	No
5	L2MPX	Yes
6	PSE	No
7	PIE	No
Instruction format traps		
8	IOPC	No
9	OPD (instruction operands excluding CSFR checks)	No
10	UOPC	No
Instruction traps		
11	Breakpoint trap or halt - BBM (trigger on address, MxCR, MxDCR, debug)	No
12	PRIV	No
13	GRWP	No
14	SYS	No

(table continues...)

Table 28 (continued) Synchronous Trap and Hypervisor Trap Priorities

Priority	Type of trap or hypervisor trap	Switch to HRHV context
15	HVCALL	Yes
16	HVCSFR	Yes
17	OPD (CSFR checks)	No
18	CSE	No
Context traps		
19	FCD	No
20	FCU	No
21	CSU	No
22	CDO	No
23	CDU	No
24	NEST	No
25	CTYP	No
Data Memory access traps		
26	MEM	No
27	ALN	No
28	MPN	No
29	VAF-D[1)]	No
30	VAP-D[1)]	No
31	MPP	No
32	MPR	No
33	MPW	No
34	L2MPR	Yes
35	L2MPW	Yes
36	DSE	No
General data traps		
37	SOVF	No
38	OVF	No
39	Breakpoint trap or halt - BAM	No

1) Only applicable if an MMU is present and enabled.

Table 29 **Asynchronous trap priorities**

Priority	Asynchronous traps	Switch to HRHV context
1	NMI	Depending on VCON.NMI
2	DAE	Yes
3	CAE	No
4	TAE	No
5	DIE	Yes

Related information

Exception priorities on page 76

14.8 Virtualization changes to trap system

The majority of traps are taken directly in the context of the virtual machine that raised them. These traps are not taken in the context of the guest VM that they arise in. If they arise in the context of a guest VM they result in a hardware resource transition and are taken as traps in VM0 using the HRHV BTV. These asynchronous traps may also arise directly when VM0 is executing. To allow the hypervisor software to determine when a hardware resource transition occurred the high order bit of D[15] (containing the TIN) is set. If after processing of the exception, the event allows continuation of the guest VM or own task, an RFE or RFH can then be simply selected based on this bit. Note that trap handlers in guest VMs will only receive TINs with values as specified in the supported traps table (see Trap types) and will never see the altered values provided to the hypervisor's normal trap handlers. A simple conditional jump can be used to select the operation required to complete the trap handler.

```
    // return from handler
    JZ.T D15, 31, rfe_target
    RFH
rfe_target:
    RFE
```

DAE - Data access asynchronous error (class 4, TIN 3)

DAE are unconditionally taken in the hypervisor context. When the hypervisor is not currently running, a DAE request triggers a switch to the hypervisor followed by a jump to the appropriate handler using HRHV_BTV.

DIE - Data memory integrity error (class 4, TIN 6)

DIE are unconditionally taken in the hypervisor context. When the hypervisor is not currently running, a DIE request triggers a switch to the hypervisor followed by a jump to the appropriate handler using HRHV_BTV.

NMI - Non-maskable interrupt (class 7)

The handling of NMI traps in a virtual system is controlled by VCON0.NNI.

- VCON0.NMI == 0, NMI handled by the currently executing VM0-7
- VCON0.NMI == 1, NMI handled by the hypervisor, Switch to hypervisor if not currently running

14.9 HRHV memory protection unit

The TriCore™ memory protection system provides the RTOS control over which regions of memory a task is allowed access, and what types of access are permitted.

- This MPU, which is always present in the architecture regardless of whether the virtualization feature is present or enabled, and is termed the level 1 MPU

Separately, the hypervisor requires the same control over which regions of memory a virtual machine is allowed access, and what types of access are permitted

- This additional MPU is termed level 2 MPU of the guest VM

When the virtualization feature is present and enabled all guest virtual machines (VM1-VM7), memory accesses must pass both level 1 and level 2 protections checks. If a level 1 protection check fails, a trap will be taken in the RTOS which is identical to what would occur if the virtualization feature was not present.

If a level 2 protection check fails a guest trap to hypervisor is taken in the hypervisor.

***Note**: When the hypervisor (VM0) is running, only the HRHV MPU resource is available and applied to check accesses. In this the HRHV MPU acts as a level 1 MPU and a violation explicitly results in a classical MPX/MPR/MPW trap in VM0*

Protection checks

Virtualization enabled

Virtualization is enabled when VCON0.VEN == 1.

- When HRHV is active and memory protection enabled (HRHV_CORECON.PROTEN == 1) then memory accesses must pass protection as defined by the PSW memory protection set. An access that fails a memory protection check will generate a trap in the HRHV context
- When HRA is active and memory protection enabled (HRA_CORECON.PROTEN == 1) then memory accesses must pass protection as defined by the HRA memory protection set. An access that fails the HRA memory protection check will generate a trap in the HRA context
- When HRB is active and memory protection enabled (HRB_CORECON.PROTEN == 1) then memory accesses must pass protection as defined by the HRB memory protection set. An access that fails the HRB memory protection check will generate a trap in the HRB context
- Level 1 protection checks are implicitly treated as passed if HRx_CORECON.PROTEN is off
- When any virtual machine (VM1-VM7) is active then all accesses must pass level 2 memory protection as defined by the selected HRHV memory protection set (VCON2.L2_PRS). An access that fails the HRHV memory protection check will generate a hypervisor trap in the HRHV context
- An access that fails both level 1 and level 2 protection will generate a trap in the level 1 context

Virtualization disabled

Only level 1 MPU is active.

Context save area protection

Memory accesses generated by CSA save and restore operations do not undergo level one protection checks since the CSA pointers are under the control of the RTOS. To ensure the virtual machine is fully encapsulated, CSA save and restore operations generated by a VM(1:7) are checked by the level 2 MPU.

CSA save and restore operations generated by the hypervisor do not undergo protection checks.

Swapping configuration

When virtual machines require to be swapped in/out of HRB, the management of the level 1 protection registers is covered by the HRB state swap. L2_PRS provides for seven unique selections of level 2 MPU resources, but some systems may require more dynamic configuration of level 2 state and selection.

Related information

Memory protection system on page 114

Crossing protection boundaries on page 116

14.9.1 Using the range based level 2 memory protection system

When the level 2 protection system is enabled, every virtual machine memory access (read, write or execute) is checked for legality before the access is permitted. The permission is determined by all of the following:

- The level 2 protection is always enabled when virtualization is present and enabled and a virtual machine (VM1-VM7) is running
- The currently selected L2 protection register set (VCON2.L2_PRS)
- The address ranges selected in the respective protection set fetch enable, read enable and write enable configuration registers

Enabling the level 2 MPU

The level 2 MPU is only active when virtualization is present and enabled and a guest virtual machine (VM1-VM7) is active.

Level 2 protection set selection

The level 2 protection register set used for a running VM is selected by VCON2.L2_PRS.

Address range

Guest VM instruction fetch addresses are checked against the currently selected code address ranges (in HRHV_CPXE_x where x is VCON2.L2_PRS).

Guest VM data read addresses are checked against the currently selected data read address ranges (in HRHV_DPRE_x where x is VCON2.L2_PRS).

Guest VM data write addresses are checked against the currently selected data write address ranges (in HRHV_DPWE_x where x is VCON2.L2_PRS).

For an access to be permitted, there must be an address range that covers the address being accessed.

The level 2 memory protection system does not differentiate between access permission levels. The data and code protection settings have the same effect, whether the permission level is currently set to Supervisor, User-1 or User-0 mode.

For guest VM instruction fetches, the PC value for the fetch is checked against the execute enabled selected level 2 code protection ranges of the selected protection set. When a PC lies within any execute enabled range the access is permitted. When a PC does not lie any of the level 2 execute enabled ranges, then an L2MPX (Level 2 Memory Protection Execute) trap is taken.

For guest VM load operations, the data address values are checked against the read enabled selected level 2 data protection ranges of the selected protection set. When an address lies within any read enabled range the access is permitted. When an address does not lie within any of the level 2 read enabled ranges, then an L2MPR (Level 2 Memory Protection Read) trap is taken.

For guest VM store operations, the data address values are checked against the write enabled selected level 2 data protection ranges of the selected protection set. When an address lies within any write enabled range the access is permitted. When an address does not lie within any of the level 2 write enabled ranges, then an L2MPW (Level 2 Memory Protection Write) trap is taken.

For guest VM load and store operations (that is those that have an atomic load and store sequence) operations, the data address values are checked against both the read and write enabled selected level 2 data protection ranges of the selected protection set. When an address lies within any read enabled range and any write enabled range (it need not be the same range pair) the access is permitted. When an address does not lie both within any of the read enabled ranges and any of the write enabled range then it is not permitted. The access will take an L2MPR trap if there is no read enabled range, and will take an L2MPW trap if there is an enabled read range but no write enabled range.

For guest VM save and restore of contexts to the context save area (as distinct from loads and stores using LDLCX,LDUCX, STLCX, STUCX), the data address values are checked against both the read and write enabled selected level 2 data protection ranges of the selected protection set. When an address lies within any read enabled range and any write enabled range (it need not be the same range pair) the access is permitted. When an address does not lie both within any of the read enabled ranges and any of the write enabled range then it is not permitted. The access will take an L2MPR trap if there is no read enabled range, and will take an L2MPW trap if there is an enabled read range but no write enabled range.

Guest VM Supervisor mode does not disable level 2 memory protection.

For a complete description of traps, see Hypervisor trap system.

Protection registers

Each hardware resource has a replicated set of memory protection registers. Their number is implementation specific. The hypervisor specific set (used for level 2 protection) are located in the HRHV resource.

15 Core register table

The following tables list all the TriCore™ CSFRs and GPRs. The memory protection system is modular and the actual number of registers is implementation-specific.

Table 30 General Purpose Register (GPR)

Register name	Description	Address offset
D[0]	Data Register 0	$FF00_H$[1]
D[1]	Data Register 1	$FF04_H$
D[2]	Data Register 2	$FF08_H$
D[3]	Data Register 3	$FF0C_H$
D[4]	Data Register 4	$FF10_H$
D[5]	Data Register 5	$FF14_H$
D[6]	Data Register 6	$FF18_H$
D[7]	Data Register 7	$FF1C_H$
D[8]	Data Register 8	$FF20_H$
D[9]	Data Register 9	$FF24_H$
D[10]	Data Register 10	$FF28_H$
D[11]	Data Register 11	$FF2C_H$
D[12]	Data Register 12	$FF30_H$
D[13]	Data Register 13	$FF34_H$
D[14]	Data Register 14	$FF38_H$
D[15]	Data Register 15 - Implicit Data Register	$FF3C_H$
A[0]	Address Register 0 - Global Address Register	$FF80_H$[2]
A[1]	Address Register 1 - Global Address Register	$FF84_H$
A[2]	Address Register 2	$FF88_H$
A[3]	Address Register 3	$FF8C_H$
A[4]	Address Register 4	$FF90_H$
A[5]	Address Register 5	$FF94_H$
A[6]	Address Register 6	$FF98_H$
A[7]	Address Register 7	$FF9C_H$
A[8]	Address Register 8 - Global Address Register	$FFA0_H$
A[9]	Address Register 9 - Global Address Register	$FFA4_H$
A[10] (SP)	Address Register 10 - Stack Pointer Register	$FFA8_H$
A[11] (RA)	Address Register 11 - Return Address Register	$FFAC_H$
A[12]	Address Register 12	$FFB0_H$
A[13]	Address Register 13	$FFB4_H$
A[14]	Address Register 14	$FFB8_H$
A[15]	Address Register 15 - Implicit Address Register	$FFBC_H$

1) These address offsets are not used by the MTCR/MFCR/MTDCR/MFDCR instructions
2) These address offsets are not used by the MTCR/MFCR/MTDCR/MFDCR instructions in the standard application but may be used by the hypervisor to target the replicated version of the global address registers of HRA/HRB

Table 31 Core Special Function Registers (CSFR)

Register name	Description	Address offset
PCXI PCX	Previous Context Information Register Previous Context Pointer Register	$FE00_H$
PSW	Program Status Word Register	$FE04_H$
PC	Program Counter Register	$FE08_H$
CORECON[1]	System Configuration Register	$FE14_H$
CPU_ID	CPU Identification Register (read only)	$FE18_H$
CORE_ID	Core Identification Register	$FE1C_H$
BIV[2]	Base Address of Interrupt Vector Table Register	$FE20_H$
BTV[2]	Base Address of Trap Vector Table Register	$FE24_H$
ISP[2]	Interrupt Stack Pointer Register	$FE28_H$
ICR	Interrupt Control Register	$FE2C_H$
PPRS	Previous PRS	$FE34_H$
FCX	Free Context List Head Pointer Register	$FE38_H$
LCX	Free Context List Limit Pointer Register	$FE3C_H$
BOOTCON	Boot Configuration Register	$FE60_H$
CCON	Clock Control Register	$FE68_H$
TCCON	TriCore™ hardware Configuration Register	$FE6C_H$
COMPAT[2] [1]	Compatibility Mode Register	9400_H
Memory Protection Registers		
DPR0_L	Data Protection Range 0, Lower	$C000_H$
DPR0_U	Data Protection Range 0, Upper	$C004_H$
DPR1_L	Data Protection Range 1, Lower	$C008_H$
DPR1_U	Data Protection Range 1, Upper	$C00C_H$
DPR2_L	Data Protection Range 2, Lower	$C010_H$
DPR2_U	Data Protection Range 2, Upper	$C014_H$
DPR3_L	Data Protection Range 3, Lower	$C018_H$
DPR3_U	Data Protection Range 3, Upper	$C01C_H$
DPR4_L	Data Protection Range 4, Lower	$C020_H$
DPR4_U	Data Protection Range 4, Upper	$C024_H$
DPR5_L	Data Protection Range 5, Lower	$C028_H$
DPR5_U	Data Protection Range 5, Upper	$C02C_H$
DPR6_L	Data Protection Range 6, Lower	$C030_H$
DPR6_U	Data Protection Range 6, Upper	$C034_H$
DPR7_L	Data Protection Range 7, Lower	$C038_H$
DPR7_U	Data Protection Range 7, Upper	$C03C_H$

(table continues...)

Table 31 (continued) Core Special Function Registers (CSFR)

Register name	Description	Address offset
DPR8_L	Data Protection Range 8, Lower	$C040_H$
DPR8_U	Data Protection Range 8, Upper	$C044_H$
DPR9_L	Data Protection Range 9, Lower	$C048_H$
DPR9_U	Data Protection Range 9, Upper	$C04C_H$
DPR10_L	Data Protection Range 10, Lower	$C050_H$
DPR10_U	Data Protection Range 10, Upper	$C054_H$
DPR11_L	Data Protection Range 11, Lower	$C058_H$
DPR11_U	Data Protection Range 11, Upper	$C05C_H$
DPR12_L	Data Protection Range 12, Lower	$C060_H$
DPR12_U	Data Protection Range 12, Upper	$C064_H$
DPR13_L	Data Protection Range 13, Lower	$C068_H$
DPR13_U	Data Protection Range 13, Upper	$C06C_H$
DPR14_L	Data Protection Range 14, Lower	$C070_H$
DPR14_U	Data Protection Range 14, Upper	$C074_H$
DPR15_L	Data Protection Range 15, Lower	$C078_H$
DPR15_U	Data Protection Range 15, Upper	$C07C_H$
DPR16_L	Data Protection Range 16, Lower	$C080_H$
DPR16_U	Data Protection Range 16, Upper	$C084_H$
DPR17_L	Data Protection Range 17, Lower	$C088_H$
DPR17_U	Data Protection Range 17, Upper	$C08C_H$
DPR18_L	Data Protection Range 18, Lower	$C090_H$
DPR18_U	Data Protection Range 18, Upper	$C094_H$
DPR19_L	Data Protection Range 19, Lower	$C098_H$
DPR19_U	Data Protection Range 19, Upper	$C09C_H$
DPR20_L	Data Protection Range 20, Lower	$C0A0_H$
DPR20_U	Data Protection Range 20, Upper	$C0A4_H$
DPR21_L	Data Protection Range 21, Lower	$C0A8_H$
DPR21_U	Data Protection Range 21, Upper	$C0AC_H$
DPR22_L	Data Protection Range 22, Lower	$C0B0_H$
DPR22_U	Data Protection Range 22, Upper	$C0B4_H$
DPR23_L	Data Protection Range 23, Lower	$C0B8_H$
DPR23_U	Data Protection Range 23, Upper	$C0BC_H$

(table continues...)

Table 31 (continued) Core Special Function Registers (CSFR)

Register name	Description	Address offset
CPR0_L	Code Protection Range 0, Lower	$D000_H$
CPR0_U	Code Protection Range 0, Upper	$D004_H$
CPR1_L	Code Protection Range 1, Lower	$D008_H$
CPR1_U	Code Protection Range 1, Upper	$D00C_H$
CPR2_L	Code Protection Range 2, Lower	$D010_H$
CPR2_U	Code Protection Range 2, Upper	$D014_H$
CPR3_L	Code Protection Range 3, Lower	$D018_H$
CPR3_U	Code Protection Range 3, Upper	$D01C_H$
CPR4_L	Code Protection Range 4, Lower	$D020_H$
CPR4_U	Code Protection Range 4, Upper	$D024_H$
CPR5_L	Code Protection Range 5, Lower	$D028_H$
CPR5_U	Code Protection Range 5, Upper	$D02C_H$
CPR6_L	Code Protection Range 6, Lower	$D030_H$
CPR6_U	Code Protection Range 6, Upper	$D034_H$
CPR7_L	Code Protection Range 7, Lower	$D038_H$
CPR7_U	Code Protection Range 7, Upper	$D03C_H$
CPR8_L	Code Protection Range 8, Lower	$D040_H$
CPR8_U	Code Protection Range 8, Upper	$D044_H$
CPR9_L	Code Protection Range 9, Lower	$D048_H$
CPR9_U	Code Protection Range 9, Upper	$D04C_H$
CPR10_L	Code Protection Range 10, Lower	$D050_H$
CPR10_U	Code Protection Range 10, Upper	$D054_H$
CPR11_L	Code Protection Range 11, Lower	$D058_H$
CPR11_U	Code Protection Range 11, Upper	$D05C_H$
CPR12_L	Code Protection Range 12, Lower	$D060_H$
CPR12_U	Code Protection Range 12, Upper	$D064_H$
CPR13_L	Code Protection Range 13, Lower	$D068_H$
CPR13_U	Code Protection Range 13, Upper	$D06C_H$
CPR14_L	Code Protection Range 14, Lower	$D070_H$
CPR14_U	Code Protection Range 14, Upper	$D074_H$
CPR15_L	Code Protection Range 15, Lower	$D078_H$
CPR15_U	Code Protection Range 15, Upper	$D07C_H$
CPXE_0	Code Protection Execute Enable Set-0	$E000_H$
CPXE_1	Code Protection Execute Enable Set-1	$E004_H$
CPXE_2	Code Protection Execute Enable Set-2	$E008_H$
CPXE_3	Code Protection Execute Enable Set-3	$E00C_H$

(table continues...)

Table 31 (continued) Core Special Function Registers (CSFR)

Register name	Description	Address offset
CPXE_4	Code Protection Execute Enable Set-4	$E040_H$
CPXE_5	Code Protection Execute Enable Set-5	$E044_H$
CPXE_6	Code Protection Execute Enable Set-6	$E048_H$
CPXE_7	Code Protection Execute Enable Set-7	$E04C_H$
DPRE_0	Data Protection Read Enable Set-0	$E010_H$
DPRE_1	Data Protection Read Enable Set-1	$E014_H$
DPRE_2	Data Protection Read Enable Set-2	$E018_H$
DPRE_3	Data Protection Read Enable Set-3	$E01C_H$
DPRE_4	Data Protection Read Enable Set-4	$E050_H$
DPRE_5	Data Protection Read Enable Set-5	$E054_H$
DPRE_6	Data Protection Read Enable Set-6	$E058_H$
DPRE_7	Data Protection Read Enable Set-7	$E05C_H$
DPWE_0	Data Protection Write Enable Set-0	$E020_H$
DPWE_1	Data Protection Write Enable Set-1	$E024_H$
DPWE_2	Data Protection Write Enable Set-2	$E028_H$
DPWE_3	Data Protection Write Enable Set-3	$E02C_H$
DPWE_4	Data Protection Write Enable Set-4	$E060_H$
DPWE_5	Data Protection Write Enable Set-5	$E064_H$
DPWE_6	Data Protection Write Enable Set-6	$E068_H$
DPWE_7	Data Protection Write Enable Set-7	$E06C_H$
TPS_CON	Timer Protection Configuration Register	$E400_H$
TPS_TIMER0	Temporal Protection Timer 0	$E404_H$
TPS_TIMER1	Temporal Protection Timer 1	$E408_H$
TPS_TIMER2	Temporal Protection Timer 2	$E40C_H$
PMA0[2)]	Physical Memory Attributes Register 0	8100_H
PMA1[2)]	Physical Memory Attributes Register 1	8104_H
PMA2[2)]	Physical Memory Attributes Register 2	8108_H
DCON2	Data Memory Configuration Register-2	9000_H
DCON1	Data memory Configuration Register-1	9008_H
SMACON[2)] [1)]	SIST mode Control Register	$900C_H$
DSTR	Data Synchronous Error Trap Register	9010_H
DATR	Data Asynchronous Error Trap Register	9018_H
DEADD	Data Error Address Register	$901C_H$
DIEAR	Data Integrity Error Address Register	9020_H
DIETR	Data Integrity Error Trap Register	9024_H
DCON0	Data Memory Configuration Register-0	9040_H

(table continues...)

Table 31 (continued) Core Special Function Registers (CSFR)

Register name	Description	Address offset
PSTR	Program Synchronous Error Trap Register	9200_H
PCON1	Program Memory Configuration Register-1	9204_H
PCON2	Program Memory Configuration Register-2	9208_H
PCON0	Program Memory Configuration Register-0	$920C_H$
PIEAR	Program Integrity Error Address Register	9210_H
PIETR	Program Integrity Error Trap Register	9214_H
Debug and Trace Registers		
DBGSR	Debug Status Register	$FD00_H$
EXEVT	External Event Register	$FD08_H$
CREVT	Core Register Event Register	$FD0C_H$
SWEVT	Software Event Register	$FD10_H$
DBGACT	Debug Action Register	$FD14_H$
TRIG_ACC	Trigger Accumulator Register	$FD30_H$
DMS	Debug Monitor Start Address Register	$FD40_H$
DCX	Debug Context Save Address Register	$FD44_H$
DBGTCR	Debug Trap Control Register	$FD48_H$
DBGCFG	Debug Configuration Register	$FD4C_H$
TRCCFG	Trace Configuration Register	$FD50_H$
TRCFLT	Trace Filtering Register	$FD54_H$
TRCLIM	Trace Bandwidth Limiter	$FD58_H$
TR0EVT	Trigger Event 0 Register	$F000_H$
TR0ADR	Trigger Address 0 Register	$F004_H$
TR1EVT	Trigger Event 1 Register	$F008_H$
TR1ADR	Trigger Address 1 Register	$F00C_H$
TR2EVT	Trigger Event 2 Register	$F010_H$
TR2ADR	Trigger Address 2 Register	$F014_H$
TR3EVT	Trigger Event 3 Register	$F018_H$
TR3ADR	Trigger Address 3 Register	$F01C_H$
TR4EVT	Trigger Event 4 Register	$F020_H$
TR4ADR	Trigger Address 4 Register	$F024_H$
TR5EVT	Trigger Event 5 Register	$F028_H$
TR5ADR	Trigger Address 5 Register	$F02C_H$
TR6EVT	Trigger Event 6 Register	$F030_H$

(table continues...)

Table 31 (continued) Core Special Function Registers (CSFR)

Register name	Description	Address offset
TR6ADR	Trigger Address 6 Register	$F034_H$
TR7EVT	Trigger Event 7 Register	$F038_H$
TR7ADR	Trigger Address 7 Register	$F03C_H$
Performance counters		
CCTRL	Counter Control Register	$FC00_H$
CCNT	CPU Clock Count Register	$FC04_H$
ICNT	Instruction Count Register	$FC08_H$
M1CNT	Multi Count Register 1	$FC0C_H$
M2CNT	Multi Count Register 2	$FC10_H$
M3CNT	Multi Count Register 3	$FC14_H$
Floating Point registers (if implemented)		
FPU_TRAP_CON	Trap Control Register	$A000_H$
FPU_TRAP_PC	Trapping Instruction Program Control Register	$A004_H$
FPU_TRAP_OPC	Trapping Instruction Opcode Register	$A008_H$
FPU_TRAP_SRC1_L	Trapping Instruction SRC1_L Operand Register	$A010_H$
FPU_TRAP_SRC1_U	Trapping Instruction SRC1_U Operand Register	$A014_H$[3]
FPU_TRAP_SRC2_L	Trapping Instruction SRC2_L Operand Register	$A018_H$
FPU_TRAP_SRC2_U	Trapping Instruction SRC2_U Operand Register	$A01C_H$[3]
FPU_TRAP_SRC3_L	Trapping Instruction SRC3_L Operand Register	$A020_H$
FPU_TRAP_SRC3_U	Trapping Instruction SRC3_U Operand Register	$A024_H$[3]
FPU_SYNC_TRAP_CON	Synchronous Trap Control Register	$A030_H$
FPU_SYNC_TRAP_OPC	Synchronous Trapping Instruction Opcode Register	$A034_H$
Virtualization registers (If implemented)[4]		
VCON0	Virtualization Configuration 0 Register	$B000_H$
VCON1	Virtualization Configuration 1 Register	$B004_H$
VCON2	Virtualization Configuration 2 Register	$B008_H$
BHV[2]	Base Hypervisor Vector Table Register	$B010_H$
VM0_ICR	Virtual Machine 0 Interrupt Control Register	$B100_H$
VM1_ICR	Virtual Machine 1 Interrupt Control Register	$B104_H$
VM2_ICR	Virtual Machine 2 Interrupt Control Register	$B108_H$
VM3_ICR	Virtual Machine 3 Interrupt Control Register	$B10C_H$
VM4_ICR	Virtual Machine 4 Interrupt Control Register	$B110_H$
VM5_ICR	Virtual Machine 5 Interrupt Control Register	$B114_H$

(table continues...)

Table 31 (continued) Core Special Function Registers (CSFR)

Register name	Description	Address offset
VM6_ICR	Virtual Machine 6 Interrupt Control Register	$B118_H$
VM7_ICR	Virtual Machine 7 Interrupt Control Register	$B11C_H$
VM0_PETHRESH	Virtual Machine 0 Interrupt Pre-emption Threshold Register	$B200_H$
VM1_PETHRESH	Virtual Machine 1 Interrupt Pre-emption Threshold Register	$B204_H$
VM2_PETHRESH	Virtual Machine 2 Interrupt Pre-emption Threshold Register	$B208_H$
VM3_PETHRESH	Virtual Machine 3 Interrupt Pre-emption Threshold Register	$B20C_H$
VM4_PETHRESH	Virtual Machine 4 Interrupt Pre-emption Threshold Register	$B210_H$
VM5_PETHRESH	Virtual Machine 5 Interrupt Pre-emption Threshold Register	$B214_H$
VM6_PETHRESH	Virtual Machine 6 Interrupt Pre-emption Threshold Register	$B218_H$
VM7_PETHRESH	Virtual Machine 7 Interrupt Pre-emption Threshold Register	$B21C_H$

1) These registers are SAFETY_ENDINIT protected
2) These registers are ENDINIT protected
3) Only if double precision FPU is implement
4) Only accessible by the hypervisor

Revision history

Reference	Description of change(s)	Comment
V0.7		
	First public release	
V0.9, 2022-11-18		
Accessing core special function registers (CSFRs)	Clarified behavior of undefined register accesses by MTDCR/MFDCR	
Internal protection traps (trap class 1)	Added mention of cache instructions as potential triggers of MPR and MPW traps	
Core register table	Added registers DPRn_U (n=16-23) and DPRn_L (n=16-23) Added footnote to BHV register to indicate endinit protection	
Program status word register (PSW)	Updated PSW.S description	
V0.9.1, 2023-02-14		
	Minor typographic (typo) corrections Fixed missing section hyperlinks Overall register cleanup	
Synthesized addressing modes	Added mention of JRI in the scope of program counter relative addressing	
General purpose registers (GPRs)	Reordered section	
Instruction errors (trap class 2)	Added mention of TRAPINV	
PSW user status bits (FPU view)	Added an FPU representation of the PSW.USB	
Hypervisor interrupt entry sequence	PSW.USB are cleared on transition from Guest VM to Hypervisor	
Crossing protection boundaries	Distinguishing code and data accesses.	
Hypervisor trap handling	PSW.USB are cleared on transition from Guest VM to Hypervisor	
Performance counter registers	Changed to sub-chapter. This change affects the numbering of the next sections Reverted to generic reset types	
Debug control registers	Reverted to generic reset types	
Trace configuration registers	Reverted to generic reset types	
V0.9.2, 2023-12-05		

Reference	Description of change(s)	Comment
Crossing protection boundaries	Clarified boundary crossing wording.	
Virtual machine pre-emption threshold (VMn_PETHRESH)	Corrected typo in register name	
V1.0.0, 2024-02-14		
Overall document	Minor formatting and type consistency updates.	
Interrupt system	Reworded introduction	
Memory model	Reordered section	
Semaphores and atomic operations	Reordered section and minor updates	
Base + offset addressing	Added clarification regarding accessing the last 32-KByte section of each segment	
Synthesized addressing modes	Reordered section and minor updates	
Software routines	Added mention of conversion between single precision floating-point and long or unsigned long	
Single precision underflow	Clarified that both conditions are required for flagging underflow	
Double precision underflow	Clarified that both conditions are required for flagging underflow	
Exceptions	FX raised asynchronous traps (CAE) rather than the previously incorrectly mentioned synchronous one (CSE).	
PSW user status bits (FPU view)	Added ALU flags for completeness.	
Address range	Reordered section and minor updates	
Switching execution from guest virtual machine in HRA to guest virtual machine in HRB	Updated text and diagram (HRHV(VM0) to HRB(VM2) switch - After the context restore) to correctly make reference to CSA HVB1 rather than the incorrect mention of CSA HV3	
Virtualization: Interrupt system	Reordered section and minor updates. Added clarification about handling of interrupts for guest VM while the hypervisor is running	
Interrupt use cases	Added clarification about handling of interrupts for guest VM while the hypervisor is running	

Edition 2024-02-14

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference IFX-hnq1591720267843