TriCore™ TC1.6.2 core architecture manual

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 (this volume) 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 labelled 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

TriCore™ Compiler Writer's Guide

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 startup 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 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.
- Pins using negative logic are indicated by an overbar: $\overline{\text{BRKOUT}}$.

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

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.

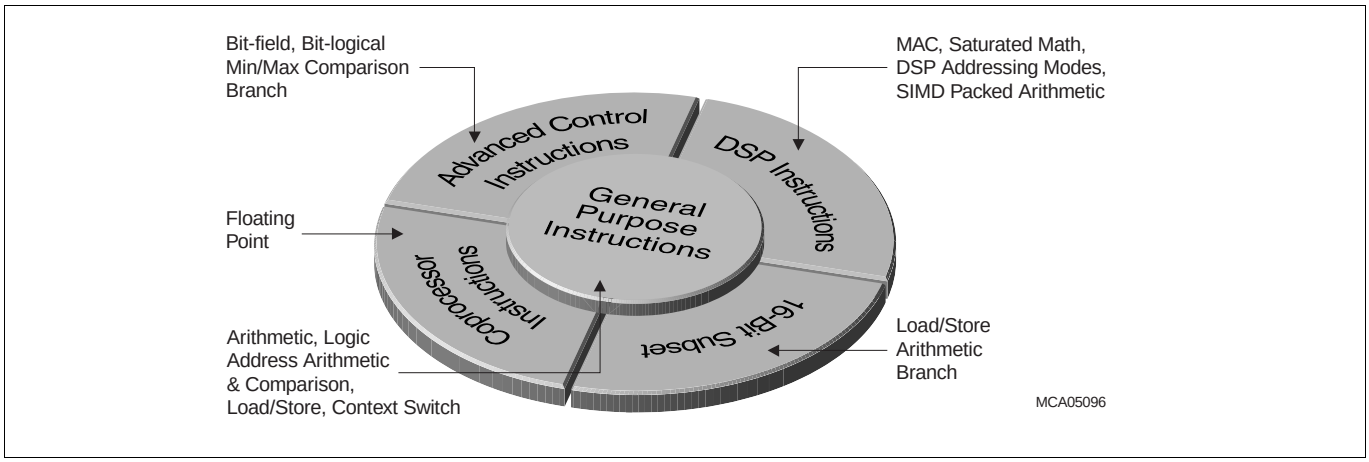


Figure 1 TriCore Architecture Overview

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
- Most instructions executed in one cycle
- Branch instructions (using branch prediction)
- Low interrupt latency with fast automatic context switch using wide pathway to on-chip memory
- Dedicated interface to application-specific coprocessors to allow the addition of customised instructions
- Zero overhead loop capabilities

- Dual, single-clock-cycle, 16x16-bit multiply-accumulate unit (with optional saturation)
- Optional Floating-Point Unit (FPU) and Memory Management Unit (MMU)
- Extensive bit handling capabilities
- Single Instruction Multiple Data (SIMD) packed data operations (2x16-bit or 4x 8-bit operands)
- Flexible interrupt prioritization scheme
- Byte and bit addressing
- Little-endian byte ordering for data memory and CPU registers
- Memory protection
- Debug support

1.2 Programming Model

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

- Architectural Registers Page 2
- Data Types Page 3
- Memory Model Page 3
- Addressing Modes Page 3

The Programming Model is described in detail in the chapter “Programming Model” on Page 1.

1.2.1 Architectural Registers

The architectural registers consist of:

- 32 General Purpose Registers (GPRs)
- Program Counter (PC)
- Two 32-bit registers containing status flags, previous execution information and protection information (PCXI - Previous Context Information register, and PSW -Program Status Word)

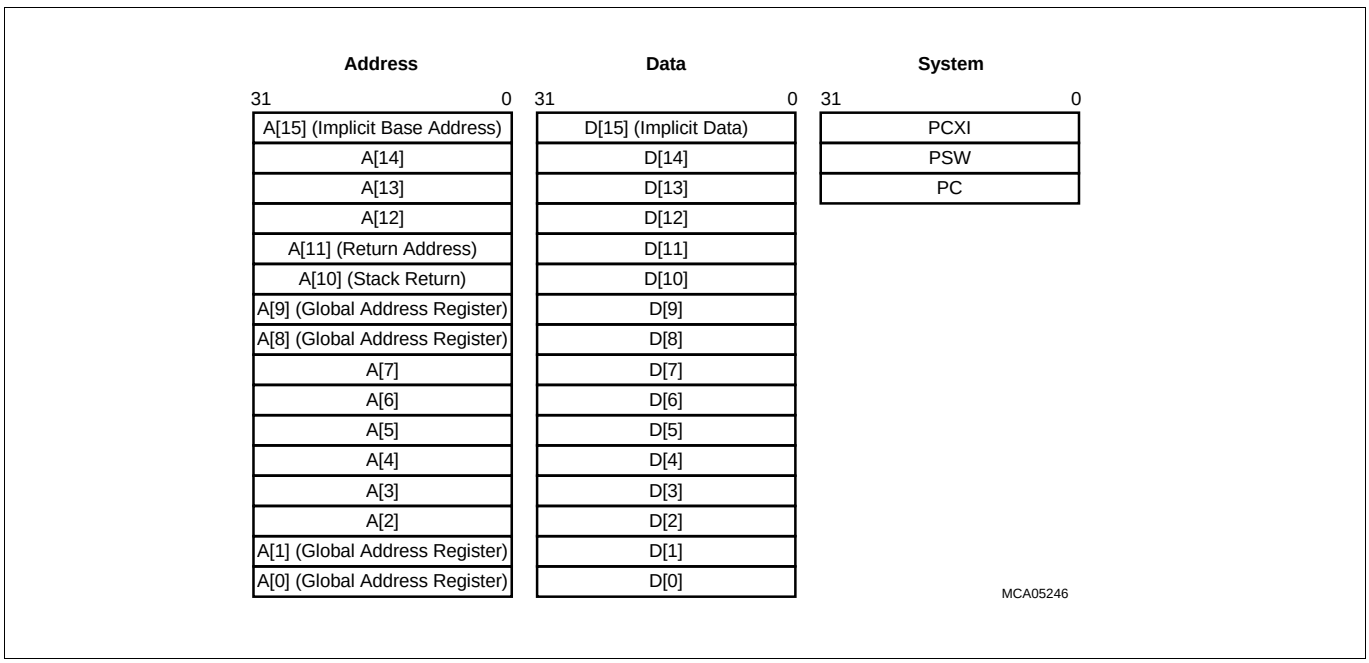


Figure 2 Architectural Registers

The PCXI, 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" on Page 1**) and are not saved and restored across calls or interrupts. They are normally used by the operating system to reduce system overhead**"Run Control Features" on Page 1**.

In addition to the General Purpose Registers (GPRs), the core registers are composed of a certain number of Core Special Function Registers (CSFRs). See **"General Purpose and System Registers" on Page 1**.

1.2.2 Data Types

The instruction set supports operations on:

- Boolean
- Bit String
- Byte
- Signed Fraction
- Address
- Signed / Unsigned Integer
- IEEE-754 Single-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-bits) of unified program and I/O memory.

The address space is divided into 16 regions or 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-bits, 16-bits, 32-bits and 64-bits wide.

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" on Page 12**.

1.3 Tasks and Contexts

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

SMTs are created through the services of a real-time kernel or Operating System, 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. SMTs 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 default behaviour of this mode may be overriden by the system control register).
- **Supervisor Mode**: Permits read/write access to system registers and all peripheral devices. Tasks in this mode may disable interrupts.

Individual modes are enabled or disabled primarily 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

See **"Tasks and Functions" on Page 1**.

1.4 Interrupt System

A key feature of the architecture is its powerful and flexible interrupt system. The interrupt system is built around programmable Service Request Nodes (SRNs).

A Service Request is defined as an interrupt request or a DMA (Direct Memory Access) request. A service request may come 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 (PS) from a vector table in data memory. In the TriCore architecture, service requests jump 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.

1.4.1 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 Interrupt Control Unit (ICU) determines which source will win arbitration based on the priority number

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 sources is 255. Programmable options 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 numbers are not hardwired to individual sources, but are assigned by software executed during the power-on boot sequence.

See **“Interrupt System” on Page 1**.

1.5 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 traps are further ~~classified as synchronous or asynchronous, hardware or software.~~ Each trap 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 comprised of a vector of code blocks. Each code block provides an entry for one trap. When a trap is taken, the TIN is placed in data register D[15].

The trap classes are:

- MMU (Memory Management Unit)
- Internal Protection
- Instruction Error
- Context Management
- System Bus and Peripherals
- Assertion Trap
- System Call
- Non-Maskable Interrupt (NMI)

See **“Trap System” on Page 1**.

1.6 Protection System

One of the domains that TriCore supports is safety-critical embedded applications. 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 unauthorised tasks from accessing critical system peripherals~~.

The protection system also facilitates debugging. It detects and traps errors that might otherwise go unnoticed until it was 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 **Section 1.5**, but covered in detail in **“Trap System” on Page 1**.

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 everything in Supervisor mode. (The default behaviour of the User-1 mode may be overriden by the system 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” on Page 1**.

1.7 Memory Management Unit

TriCore can make use of an optional Memory Management Unit (MMU). When configured with an MMU, the memory space has two addressing regions; physical and virtual. The physical and virtual address space is 4 GBytes in each instance, with those 4 GBytes each divided into sixteen, 256 MByte segments.

Segments [8_H-F_H] bypass virtual mapping and are directly, physically used. Segments [0_H-7_H] are virtually mapped by the MMU when it is present and enabled, or physically mapped when the MMU is not present or disabled.

~~Virtual addresses are always translated into physical addresses before accessing memory~~. This translation to a physical address is either a Direct Translation or a Page Table Entry (PTE) Translation, depending on MMU mode and virtual address region:

- **Direct Translation**
  - If the virtual address belongs to the upper half of the virtual address space, then the virtual address is directly used as the physical address. If the virtual address belongs to the lower half of the address space and the processor is operating in Physical mode, then the virtual address is used indirectly as the physical address.
- **PTE**
  - If the processor is operating in Virtual mode and the virtual address belongs to the lower half of the address space, then the virtual address is translated using PTE. PTE translation is performed by replacing the Virtual Page Number (VPN) of the virtual address by a Physical Page Number (PPN) to obtain a physical address.

See **“Memory Management Unit (MMU)” on Page 1**.

1.8 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 Core 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 via 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” on Page 1**

1.9 TriCore Coprocessor Interface

TriCore implementations may choose to implement a coprocessor interface. Such interfaces allows hardware extensions to the standard TriCore instruction set.

2 Programming Model

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

- Supported data types **Page 1**
- Data formats in registers and memory **Page 2**
- The Memory model **Page 6**
- Addressing modes **Page 7**

2.1 Data Types

The instruction set supports operations on the following Data Types:

- Boolean **Page 1**
- Bit String **Page 1**
- Byte **Page 1**
- Signed Fraction **Page 1**
- Address **Page 1**
- Signed and Unsigned Integers **Page 2**
- IEEE-754 Single-precision Floating-point Number **Page 2**

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 and 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

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 Single-Precision Floating-Point Number

Depending on the particular implementation of the core architecture, IEEE-754 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

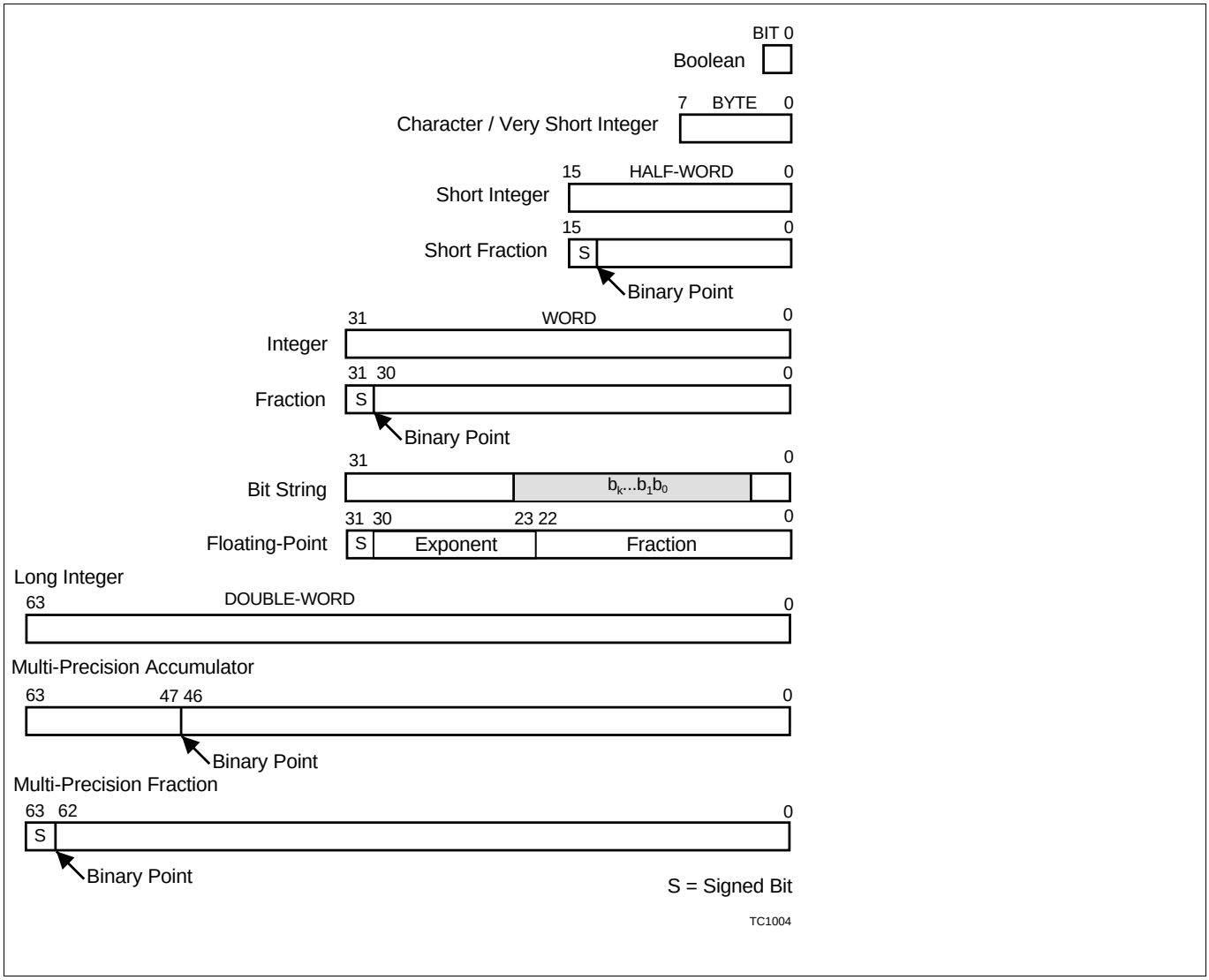


Figure 3 Supported Data Formats

2.2.1 Alignment Requirements

Alignment requirements differ for addresses and data (see **Table 1**). Address variables loaded into or stored from address registers, must always be Word-aligned.

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 of which programmers must be aware, specifically:

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

Alignment Rules

Table 1 Alignment rules for non-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	2 bytes (2_H)
	Double-Word	2 bytes (2_H)
Load, Store Address Register	Word	4 bytes (4_H)
	Double-Word	4 bytes (4_H)
SWAP.W, LDMST	Word	4 bytes (4_H)
CMPSWAP.W, SWAPMSK.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 2 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)
Load, Store Address Register	Word	4 bytes (4_H)
	Double-Word	8 bytes (8_H)
SWAP.W, LDMST, ST.T	Word	4 bytes (4_H)
CMPSWAP.W, SWAPMSK.W	Word	4 bytes (4_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.

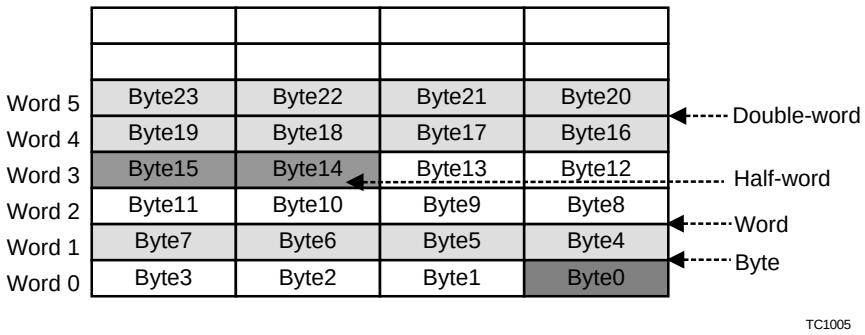


Figure 4 Byte Ordering

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 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" on Page 1** for more information on Traps.

Physical Memory Attributes

The physical memory attributes of segments zero to seven are implementation dependent. If an MMU is present and enabled, segments [0_H - 7_H] are considered virtual addresses that must be translated. If an MMU is not present the access characteristics are implementation dependent and may cause a trap.

Physical Memory Addresses

Physical memory addresses in ~~segment 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 space in the memory map. The base location of this 64 KBytes space is implementation-dependent.

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 memory mapping are implementation-specific.

For more information see **"Physical Memory Attributes (PMA)" on Page 1**.

Table 3 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 and/or write memory in atomic fashion:

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

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 SWAP.W is provided for this purpose. If 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 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.

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

2.5 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-bits, 16-bits, 32-bits, or 64-bits 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 4 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” on Page 12**.

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 to the most significant bits of the 32-bit address (**Figure 5**). Other bits are zero-filled.

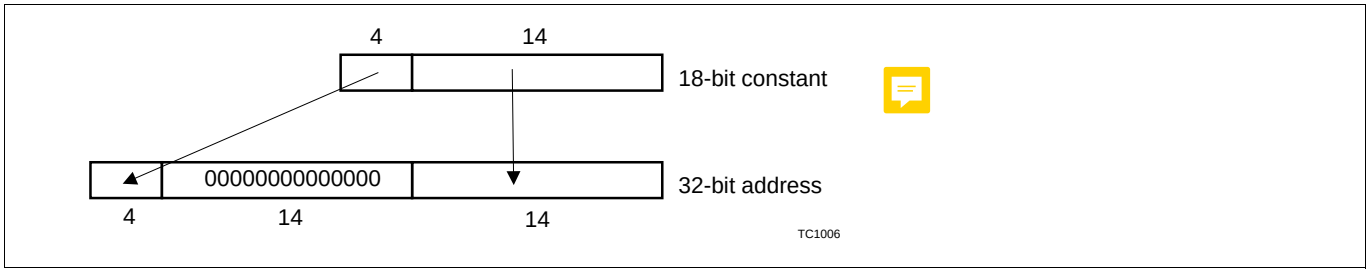


Figure 5 Translation of Absolute Address to Full Effective Address

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.

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.

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 sign-extended 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.

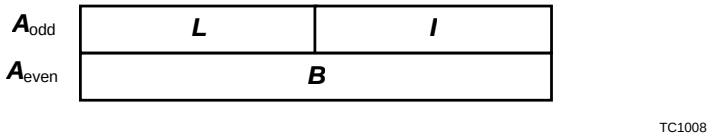


Figure 6 Circular Addressing Mode

The circular addressing mode uses an address register pair to hold the state it requires:

- The even register is always a base address (B).
- The most significant half of the odd register is the buffer size (L).
- The least significant half holds the index into the buffer (I).
- 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’ behaviour 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-bytes 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.

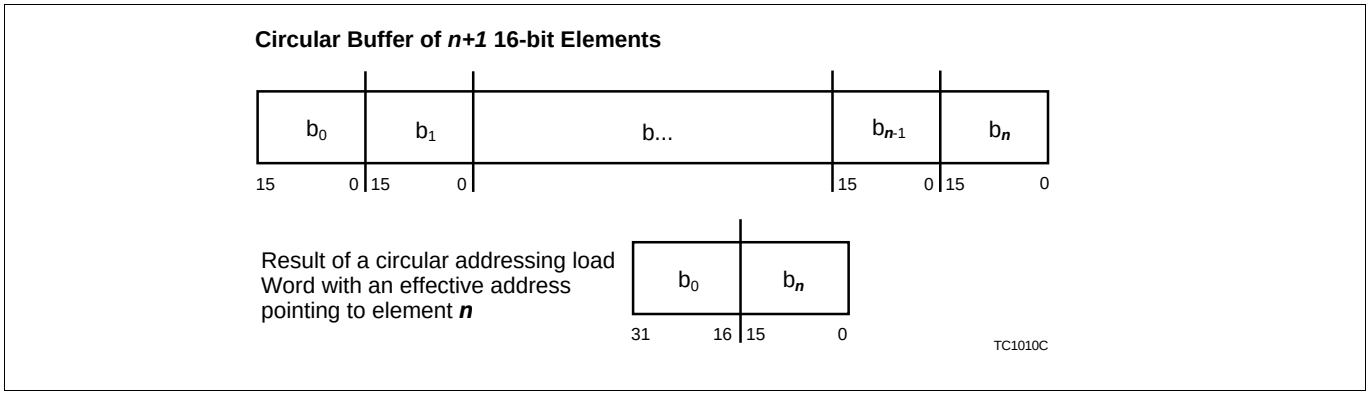


Figure 8 Circular Buffer End Case

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 user of optimal alignment of circular buffers etc., 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 ("Bit-Reverse Addressing" on Page 11).

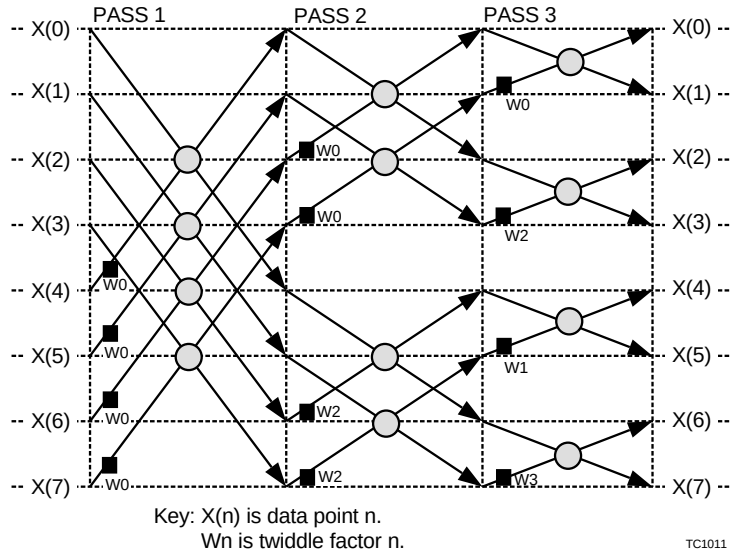


Figure 9 Bit-Reverse Addressing

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

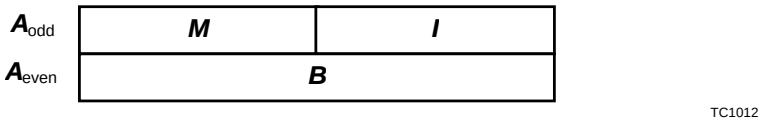


Figure 10 Register Pair for Bit-Reverse Addressing

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

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 5 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

Table 5 1024-point FFT Using 16-bit Values (cont'd)

I (decimal)	I (binary)	Reverse(I)	Rev[Rev(I) + Rev(M)]
512	0000001000000000_B	0000000001000000_B	0000011000000000_B
1536	0000011000000000_B	0000000001100000_B	0000010001100000_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 hardware 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 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.

PC-Relative Addressing

PC-relative addressing is the normal mode for branches and calls. However the architecture does not support direct PC-relative addressing of data. This is because the separate on-chip instruction and data memories make data access to the program memory expensive.

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 (as it almost always is for embedded systems), 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. The absolute address of the PC relative data is also known, and there is no need to synthesize PC-relative addressing.

For code that is dynamically loaded, or assembled into a binary image from position-independent pieces without the benefit of a relocating linker, the appropriate way to load a code address for use in PC-relative 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 copy the actual return address of the current function to another register.

3 General Purpose and System Registers

There are two types of Core Register, 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)
- Stack Management registers are (A[10] and ISP)
- SYSCON and CPU_ID registers
- Trap registers
- Context Management registers
- Memory Protection registers
- Memory Management registers
- Debug registers
- Floating Point 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.

ENDINIT Protection

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

~~When in the initialisation state, all Core Special Function Registers can be modified~~, using the MTCR instruction. 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 initialisation state are described as ENDINIT protected.

The transition between the initialisation 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 initialisation state.

The following registers are ENDINIT protected:

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

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

- SMACON, SYSCON, COMPAT, TPS_EXTIM_ENTRY_LVAL, TPS_EXTIM_EXIT_LVAL

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. The address registers (P[0], P[2], P[4], P[6], P[8], P[10], P[12], and P[14]) can be used in the same way.

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" on Page 10**.

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 odd/even 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].

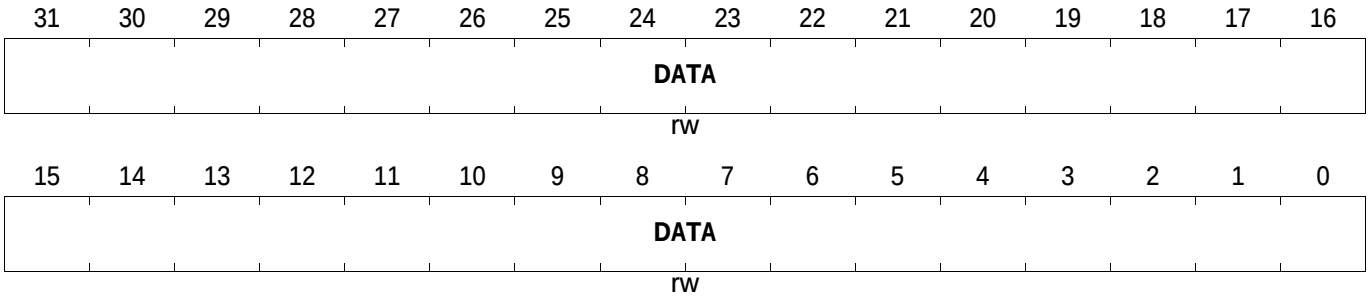
In order to support extended addressing modes, an even/odd address register pair holds the extended address reference as a pair of 32-bit address registers (A[8]/A[9] for example).

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.

Figure 11 shows the 32-bit wide GPRs.

Data General Purpose Registers

Dn (n=0-15)
Data Register n **($FF00_H$+n*4)** **Reset Value: Implementation Specific**



Field	Bits	Type	Description
DATA	[31:0]	rw	Data Register n Value

Address General Purpose Registers

An (n=0-15)

Address Register n **($FF80_H$+n*4)** **Reset Value: Implementation Specific**

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

General Purpose Registers (GPRs)

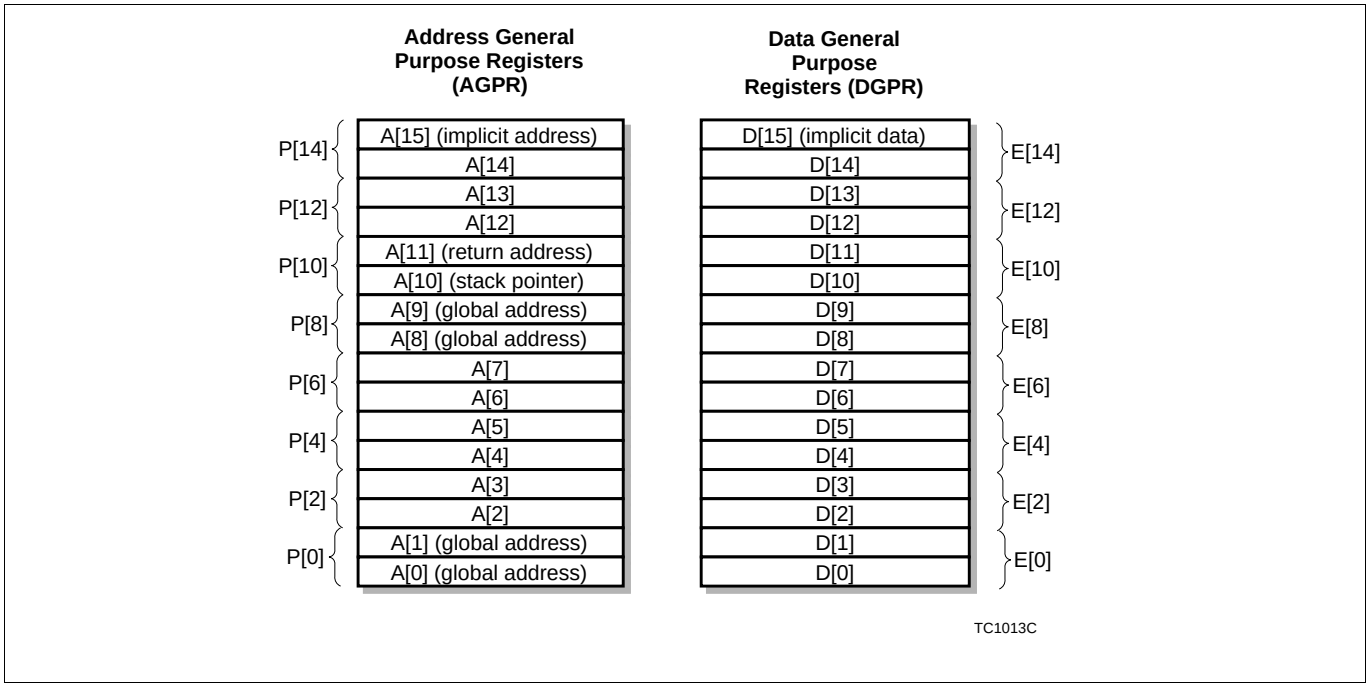


Figure 11 General Purpose Registers (GPRs)

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 **Chapter 4**.*

3.2 Program State Information Registers

The PC, PSW, and PCXI 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

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 when the core is halted. If the core is not in halt a write will have no effect.

PC
Program Counter Register **($FE08_H$)** **Reset Value: Implementation Specific**

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**

Program Status Word Register (PSW)

The Program Status Word register (PSW) is a 32-bit register that contains a 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
Program Status Word **($FE04_H$)** **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
PRS	S	PRS		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**
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	**Safety Task Identifier** The current task should be identified as a Safe Task.
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

Field	Bits	Type	Description
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 behaviour of this mode may be overriden by the system 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).
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.

Field	Bits	Type	Description
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.
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. $0ccccc_B$: 6-bit counter; trap on overflow. $10cccc_B$: 5-bit counter; trap on overflow. $110ccc_B$: 4-bit counter; trap on overflow. $1110cc_B$: 3-bit counter; trap on overflow. $11110c_B$: 2-bit counter; trap on overflow. 111110_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.

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 6 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:

Bits [23:16] of the PSW are reserved bits with no defined use in current versions of the architecture. They read as zero when the PSW is read via the MFCR (Move From Core Register) instruction after a system reset. Their value after writing to the PSW via the MTCR (Move To Core Register) instruction, is architecturally undefined and should be written as zero.

- 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.

Access Privilege Level Control (I/O Privilege)

Software Managed Tasks (SMTs) are created through the services of a real-time kernel or Operating System, 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. SMTs 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 default behaviour of this mode may be overriden by the system control register).
- **Supervisor Mode**: Permits read/write access to system registers and all peripheral devices. Tasks in this mode may disable interrupts.

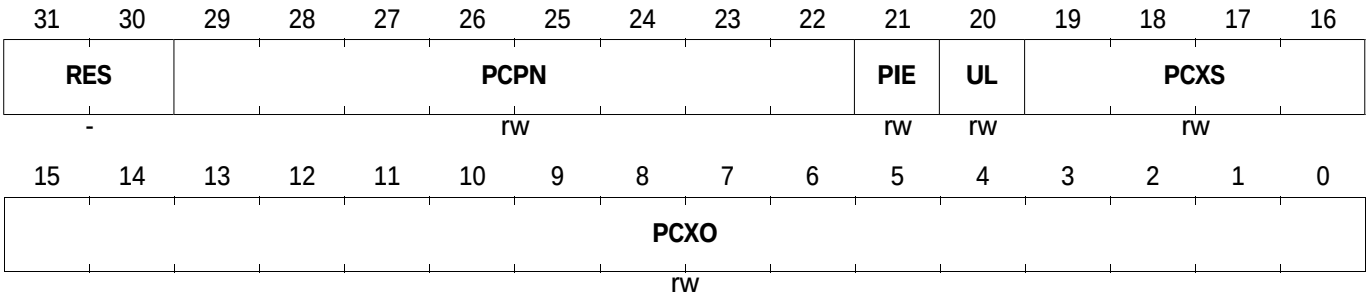
A set of state elements are associated with any task, and these are known collectively as the task's context. The context is everything the processor needs to define the state of the associated task and enable its continued execution. This includes the CPU General Registers that the task uses, the task's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

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
Previous Context Information and Pointer Register

($FE00_H$) **Reset Value: Implementation Specific**



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.

3.3 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 a PSW bit 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 contents are 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.

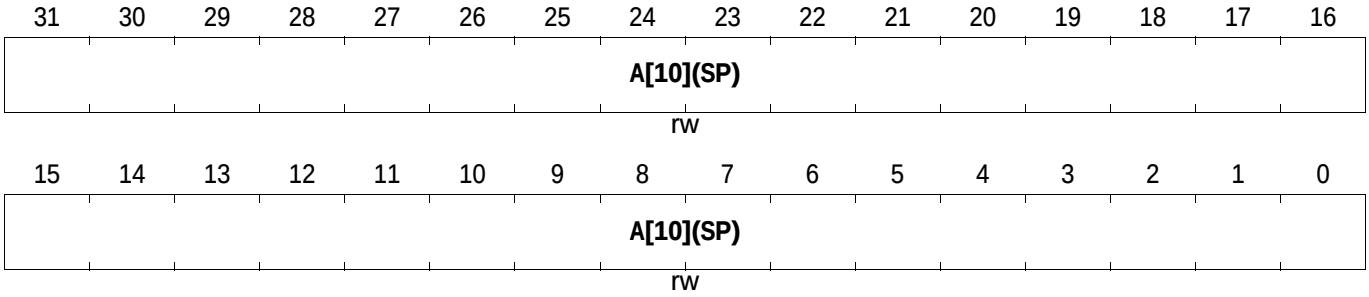
Usually it is only necessary to initialize the ISP once during the initialization routine. However, depending on application needs, the ISP can be modified during execution. Note that there is nothing preventing an ISR or system service routine from executing on a private stack.

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

Address Register A[10] (SP)

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

A[10](SP)
Address Register A[10] (Stack Pointer) ($FFA8_H$) Reset Value: Implementation Specific



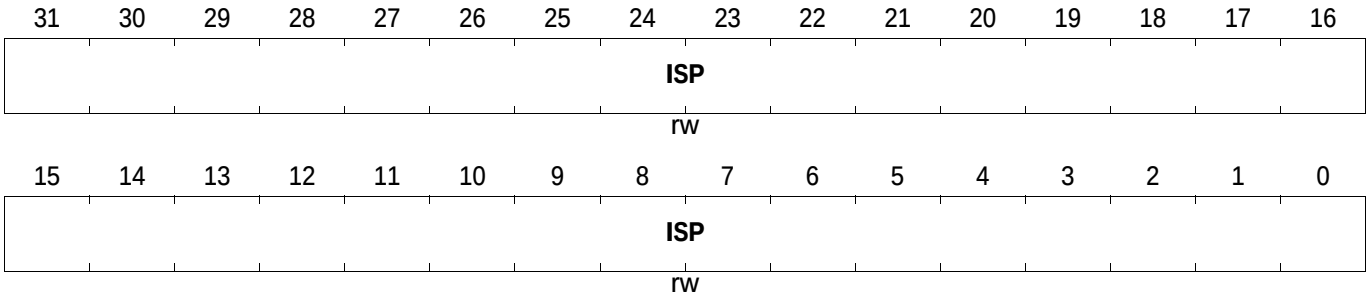
Field	Bits	Type	Description
A[10](SP)	[31:0]	rw	**Address Register A[10] (Stack Pointer)**

Interrupt Stack Pointer Register (ISP)

The Interrupt Stack Pointer is defined as follows.

Note: This register is ENDINIT protected.

ISP
Interrupt Stack Pointer **($FE28_H$)** **Reset Value: Implementation Specific**



Field	Bits	Type	Description
ISP	[31:0]	rw	**Interrupt Stack Pointer**

System Control Register (SYSCON)

The System Configuration Register provides the following functionality.

- Enable bit for Temperal 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.
- Disable for User-1 IO mode ability to enable and disable interrupts
- Boot halt status and release bit.
- Status indicator of the Free Context List Depletion condition.

Note: This register is SAFETY_ENDINIT protected with the exception of the FCDSF bit.

SYSCON
System Configuration Register (FE14$_H$) Reset Value: 0000 0000$_H$

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

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

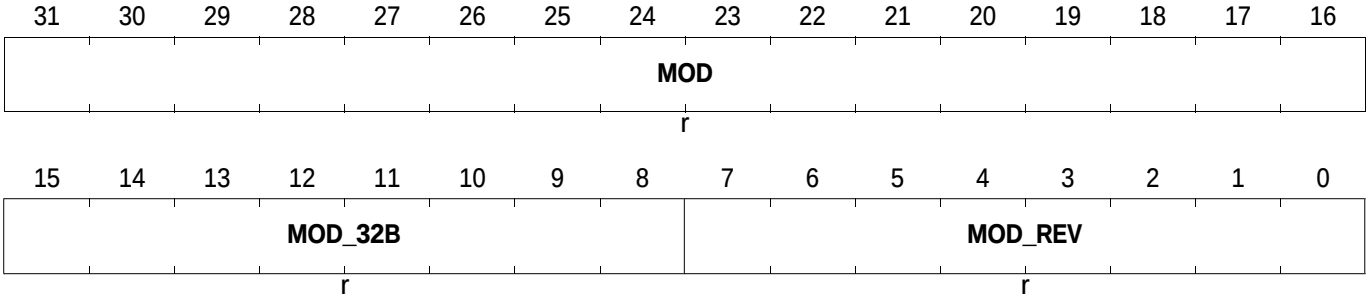
Field	Bits	Type	Description
RES	[31:25]	-	**Reserved**
BHALT	24	rwh	**Boot halt status and release** Following reset a CPU may be immediately placed in halt. In this case the BHALT bit will be set to “1”. The CPU will remain in halt until this bit is written to “0”. On a write from “1” to “0” the CPU will start execution from the program address defined program counter (PC) register. A write of this bit to “1” will be ignored.
RES	[23:18]	-	**Reserved**
U1_IOS	17	rw	**User-1 Peripheral access as supervisor.** 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.** Disable the execution of User-1 mode instructions in User-1 IO mode. Disables User-1 ability to enable and disable interrupts
RES	[15:9]	-	**Reserved**
ESDIS	8	rw	Emulator Space Disable
RES	[7:5]	rw	Reserved

Field	Bits	Type	Description
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.

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 are described in the product documentation. The CPU Identification Register identifies the CPU type and revision.

CPU_ID
CPU Module Identification **($FE18_H$)** **Reset Value: Implementation Specific**

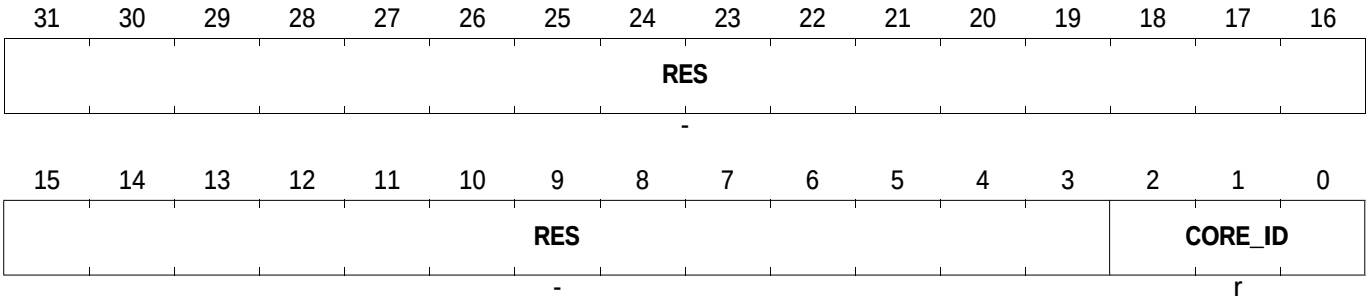


Field	Bits	Type	Description
MOD	[31:16]	r	**Module Identification Number** Used for module identification.
MOD_32B	[15:8]	r	**32-Bit Module Enable** A value of $C0_H$ in this field indicates a 32-bit module with a 32-bit module ID register.
MOD_REV	[7:0]	r	**Module Revision Number** Used for revision numbering. The value of the revision starts at 01_H (first revision) up to FF_H.

Core Identification Register (CORE_ID)

In a multiprocessor system each logical processor core is given a unique identification number. The Core Identification Register holds this number.

Core_ID
Core Identification **($FE1C_H$)** **Reset Value: Implementation Specific**



Field	Bits	Type	Description
RES	[31:3]	-	**Reserved**
CORE_ID	[2:0]	r	**Core Identification Number**

3.4 Compatibility Mode Register (COMPAT)

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

Compatibility Mode Register (COMPAT)

The contents of the register are implementation specific.

Note: This register is SAFETY_ENDINIT protected.

COMPAT
Compatibility Mode Register (9400_H) Reset Value: Implementation Specific

Field	Bits	Type	Description
Implementation Specific	[31:0]	-	**Implementation Specific**

3.5 Access Control Registers

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
SIST Mode Access Control (900C$_H$) Reset Value: Implementation Specific

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 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” on Page 1**.

3.7 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 register set is made up of several range registers (also called Range Table Entries).

Each Range Table Entry consists of a Segment Protection register pair and a bit field within a common Mode register. The register pair 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” on Page 1**.

3.8 Trap Registers

The Core Special Function Registers (CSFR) which control the Trap Registers are described in **“Trap System” on Page 1**.

3.9 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 memoryconfiguration are described in **“Physical Memory Attributes (PMA)” on Page 1**.

3.10 Core Debug Controller Registers

TriCore registers that support debugging are described in **“Core Debug Controller” on Page 1**

3.11 Floating Point Registers

The registers for the optional TriCore Floating Point Unit are described on **“FPU_TRAP_CON” on Page 11**.

3.12 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.

A MTCR instruction that accesses an undefined register location will have no effect. A MFCR instruction that accesses an undefined register location will return undefined data.

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 ‘virtual’ microcontroller. That model is generally supported by the services of a Real-time Executive 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.

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 and PSW. 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).

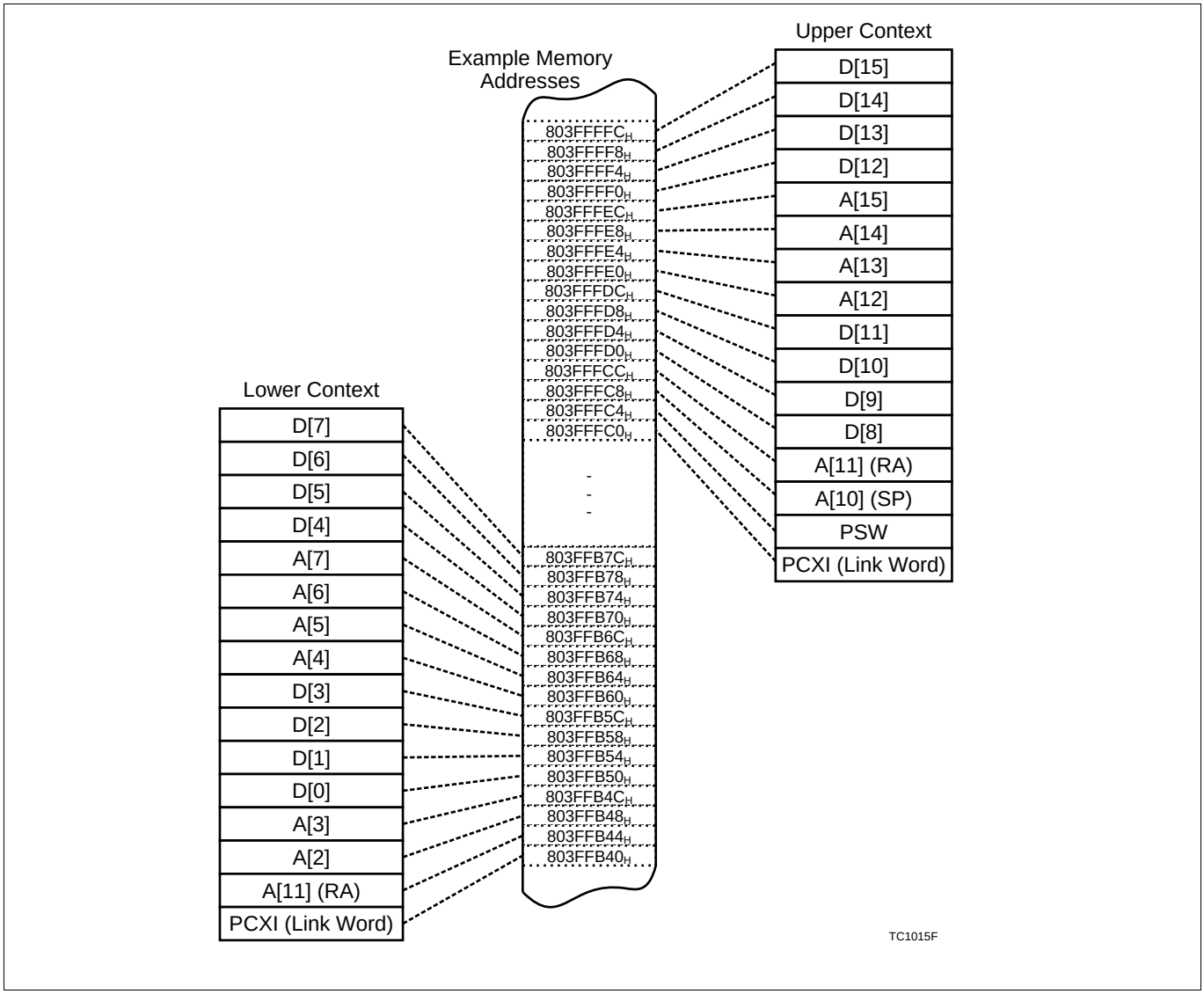


Figure 12 Upper and Lower Contexts

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.

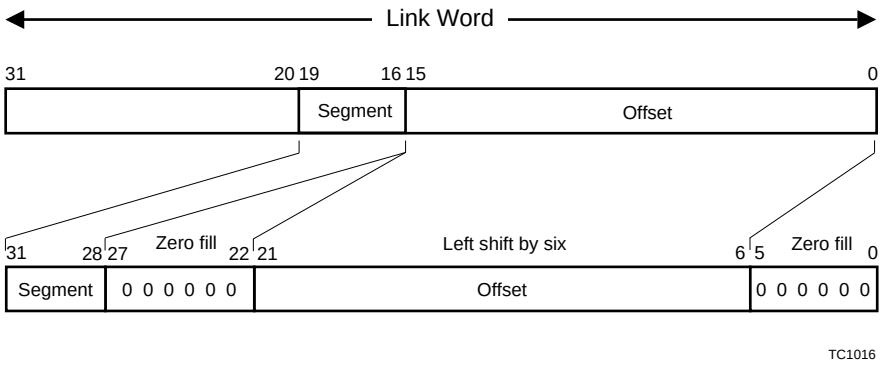


Figure 13 Generation of the Effective Address of a Context Save Area (CSA)

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" on Page 4**

4.2 Task Switching Operation

The architecture switches tasks when one of the events or instructions listed in **Table 7**, 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 7** '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 a 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 to restore the original values before exiting.

The upper context registers are not guaranteed to be static hardware registers. Conceptually, a function call or interrupt handler always begins execution with its own private set of upper context registers. The upper context registers of the interrupted or calling function are not inherited.

Only the A[10](SP), A[11](RA), PSW, PCXI and (in the case of a trap) D[15] registers start with architecturally defined values in the called function, trap handler or interrupt handler. A function, 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 7 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

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 (**Figure 14**) shows a simple configuration of CSAs within both context lists.

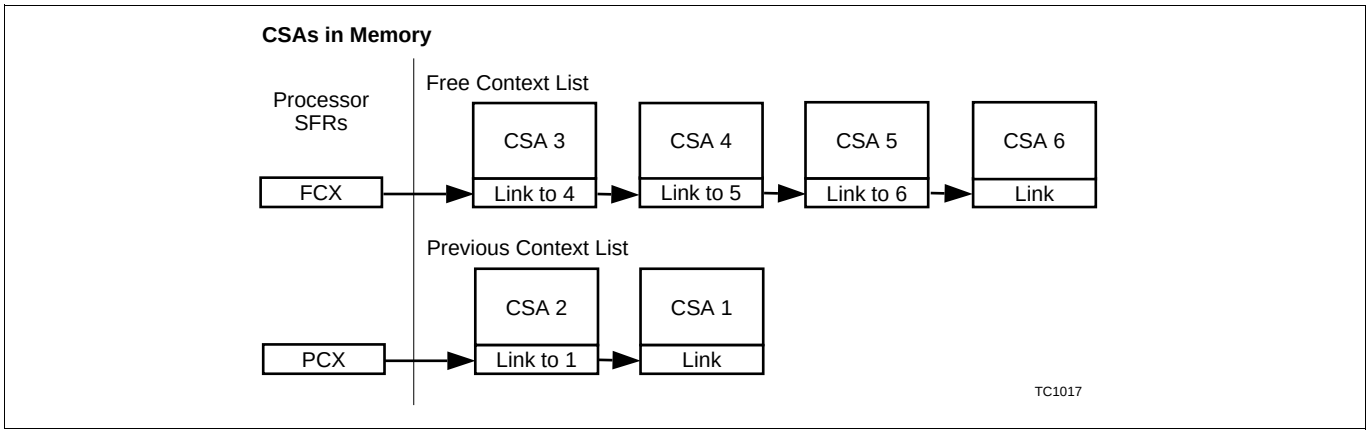


Figure 14 CSAs in Context Lists

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 recognise 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; i.e., 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 Management (Trap Class 3)” on Page 8**.

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.UL == 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 subfields; 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 SYSTRAP) 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 (SMTs).

PCXI.PIE is only typically zero within Trap handlers started within ISRs, e.g. an NMI or SYSTRAP 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 interrupt or trap sequence. For small interrupt and trap handlers that can execute entirely within this set of registers saved on the interrupt, 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 segments.

4.5 Context Switching for Function Calls

When a function call is made (the CALL 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 instruction 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 Fast Function Calls with FCALL/FRET

In situations where the saving and restoring of the upper context registers is not required an FCALL instruction may be used in preference to a CALL. The FCALL instruction performs a call jump and in parallel saves the current return address (A11) 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 A10 and A11).

To return from a function called by an FCALL, an FRET instruction is executed. This performs a jump to the current return address (A11) and loads the previous A11 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.

4.7 Context Save and Restore Examples

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

4.7.1 Context Save

Figure 15 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.

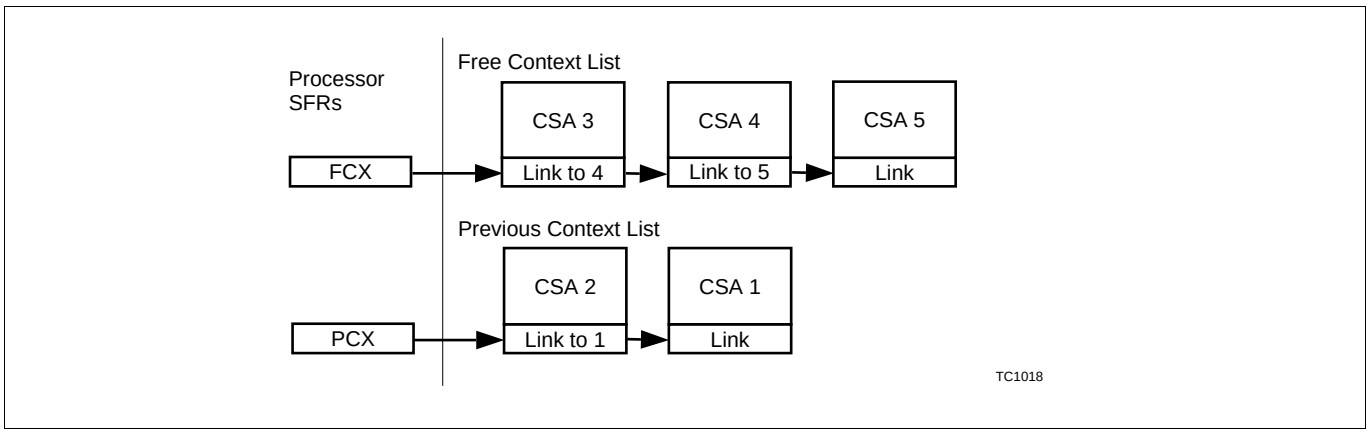


Figure 15 CSAs and Processor State Prior to Context Save

Figure 16 shows the steps taken during the context save operation. The numbers in the figure correspond to the steps listed after the figure.

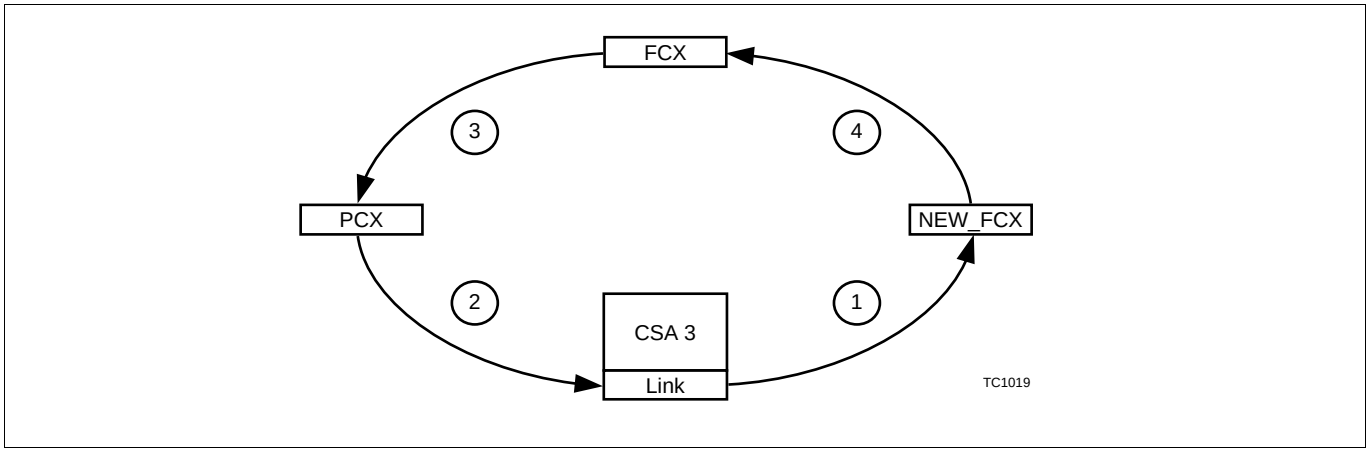


Figure 16 CSA and Processor SFR Updates on a Context Save Process

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 user.
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 **Figure 17**. The processor context to be saved is now written into the rest of CSA3.

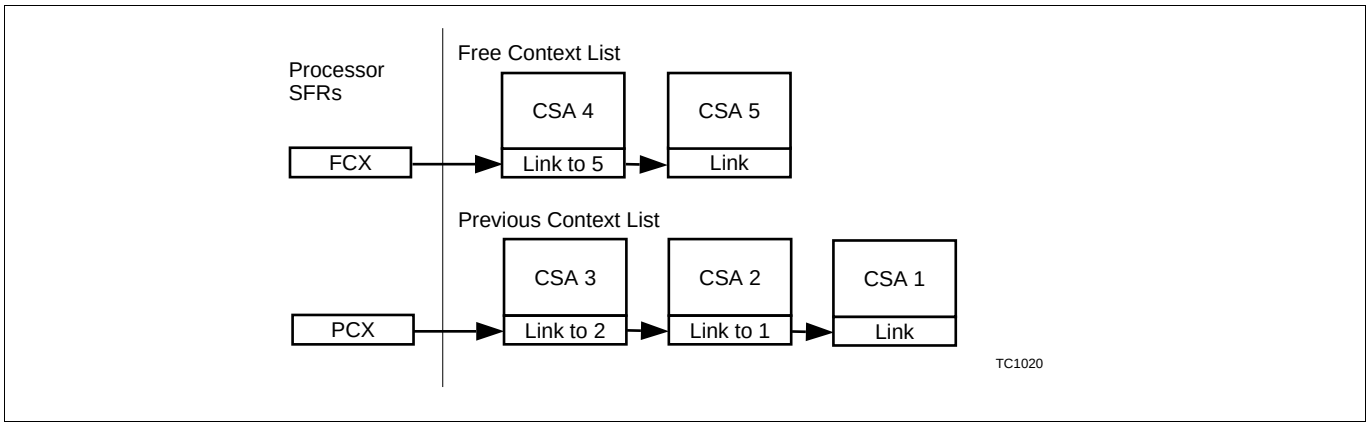


Figure 17 CSAs and Processor State After Context Save

4.7.2 Context Restore

The example in **Figure 18**, 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.

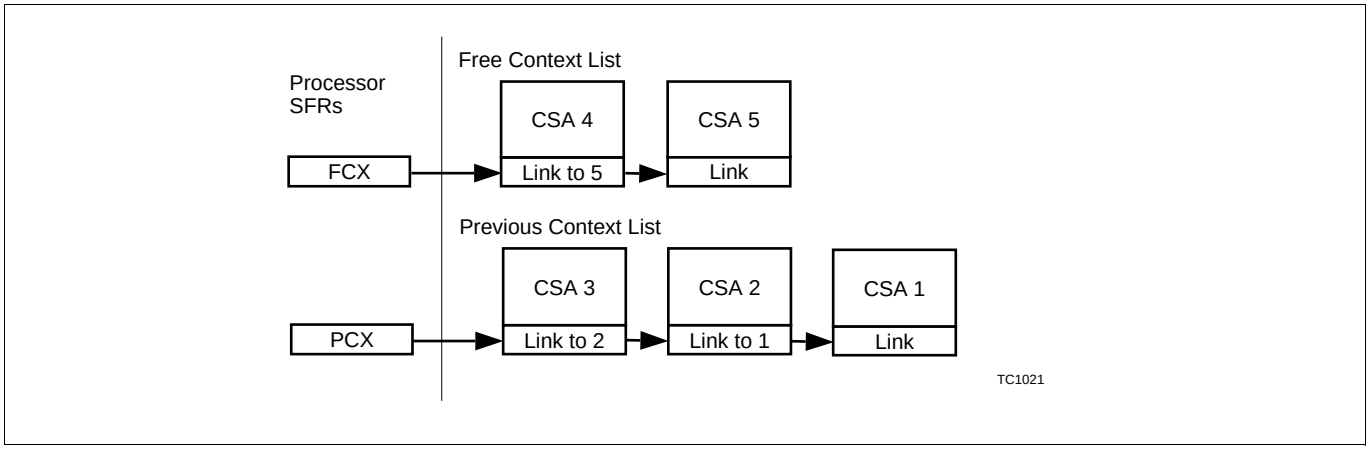


Figure 18 CSAs and Processor State Prior to Context Restore

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.

Figure 19 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 user.
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.

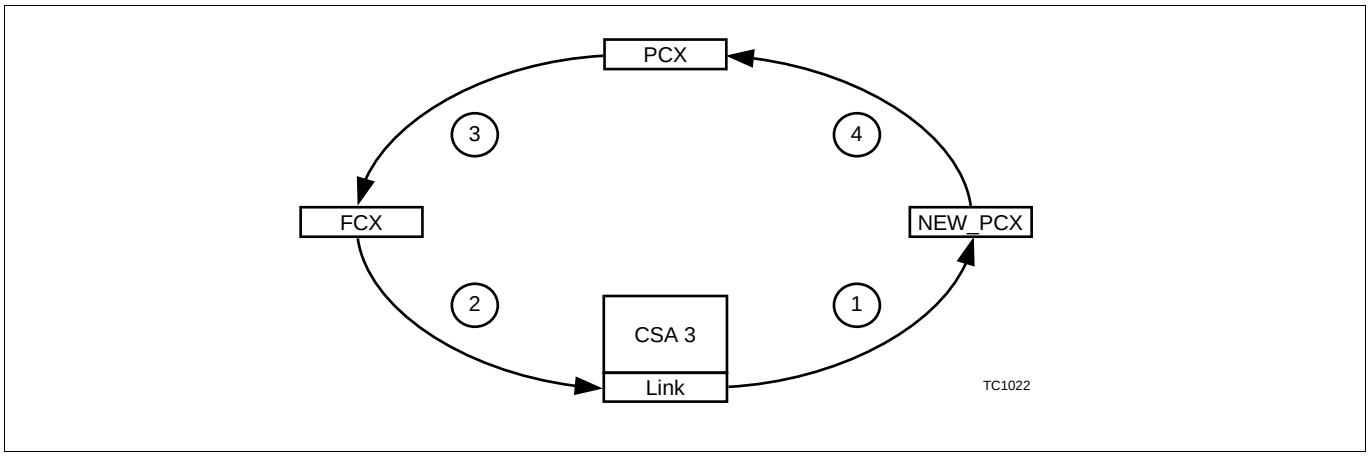


Figure 19 CSA and Processor SFR Updates on a Context Restore Process

The processor SFRs and CSAs now look as shown in **Figure 20**. The restored context is then written into the upper or lower context registers.

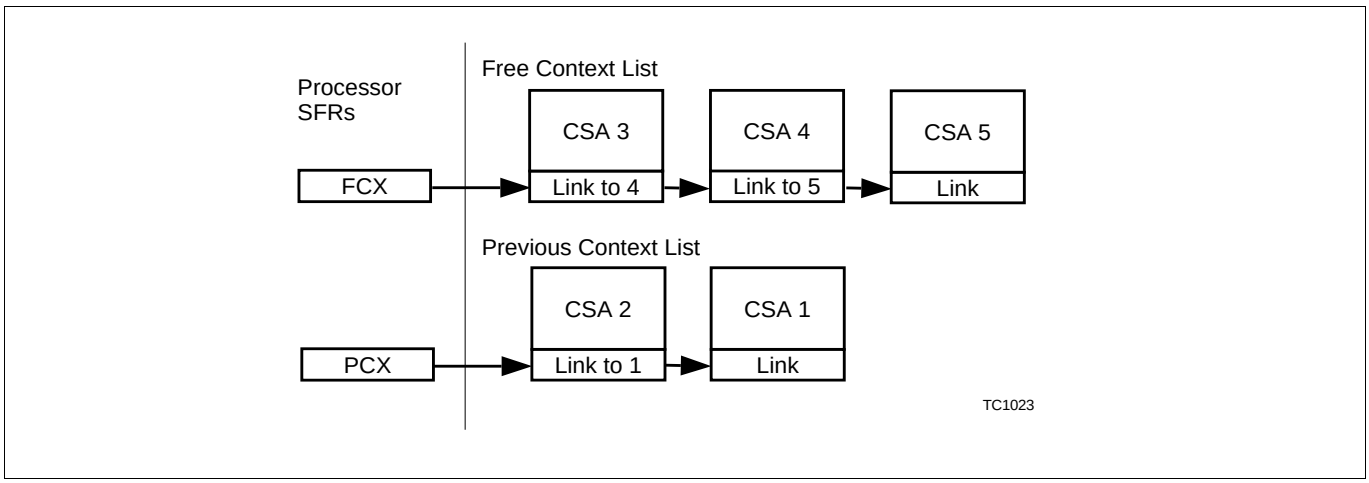


Figure 20 CSAs and Processor State After Context Restore

4.8 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**Page 12**.
- PCX: Previous Context Pointer**Page 13**.
- LCX: Free CSA List Limit Pointer**Page 14**.

Each pointer consists of two fields:

- A16-bit offset.
- A 4-bit segment specifier.

Table 21 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 64 KByte CSAs.

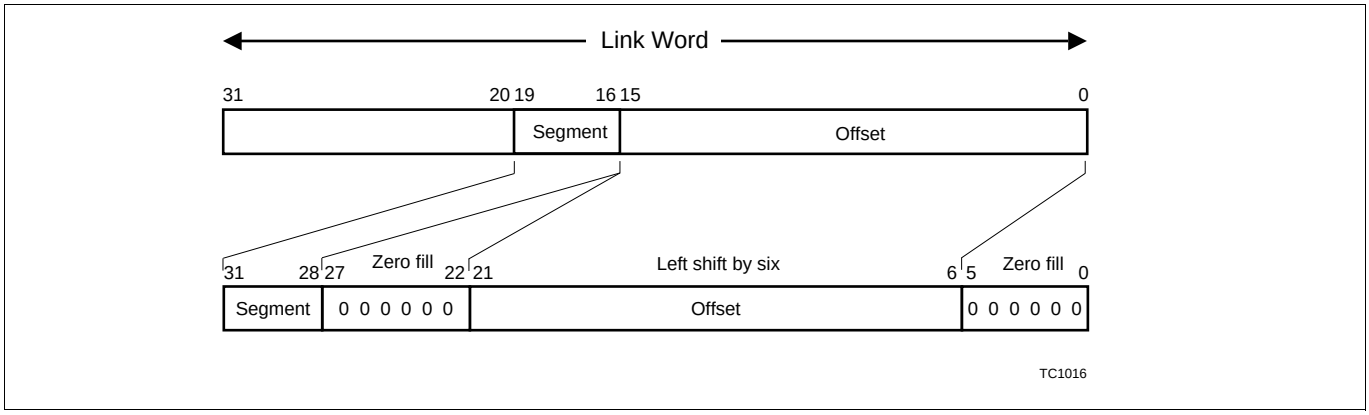


Figure 21 Generation of the Effective Address of a Context Save Area (CSA)

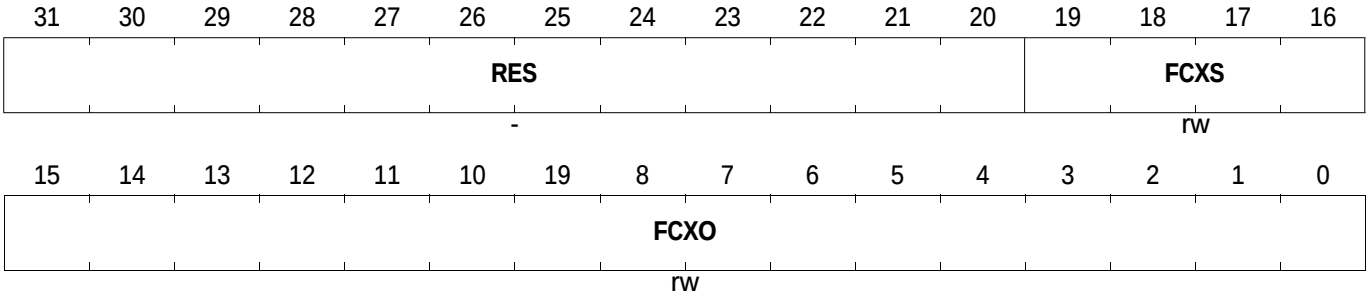
*Note: See **"Context Save Area" on Page 2** for additional constraints on the Effective Address (EA).*

4.8.1 Registers

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
Free CSA List Head Pointer **($FE38_H$)** **Reset Value: Implementation Specific**

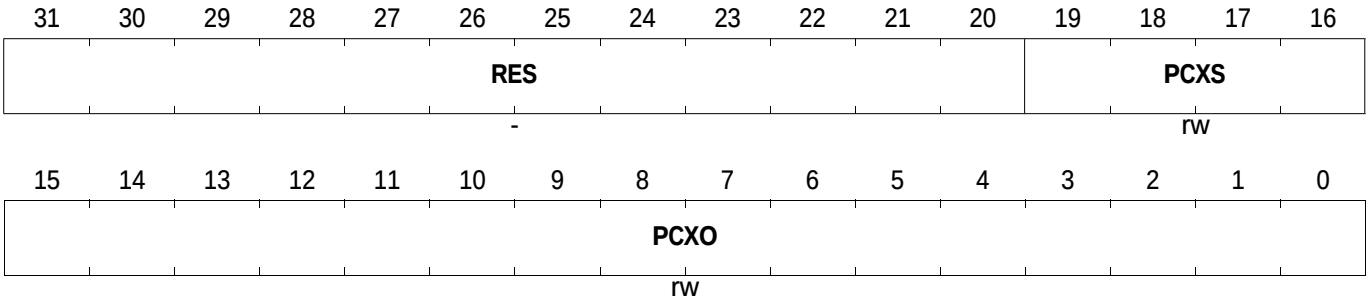


Field	Bits	Type	Description
RES	[31:20]	-	**Reserved**
FCXS	[19:16]	rw	**FCX Segment Address** Used in conjunction with the FCXO field.
FCXO	[15:0]	rw	**FCX Offset Address** The FCXO and FCXS fields together form the FCX pointer, which points to the next available CSA.

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
Previous Context Pointer Register **($FE00_H$)** **Reset Value: Implementation Specific**



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.

4.8.2 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; i.e. 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)" on Page 8****.*

Free CSA List Limit Pointer Register (LCX)

LCX
Free CSA List Limit Pointer **($FE3C_H$)** **Reset Value: Implementation Specific**

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.
LCXO	[15:0]	rw	**LCX Offset** The LCXO and LCXS fields form the pointer LCX, which points to the last available CSA.

4.9 Accessing CSA Memory Locations

Implementations may internally buffer context information to increase performance. To ensure memory coherency, a DSYNC instruction must be executed prior to any access to an active CSA memory location. The DSYNC instruction forces all internally buffered CSA register state to be written to memory.

4.10 Context Save Area Placement

Context Save Areas (CSAs) may not be placed in memory segments which have the peripheral space attribute (**Section 1.2.1**), 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 documentation 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 DMA channels. This chapter describes the interrupt processing capabilities of the CPU including the interrupt prioritisation scheme and access to the vector table.

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 Control Unit (ICU) to prioritise between multiple concurrent interrupt requests. The SRPN of the winning request is supplied to the CPU as a Pending Interrupt Priority Number (PIPN) along with an request trigger. The CPU decides whether to accept a requested interrupt by comparing the PIPN with its Current CPU Priority Number (CCPN). If the CPU decides to accept the requested interrupt it responds with an Interrupt Acknowledge and the returns the priority number of the taken interrupt. The ICU will then clear down the requesting interrupt source.

5.1.1 ICU Interrupt Control Register (ICR)

The ICU 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 and acknowledges the ICU, which in turn sends acknowledgement back to the pending interrupt request.

Several conditions could block the CPU from immediately responding to the interrupt request generated by the ICU. These are:

- The interrupt system is globally disabled (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.

5.1.3 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.
- 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 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 (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 SYSCON register. PSW.S = SYSCON.IS.
- Write permission to global registers A[0], A[1], A[8], A[9] is disabled: PSW.GW = 0.
- The interrupt system is globally disabled: 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 saved into 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 globally disabled and the current CPU priority (CCPN) set to the priority (PIPN) of the interrupt being serviced. It is up to the user to enable the 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” on Page 5**.

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 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, 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 is an exception and must be followed by an ISYNC instruction.

5.2 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.

5.3 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 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 in the program counter. The CPU begins executing instructions at this address in the Interrupt Vector Table. The code 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 store the initial portion of the ISR, such as a jump instruction that vectors the CPU to the rest of the ISR elsewhere in memory.

The Base of Interrupt Vector Table register (BIV) 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 to 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 ORed 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 [14:5] (VSS=0) or bits[12:3] (VSS=1) of register BIV are set to 0.

```
if (BIV.VSS == 1'b0)
 ISR_Entry_PC = {BIV[31:1],1'b0} | {PIPN<<5};
else
 ISR_Entry_PC = {BIV[31:1],1'b0} | {PIPN<<3};
```

If an interrupt handler is very short it may fit entirely within the words available in the vector code segment. 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" on Page 5**

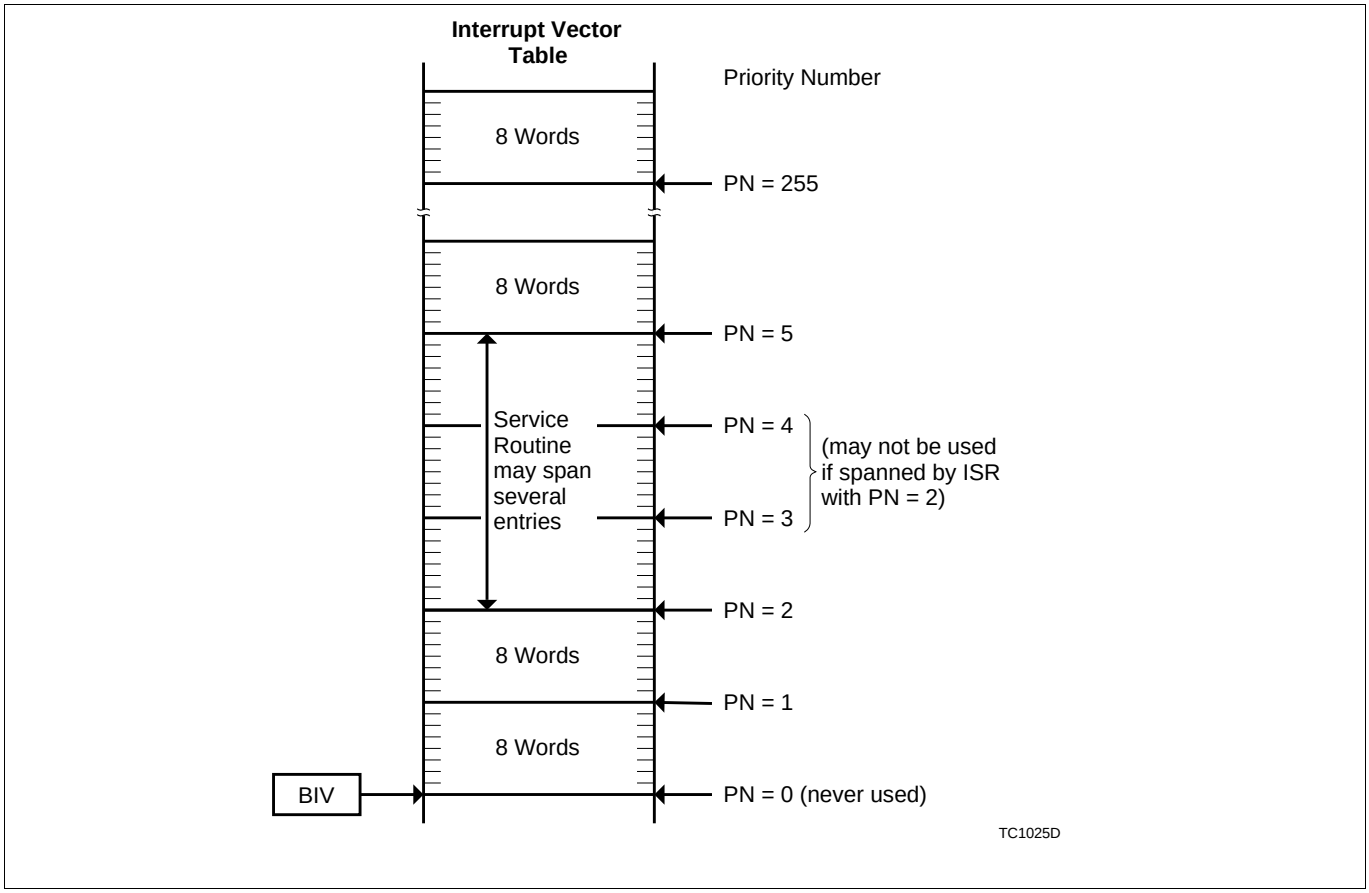


Figure 22 Interrupt Vector Table (VSS=0)

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.

5.4 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.

5.4.1 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 Page 3**. Spanning eliminates the need of a jump to the rest of the interrupt handler if it would not fit into the available 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 22Page 3**, vector locations three and four are covered through the service routine for 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, but the size of the vector table must be adjusted accordingly.

5.4.2 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 globally disabled 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; i.e. 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. **Figure 23** shows an example.

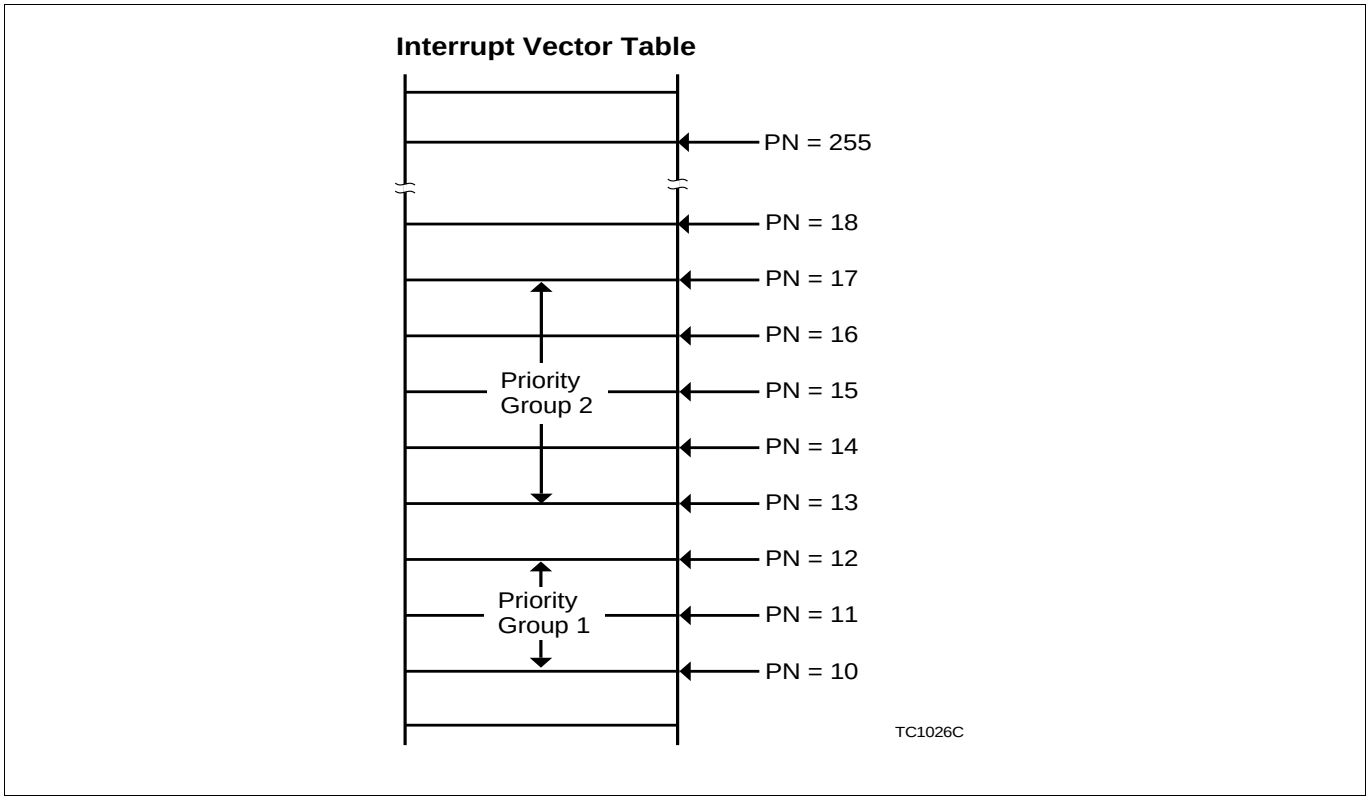


Figure 23 Interrupt Priority Groups

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; i.e. 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.

5.4.3 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 via 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.

5.4.4 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.

5.4.5 Interrupt Control Registers

Two CSFRs support interrupt handling:

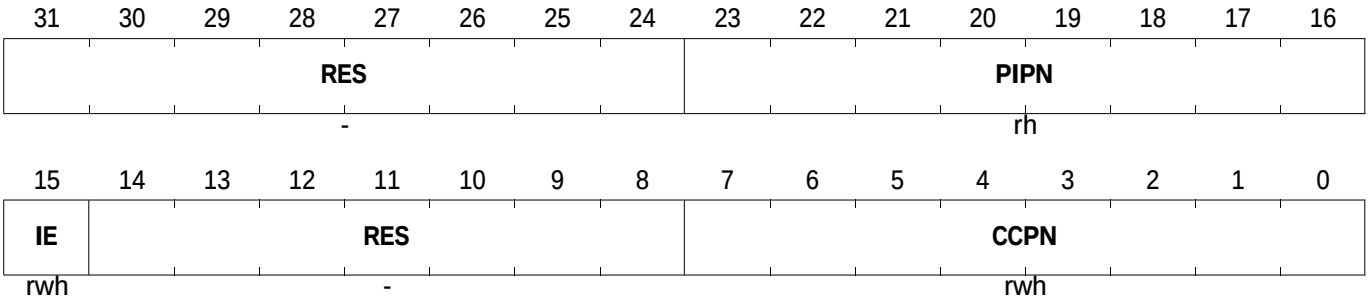
- ICR: Interrupt Control Register**Page 8**
- BIV: Base Interrupt Vector Table Pointer**Page 10**

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” on Page 1**.

ICU Interrupt Control Register (ICR)

The ICU Interrupt Control register is defined as follows:

ICR
ICU Interrupt Control **($FE2C_H$)** **Reset Value: $0000\ 0000_H$**



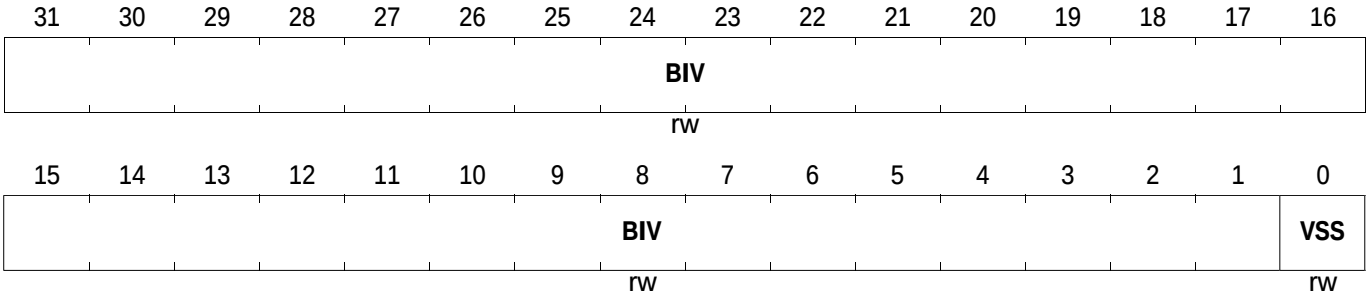
Field	Bits	Type	Function
RES	[31:24]	-	**Reserved**
PIPN	[23:16]	rh	**Pending Interrupt Priority Number** A read-only bit field that is updated by the ICU at the end of each interrupt arbitration process. It indicates the priority number of the pending service request. ICR.PIPN is set to 0 when no request 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	**Global Interrupt Enable Bit** The interrupt enable bit globally enables the CPU service request system. Whether a service request is delivered to the CPU depends on the individual Service Request Enable Bits (SRE) in the SRNs, and the current state of the CPU. 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, MTCR, and BISR instructions. 0 : Interrupt system is globally disabled. 1 : Interrupt system is globally enabled.

Field	Bits	Type	Function
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.

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 eight bytes or 32 bytes between the individual entries in the vector table dependent on the vector spacing selected by the VSS bit.

BIV
Base Interrupt Vector Table Pointer **($FE20_H$)** **Reset Value: Implementation Specific**



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.

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 subhandler 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, summarising 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 8 Supported Traps

TIN	Name	Synch. / Asynch.	HW / SW	Definition	Page
Class 0 — MMU					
0	VAF	Synch.	HW	Virtual Address Fill.	Page 6
1	VAP	Synch.	HW	Virtual Address Protection.	Page 6
Class 1 — Internal Protection Traps					
1	PRIV	Synch.	HW	Privileged Instruction.	Page 6
2	MPR	Synch.	HW	Memory Protection Read.	Page 6
3	MPW	Synch.	HW	Memory Protection Write.	Page 6
4	MPX	Synch.	HW	Memory Protection Execution.	Page 6
5	MPP	Synch.	HW	Memory Protection Peripheral Access.	Page 7
6	MPN	Synch.	HW	Memory Protection Null Address.	Page 7
7	GRWP	Synch.	HW	Global Register Write Protection.	Page 7
Class 2 — Instruction Errors					
1	IOPC	Synch.	HW	Illegal Opcode.	Page 7
2	UOPC	Synch.	HW	Unimplemented Opcode.	Page 7
3	OPD	Synch.	HW	Invalid Operand specification.	Page 7
4	ALN	Synch.	HW	Data Address Alignment.	Page 7
5	MEM	Synch.	HW	Invalid Local Memory Address.	Page 7
Class 3 — Context Management					
1	FCD	Synch.	HW	Free Context List Depletion (FCX = LCX).	Page 8
2	CDO	Synch.	HW	Call Depth Overflow.	Page 9
3	CDU	Synch.	HW	Call Depth Underflow.	Page 9

Table 8 Supported Traps (cont'd)

TIN	Name	Synch. / Asynch.	HW / SW	Definition	Page
4	FCU	Synch.	HW	Free Context List Underflow (FCX = 0).	Page 9
5	CSU	Synch.	HW	Call Stack Underflow (PCX = 0).	Page 9
6	CTYP	Synch.	HW	Context Type (PCXI.UL wrong).	Page 9
7	NEST	Synch.	HW	Nesting Error: RFE with non-zero call depth.	Page 9
Class 4 — System Bus and Peripheral Errors					
1	PSE	Synch.	HW	Program Fetch Synchronous Error.	Page 10
2	DSE	Synch.	HW	Data Access Synchronous Error.	Page 10
3	DAE	Asynch.	HW	Data Access Asynchronous Error.	Page 10
4	CAE	Asynch	HW	Coprocessor Trap Asynchronous Error.	Page 10
5	PIE	Synch	HW	Program Memory Integrity Error.	Page 11
6	DIE	Asynch	HW	Data Memory Integrity Error.	Page 11
7	TAE	Asynch	HW	Temporal Asynchronous Error	Page 11
Class 5— Assertion Traps					
1	OVF	Synch.	SW	Arithmetic Overflow.	Page 11
2	SOVF	Synch.	SW	Sticky Arithmetic Overflow.	Page 11
Class 6 — System Call[1)]					
	SYS	Synch.	SW	System Call.	Page 11
Class 7 — Non-Maskable Interrupt					
0	NMI	Asynch.	HW	Non-Maskable Interrupt.	Page 11

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 via 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)” on Page 11**.

6.1.5 Unrecoverable Traps

An unrecoverable trap is one from which software can not recover; i.e. 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)” on Page 9** 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 or software interrupts. A trap does not change the CPU interrupt priority, so the ICR.CCPN field is not updated. See **“Exception Priorities” on Page 12**.

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 segments, 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 segment. If it does not fit the vector code segment 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 behaviour for the FCD trap is described in **“FCD - Free Context list Depletion (TIN 1)” on Page 8**.

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. It can be assigned to 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.

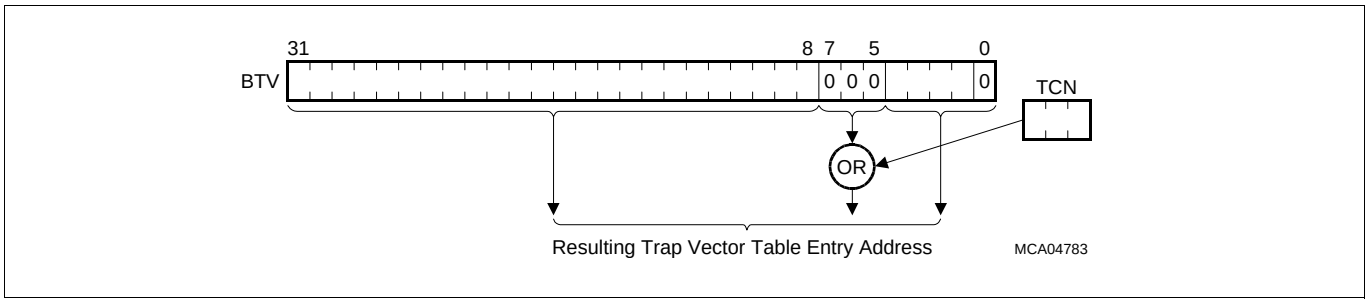


Figure 24 Trap Vector Table Entry Address Calculation

6.2.5 Initial State upon a Trap

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

- The upper context is saved.
- The return address in A[11] is updated.
- 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 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 SYSCON register. PSW.S = SYSCON.TS.
- Write permission to global registers A[0], A[1], A[8], A[9] is disabled: PSW.GW = 0.
- The interrupt system is globally disabled: 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 (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 SYSCON register: PSW.S = SYSCON.TS.
- The interrupt system is globally disabled: 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 **Table 8 "Supported Traps" on Page 1**.

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 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" on Page 1** chapter 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 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, LDMST, SWAP or ST.T 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, LDMST, SWAP or ST.T 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 is configured to be a peripheral segment. See **"Physical Memory Attributes (PMA)" on Page 3**.

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 signalling 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.

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 via 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 other cases where operands are invalid.

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" on Page 4**, 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" on Page 9** 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 behaviour 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).

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 SYSCON (system 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 **“System Control Register (SYSCON)” on Page 13**.

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 (i.e. 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 asychronous exception may be lost whan 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 (SMTs). 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; i.e. 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[1] occurred because of an instruction fetch.
- An instruction fetch targets a segment that does not have the code fetch property. See **"Physical Memory Attributes (PMA)" on Page 3**.

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" on Page 5**) 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.

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 unimplemented coprocessor instructions and arithmetic errors (as found in the Floating Point Unit for example).

CAE is shared amongst all coprocessors in a given system. A trap handler must therefore inspect all coprocessors to determine the cause of a trap.

1) 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.

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 localise 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 localise the error to the access width of the operation. DIE traps raised durring 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 b e 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 not fixed, but 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" on Page 1**.

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.

Table 9 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
6	PIE
Instruction Format Traps	
7	IOPC
8	OPD
9	UOPC
Instruction Traps	
10	Breakpoint trap or halt - BBM (Trigger on Address, MxCR, Debug)
11	PRIV
12	GRWP
13	SYS

Table 9 Synchronous Trap Priorities (cont'd)

Priority	Type of Trap
Context Traps	
14	FCD
15	FCU (Synchronous)
16	CSU
17	CDO
18	CDU
19	NEST
20	CTYP
Data Memory Access Traps	
21	MEM (Data address)
22	ALN
23	MPN
24	VAF-D
25	VAP-D
26	MPP
27	MPR
28	MPW
29	DSE
General Data Traps	
30	SOVF
31	OVF
32	Breakpoint trap or halt - BAM

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

Table 10 Asynchronous Trap Priorities

Priority	Asynchronous Traps
1	NMI
2	DAE[1)]
3	CAE
4	TAE
5	DIE

1) DAE is used for store errors.

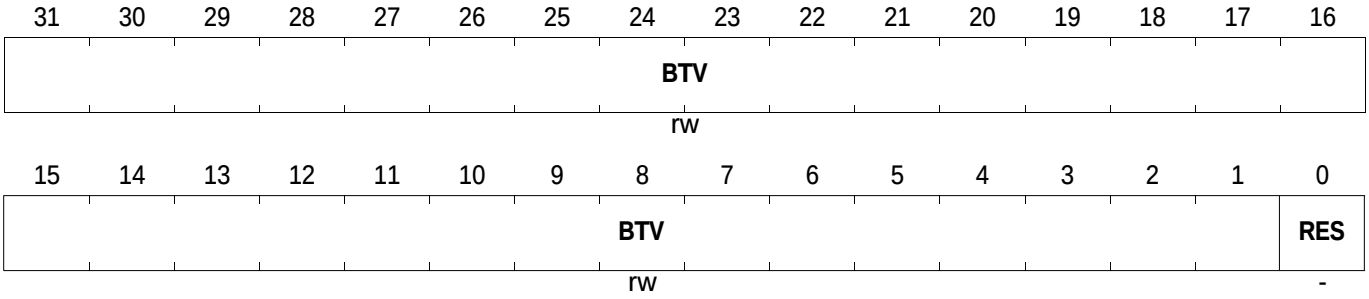
6.5 Trap Control Registers

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
Base Trap Vector Table Pointer **($FE24_H$)** **Reset Value: Implementation Specific**



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**

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

Program Synchronous Error Trap Register (9200$_H$) Reset Value: Implementation Specific

Field	Bits	Type	Description
Implementation Specific	[31:0]	-	**Implementation Specific**

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
Data Synchronous Error Trap Register (9010_H) Reset Value: Implementation Specific

Field	Bits	Type	Description
Implementation Specific	[31:0]	-	**Implementation Specific**

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
Data Asynchronous Error Trap Register (9018$_H$) Reset Value: Implementation Specific

Field	Bits	Type	Description
Implementation Specific	[31:0]	-	**Implementation Specific**

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
Data Error Address Register ($901C_H$) Reset Value: 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.

This chapter should be read in conjunction with **"Scenarios for Memory Integrity Error Mitigation" on Page 1**.

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 catagorised 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 and TIN 5.

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

Note: Implementation specific registers that 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 whenever an uncorrectable memory integrity error is detected in a data access to a local memory. The trap is of Class 4 and 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 localise 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 Registers

7.3.1 Error Information 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.

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
Program Integrity Error Trap Register (9214$_H$) Reset Value: 0000 0000$_H$

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

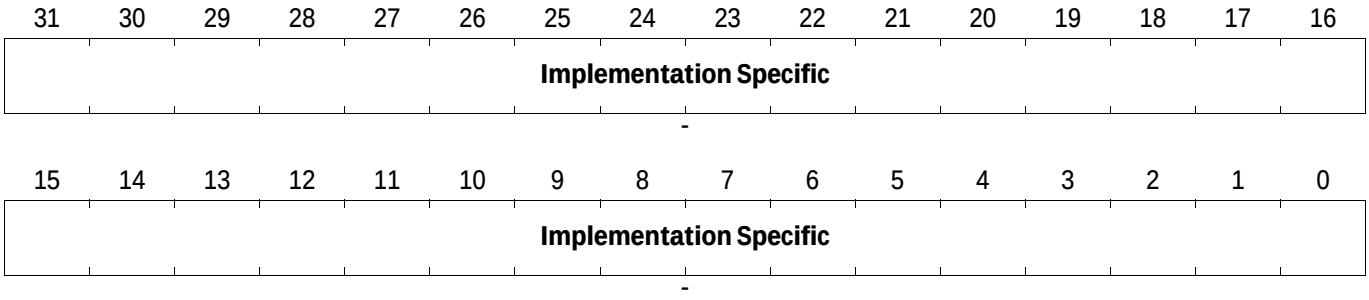
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Implementation Specific															
-															

Field	Bits	Type	Description
Implementation Specific	[31:0]	-	Implementation Specific

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
Program Integrity Error Address Register (9210_H) Reset Value: $0000\ 0000_H$

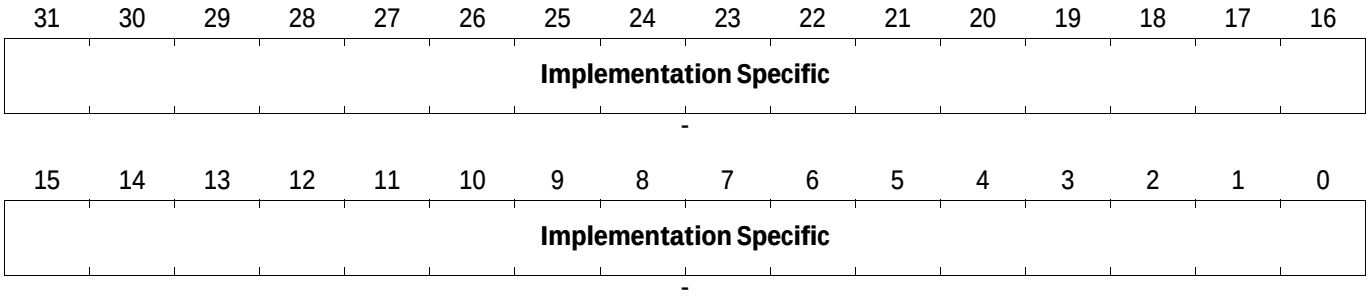


Field	Bits	Type	Description
Implementation Specific	[31:0]	-	**Implementation Specific**

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
Data Integrity Error Trap Register (9024$_H$) Reset Value: 0000 0000$_H$

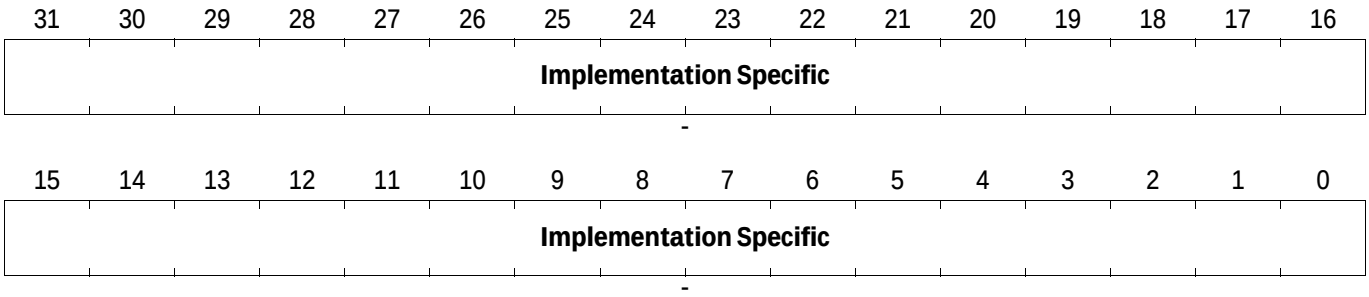


Field	Bits	Type	Description
Implementation Specific	[31:0]	-	**Implementation Specific**

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
Data Integrity Error Address Register (9020_H) Reset Value: $0000\ 0000_H$



Field	Bits	Type	Description
Implementation Specific	[31:0]	-	**Implementation Specific**

7.4 Summary

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

- A correctable error and an increment of one of the CCPIE counters
- An uncorrectable error triggering a PIE trap

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

- A correctable error and an increment of one of the CCDIE counters
- An uncorrectable error triggering a DIE trap

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

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 256MByte 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 will result in a trap.

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

Table 11 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 256MBytes 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 Cached 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 fetches to the memory.

8.3.1.2 Non-cached 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.

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 12 Default Memory Access Types for all Segments

Segment	Attributes
FH	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

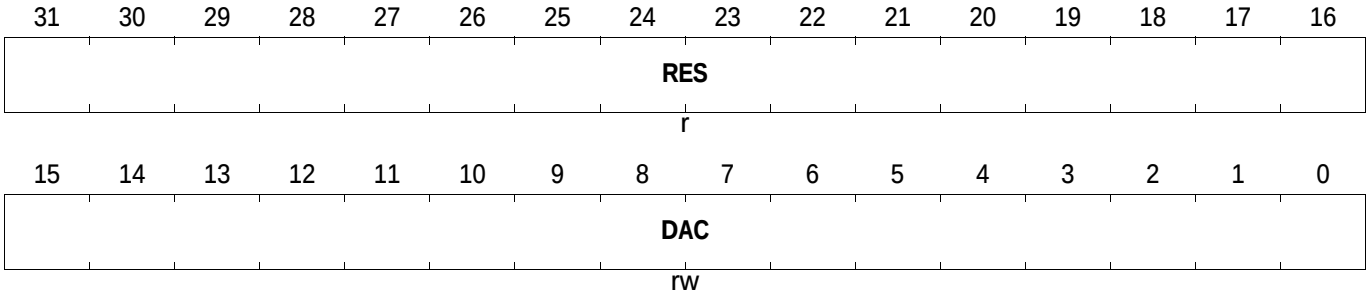
8.4.1 Programmable Memory Access Register-0 (PMA0)

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
Programmable Memory Access Register-0 (8100_H) Reset Value: 0000 0300



Field	Bits	Type	Description
RES	[31:16]	r	Reserved
DAC	[15:0]	rw	Data Access Cacheability - Implementation defined.

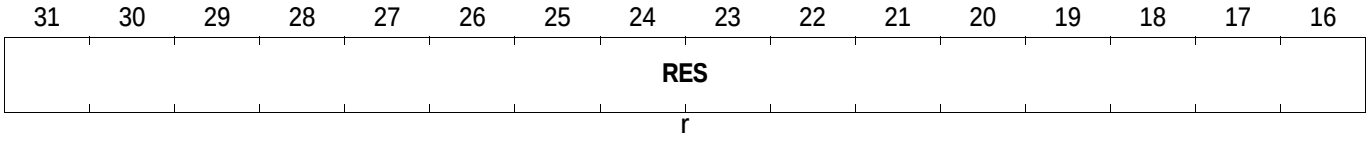
8.4.2 Programmable Memory Access Register1 (PMA1)

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
Programmable Memory Access Register-1 (8104_H) Reset Value: 0000 0300



PMA1
Programmable Memory Access Register-1 (8104_H) Reset Value: 0000 0300

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CAC															

Field	Bits	Type	Description
RES	[31:16]	r	Reserved
CAC	[15:0]	rw	Code Accesses Cacheability - Implementation defined

8.4.3 Programmable Memory Access Register2 (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
Programable Memory Access Register-2 (8108_H) Reset Value: 0000 C000

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

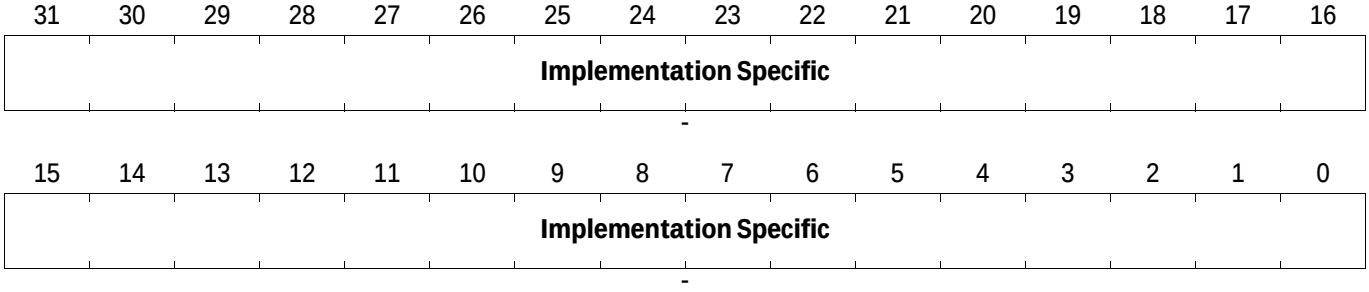
8.4.4 Program Memory Configuration Registers (PCON0, PCON1, PCON2)

TriCore Implementations may control and provide information on the status and configuration of the program cache and scratch memories via 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.

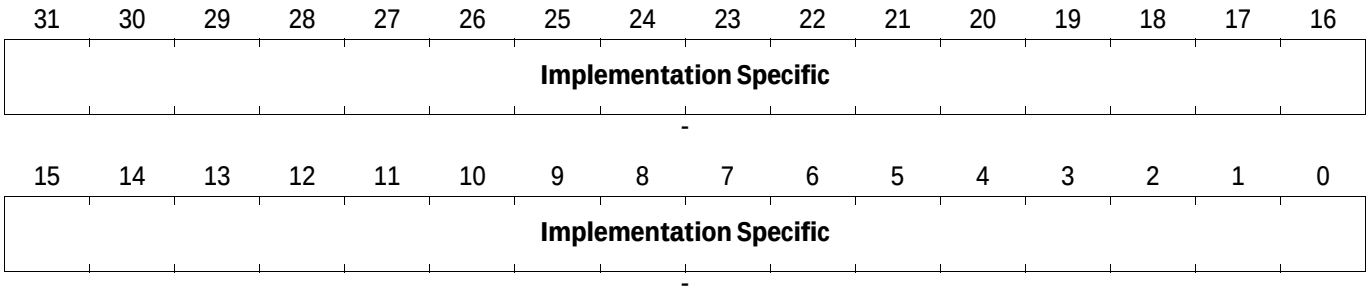
Implementations may ENDINIT protect these registers.

PCON0
Program Memory Configuration Register 0 ($920C_H$) Reset Value: Implementation Specific



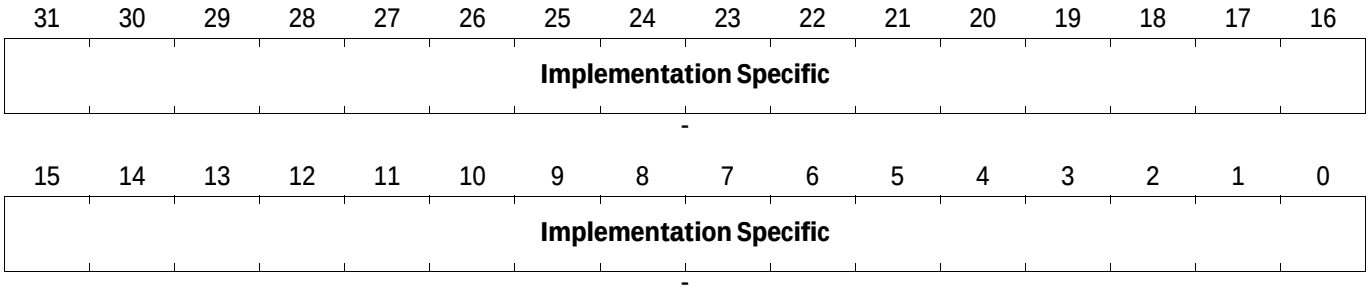
Field	Bits	Type	Description
Implementation Specific	[31:0]	-	Implementation Specific

PCON1
Program Memory Configuration Register 1 (9204_H) Reset Value: Implementation Specific



Field	Bits	Type	Description
Implementation Specific	[31:0]	-	Implementation Specific

PCON2
Program Memory Configuration Register 2 (9208_H) Reset Value: 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 via 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.

DCON0

Data Memory Configuration Register 0 (9040_H) Reset Value: Implementation Specific

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

DCON1

Data Memory Configuration Register 1 (9008_H) Reset Value: Implementation Specific

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

DCON2

Data Memory Configuration Register 2 (9000_H) Reset Value: Implementation Specific

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															

DCON2

Data Memory Configuration Register 2 (9000_H) Reset Value: Implementation Specific

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

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 FPU is an optional component in TriCore configurations. It need not be present in every system that uses the core.

The optional FPU is an IEEE-754 compatible floating-point unit to accompany the TriCore instruction set.

9.1 Functional Overview

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

- Floating-point add, subtract, multiply, MAC, and divide instructions.
- Conversion to or from IEEE-754 single precision format from or to TriCore signed and unsigned integers and 32-bit signed fractions (Q31 format).
- QSEED.F instruction 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 rounding modes are implemented.
- Asynchronous traps can be generated on selected IEEE-754 exceptions (TriCore 1.3.1 and TriCore 1.6).

Restrictions

The FPU has the following restrictions and usage limitations:

- Only IEEE-754 single precision format is supported.
- IEEE-754 denormalized numbers are not supported for arithmetic operations.
- IEEE-754 compliant remainder function cannot be implemented using FPU instructions because of the effects of multiple rounding when using a sequence of individually rounded instructions.
- Fused multiply-and-accumulate operations (MACs) are not part of the IEEE-754 standard. 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 standard is not achieved because denormal numbers are not supported.
- If no FPU is present, then FPU instructions will cause a UOPC (unimplemented opcode) trap.

9.2 IEEE-754 Compliance

9.2.1 IEEE-754 Single Precision Data Format

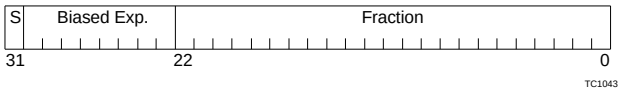


Figure 25 Single Precision IEEE-754 Floating-Point Format

The single precision IEEE-754 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. **Table 13** shows the different types of number representation in IEEE-754 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 13 IEEE-754 Single Precision Representation Types

Condition	Represented Value	Description
0 < e < 255	(-1)s*2(e-127)*1.f	Normal number.
e == 0 AND f != 0	(-1)s*2(-126)*0.f	Denormal number.
e == 0 AND f == 0	(-1)s*0	Signed zero.
s == 0 AND e == 255 AND f == 0	+ ∞	+ infinity.
s == 1 AND e == 255 AND f == 0	- ∞	– infinity.
e == 255 AND f != 0 AND f[22] == 0		Signalling NaN[1].
e == 255 AND f != 0 AND f[22] == 1		Quiet NaN[1].

1) IEEE-754 does not define how to distinguish between signalling NaNs and quiet NaNs, but bit[22] 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.

9.2.2 Denormal Numbers

Denormal numbers are not supported for arithmetic operations. With the exception of the CMP.F instruction, all instructions replace denormal operands with the appropriately signed zero before computation. Following computation, if a denormal number would otherwise be the result, it is replaced with the appropriately signed zero.

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

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

This is replaced with:

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

This procedure has a subtle effect on underflow; see **Round to Nearest: Denormals and Zero Substitution, page 9-7**.

Denormal numbers are supported only by the CMP.F instruction which makes comparisons of denormal numbers in addition to identifying denormal operands.

9.2.3 NaNs (Not a Number)

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

When invalid operations are performed (including operations with a signalling 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 **Invalid Operations and their Quiet NaN Results, page 9-8**.

IEEE-754 suggests that quiet NaNs should be propagated so that the result of an instruction receiving a quiet NaN as an operand (with no signalling 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 signalling NaN operands is always the quiet NaN $7FC00000_H$.

9.2.4 Underflow

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

IEEE-754 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 allows this to be detected either before or after rounding.
- There must be a loss of accuracy in the stored result.

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

Denormalization 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 denormalized to fit into IEEE-754 format and the resultant denormalized number differs from the normalized result with unbounded exponent range, then a denormalization 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 denormal 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 denormal that is replaced with zero or rounded up. Therefore underflow detection can be simplified to tiny number detection alone; i.e. any non-zero unrounded number whose magnitude is $< 2^{-126}$.

9.2.5 Fused MACs

Fused multiply-and-accumulate operations (MACs) are not supported by the IEEE-754 standard. Using FPU MAC operations (MADD.F and MSUB.F) can give different results from using separate multiply (MUL.F) and accumulate (ADD.F or SUB.F) operations because the result is only rounded once at the end of a MAC.

9.2.6 Traps

IEEE-754 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 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 are asynchronous and therefore are not IEEE-754 compliant traps. Since IEEE-754 traps are optional this does not cause any IEEE-754 non compliance.

9.2.7 Software Routines

Operations required for IEEE-754 compliance, but not implemented in the FPU instruction set, are detailed in **Table 14**.

Table 14 IEEE-754 Operations Requiring Software Implementation

IEEE-754 Operation	Suggested Implementation
Square root	Newton-Raphson using QSEED.F instruction.
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. rem = x - (d * (FTOI(x/d)[1)])) rem: remainder x: dividend d: divisor
Round to integer in Floating-point format	ITOF(FTOI(x)).
Convert between binary and decimal	-

1) Round to nearest.

9.3 Rounding

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

Table 15 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 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** 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: Even, page 9-6**, 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 denormal numbers with the appropriately signed 0.

IEEE-754 does not specify the MAC instructions (MADD.F and MSUB.F) that combine multiplication and addition in a single operation. 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. It is therefore possible that the result from a MADD.F instruction will differ from the result that would be obtained using the same operands in a MUL.F followed by an ADD.F.

Rounding Mode Restored

The rounding mode is not restored on a RET (Return From Call) instruction. The rounding mode is restored on an RFE (Return From Exception) instruction or an RFM (Return From Monitor) instruction.

9.3.1 Round to Nearest: 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 (i.e. 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 0000B * 20

This is half way between:

1.0000 0000 0000 0000 0000 000B * 21

and:

1.111 1111 1111 1111 1111 1111B * 20

The result with the larger exponent is returned.

9.3.2 Round to Nearest: Denormals and Zero Substitution

Following computation, results are first rounded to IEEE-754 representable numbers and then the appropriately signed zero is substituted for any denormal 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 normal number (1.000 … 000 * 2^{-126}) and the largest representable positive IEEE-754 denormal number (0.111 … 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 denormal number, then the result is rounded to the denormal value. Zero is then substituted for the denormal result.

The FPU architecture cannot produce denormal results, however the concept of denormal 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 … 000 * 2^{-126}) or (0). Such an implementation would mean that all unrounded results between (+1.000 … 000 * 2^{-126}) and (+0.100 … 000 * 2^{-126}) would be rounded to the smallest representable positive IEEE-754 normal number.

9.3.3 Round Towards ± ∞: Denormals and Zero Substitution

Following computation results are first rounded to IEEE-754 representable numbers, then the appropriately signed zero is substituted for any denormal results that may have occurred. See **Denormal Numbers, page 9-2**.

According to the IEEE-754 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 denormal 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.

9.4 Exceptions

The FPU implements all five IEEE-754 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 exception occurs, however IEEE-754 compliant traps are not implemented, see **Section 9.5 Asynchronous Traps** (**Page 10**).

IEEE-754 Exception Flags

The IEEE-754 exception flags are stored as part of the PSW register as shown in the following table. In accordance with IEEE-754, 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 16 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 exception flags are sticky, it can be impossible to tell if an 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 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 exceptions to occur. If any of the IEEE-754 exceptions (FI, FV, FZ, FU, FX) have occurred during that instruction, FS is also asserted.

Note: UPDFL can assert IEEE-754 exceptions without asserting FS.

FI - Invalid Operation

FI is asserted in three circumstances:

- When a signalling NaN (see **NaNs (Not a Number), page 9-3**) is an operand for a FPU instruction.
- For invalid operations such as QSEED.F ($^a1/\div x$) of a negative number.
- Conversions from floating-point to other 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 signalling NaN or invalid operation, the result is a quiet NaN.

Table 17 Invalid Operations and their Quiet NaN Results

Invalid Operation	Quiet NaN
Signalling NaN operand for arithmetic instructions.[1)]	$7FC00000_H$[2)]
Signalling NaN operand for CMP.F instruction.	n.a.[)]
ADD.F with + ∞ and - ∞ as operands.	$7FC00001_H$
SUB.F with (+ ∞ and + ∞) or (- ∞ and - ∞) as operands.	$7FC00001_H$

Table 17 Invalid Operations and their Quiet NaN Results (cont'd)

Invalid Operation	Quiet NaN
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]
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]
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 signalling. See **NaNs (Not a Number), page 9-3**.
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; '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.

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 ± ∞ as an operand is defined as being exact, except for invalid operations where FI is asserted. Therefore for ± ∞/ ± 0 FZ is not asserted, the appropriately signed ∞ is returned as the result with no other exceptions occurring.

FU - Underflow

As discussed in **Underflow, page 9-4**, underflow is detected and so FU is asserted, when the unrounded result is smaller in magnitude than the smallest representable normal number (2^{-126}).

The Q31TOF instruction can cause an underflow as well as the arithmetic instructions ADD.F, SUB.F, MUL.F, MADD.F, MSUB.F, and DIV.F.

The return result for instructions flagging an underflow are complicated by the way that FPU treats denormal numbers. This is described in detail in **Round to Nearest: Denormals and Zero Substitution, page 9-7**.

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 denormalization return result rules are followed.

9.5 Asynchronous Traps

The FPU can be configured such that a trap is signalled to the TriCore core when an FPU instruction causes an IEEE-754 exception. The trap generated is a Co-Processor Asynchronous Error (CAE), Trap Class 4 - TIN 4. FPU CAE traps should not be confused with the synchronous exception traps optional to IEEE-754 which allow software routines to correct arithmetic overflow or underflow.

The FPU CAE trap is intended for debug 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 signalled 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 (FIE, FVE, FZE, 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 register. 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.

9.6 FPU CSFR Registers

The FPU CSFR registers are used to store the details of instructions causing traps.

FPU Trap Control Register

FPU_TRAP_CON
Trap Control Register **($A000_H$)** **Reset value: $0000\ 0000_H$**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RES	FI	FV	FZ	FU	FX	RES			FIE	FVE	FZE	FUE	FXE	RES	
-	rh	rh	rh	rh	rh	-			rw	rw	rw	rw	rw	-	

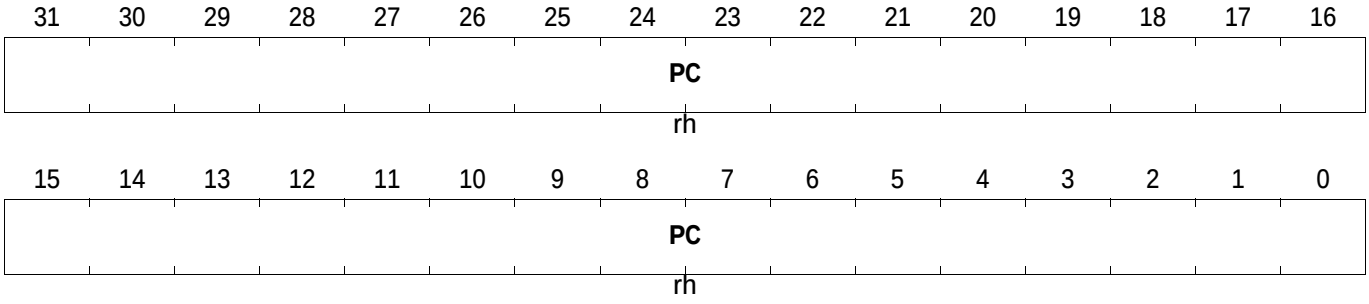
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES						RM		RES						TCL	TST
-						rh		-						w	rh

Field	Bits	Type	Description
RES	31	-	**Reserved**
FI	30	rh	**Captured FI** Asserted if the captured instruction asserted FI. Only valid when TST is asserted.
FV	29	rh	**Captured FV** Asserted if the captured instruction asserted FV. Only valid when TST is asserted.
FZ	28	rh	**Captured FZ** Asserted if the captured instruction asserted FZ. Only valid when TST is asserted.
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	[25:23]	-	**Reserved**
FIE	22	rw	**FI Trap Enable** When set, an instruction generating an FI exception will trigger a trap.
FVE	21	rw	**FV Trap Enable** When set, an instruction generating an FV exception will trigger a trap.
FZE	20	rw	**FZ Trap Enable** When set, an instruction generating an FZ exception will trigger a trap.
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**

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.

FPU Trapping Instruction Program Counter Register

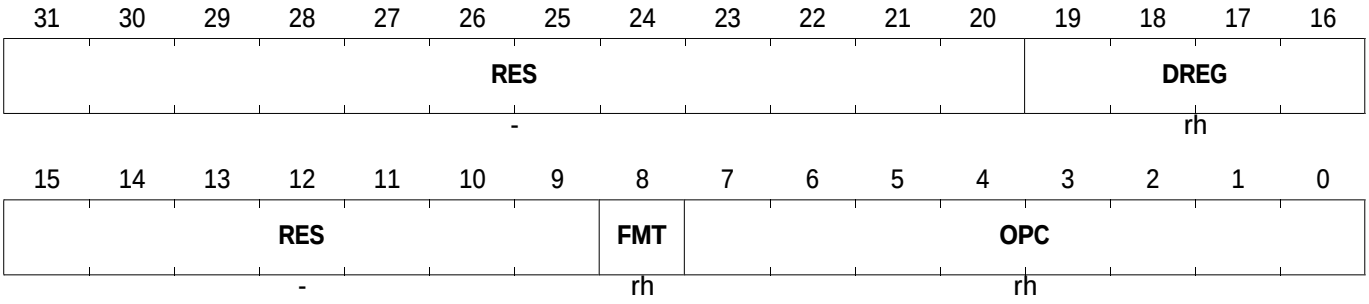
FPU_TRAP_PC
Trapping Instruction Program Counter **($A004_H$)** **Reset value: Implementation Specific**



Field	Bits	Type	Description
PC	[31:0]	rh	**Captured Program Counter** The program counter (virtual address) of the captured instruction. Only valid when FPU_TRAP_CON.TST is asserted.

FPU Trapping Instruction Opcode Register

FPU_TRAP_OPC
Trapping Instruction Opcode **($A008_H$)** **Reset value: Implementation Specific**



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. …$_H$ F_H : Data general purpose register 15. Only valid when FPU_TRAP_CON.TST is asserted.
RES	[15:9]	-	**Reserved**
FMT	8	rh	**Captured Instruction Format** The format of the captured instruction's opcode. 0 : RRR. 1 : RR. Only valid when FPU_TRAP_CON.TST is asserted.
OPC	[7:0]	rh	**Captured Opcode** The secondary opcode of the captured instruction. When FPU_TRAP_OPC.FMT=0 only bits [3:0] are defined. OPC is valid only when FPU_TRAP_CON.TST is asserted.

FPU Trapping Instruction Operand SRC1 Register

FPU_TRAP_SRC1
Trapping Instruction Operand **($A010_H$)** **Reset value: Implementation Specific**

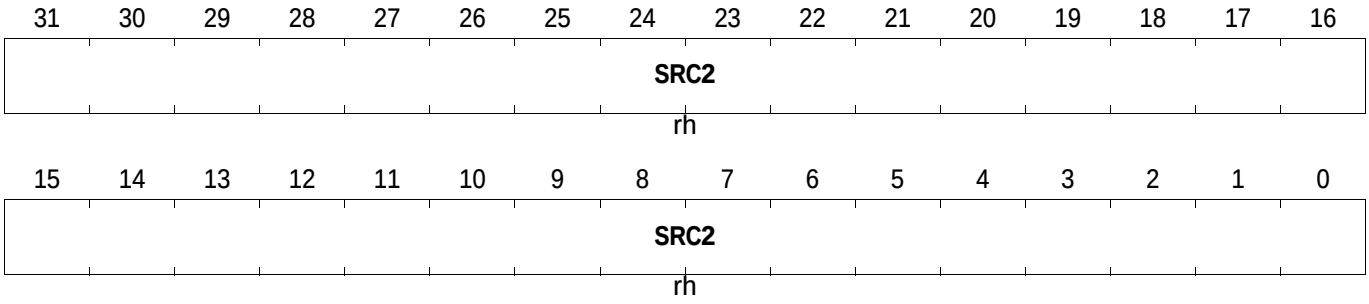
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SRC1															
rh															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRC1															
rh															

Field	Bits	Type	Description
SRC1	[31:0]	rh	**Captured SRC1 Operand** The SRC1 operand of the captured instruction. Only valid when FPU_TRAP_CON.TST is asserted.

FPU Trapping Instruction Operand SRC2 Register

FPU_TRAP_SRC2
Trapping Instruction Operand **($A014_H$)** **Reset value: Implementation Specific**



Field	Bits	Type	Description
SRC2	[31:0]	rh	**Captured SRC2 Operand** The SRC2 operand of the captured instruction. Only valid when FPU_TRAP_CON.TST is asserted.

FPU Trapping Instruction Operand SRC3 Register

FPU_TRAP_SRC3
Trapping Instruction Operand **($A018_H$)** **Reset value: Implementation Specific**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SRC3															
rh															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRC3															
rh															

Field	Bits	Type	Description
SRC3	[31:0]	rh	**Captured SRC3 Operand** The SRC3 operand of the captured instruction. Only valid when FPU_TRAP_CON.TST is asserted.

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 behaviour.

The protection system incorporates hardware mechanisms that protect user-specified memory ranges from unauthorized read, write, or instruction fetch accesses.

The protection hardware can also facilitate application debugging.

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” on Page 1**.

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 behaviour of User-1 mode may be overriden by the system control register).

For more information see **“Access Privilege Level Control (I/O Privilege)” on Page 8**.

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 small and low cost 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-bytes for data protection ranges and 32-bytes for code protection ranges

The three least significant bits of the Data Protection upper and lower bound registers are not writeable and always return zero.

The five least significant bits of the Code Protection upper and lower bound registers are not writeable 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 18 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 (code or data) crossing the boundary of a permitted region and a ‘not permitted’ region of memory. In this situation it is implementation defined (not architecturally defined) as to whether or not a memory protection trap is taken.

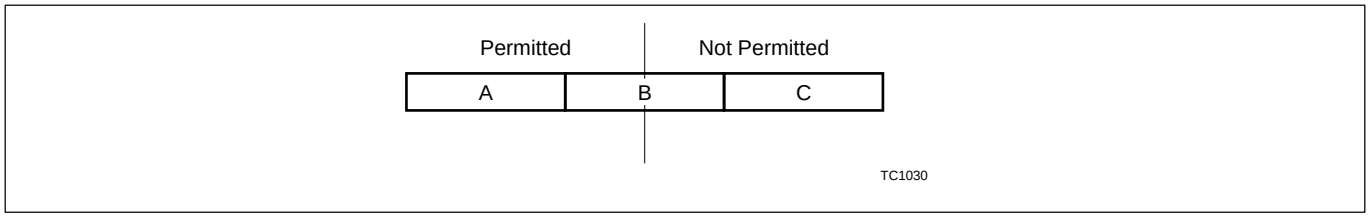


Figure 26 Protection Boundaries

Note: To ensure deterministic behaviour in all implementations of TriCore, a region at least twice the size of the largest memory accesses, minus one byte, should be left as a buffer between each memory protection region. Some implementations may require less spacing between buffers, 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 performed. The legality is determined by all of the following:

- The Protection Enable bit in the SYSCON register (SYSCON.PROTEN)
- The currently selected protection register set (PSW.PRS)
- The ranges selected in the protection register set
- The access permissions set for the ranges selected for the protection set

10.3.1 Protection Enable Bit

For the memory protection system to be active, the Protection Enable bit (SYSCON.PROTEN) must be set to one (SYSCON.PROTEN == 1).

If the memory protection system is disabled (SYSCON.PROTEN == 0), then any access to any memory address is permitted.

10.3.2 Set Selection

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 or Interrupt Service Routine (ISR).

The **PSW.PRS** field indicates the current Protection Register Set number.

10.3.3 Address Range

Data addresses (read and write accesses) are checked against the currently selected data address range table.

Instruction fetch addresses are checked against the currently selected code address range tables.

The mode entries for the data range table entries enable only read and write accesses, while the mode entries for the code range table entries enable only execute access.

In order for data to be read from program space, there must be an entry in the data address range table that covers the address being read. Conversely there must be an entry in the code address range table that covers the instruction being read.

The 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 instruction fetches, the PC value for the fetch is checked against the execute enabled selected code protection ranges for the current protection set. When a PC is found to fall outside of all of the execute enabled selected ranges, then permission for the access is denied. When a PC is found to fall within an execute enabled range the access is permitted.

When an address is found to fall within one of the selected ranges the associated access permission is checked and the access allowed or denied as appropriate.

For load and store operations, data address values are checked against the selected data protection ranges for the current protection set. When an address is found to fall outside of all of the selected ranges then permission for the access is denied. When an address is found to fall within an enabled range the access is permitted.

For load operations, data address values are checked against the read enabled data protection ranges for the current protection set. When an address is found to fall outside of all of the selected ranges then permission for the access is denied. When an address is found to fall within an enabled range the access is permitted.

For store operations, data address values are checked against the write enabled data protection ranges for the current protection set. When an address is found to fall outside of all of the selected ranges then permission for the access is denied. When an address is found to fall within an enabled range the access is permitted.

When an address is found to fall within one of the selected ranges the associated access permissions are checked and access is allowed or denied as appropriate.

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 maps, 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 do not require the permission of the protection system to proceed.

10.3.4 Traps

There are three traps generated by the range based memory protection system, each corresponding to the three protection mode register bits:

- MPW (Memory Protection Write) trap = WE bit
- MPR (Memory Protection Read) trap = RE bit
- MPX (Memory Protection Execute) trap = XE bit

Refer to the Trap System chapter for a complete description of Traps.

10.3.5 Protection Register Naming Convention

Data Protection range registers are named as follows:

- DPRx_L - Defines the lower address boundary for data Range Pair x.
- DPRx_U - Defines the upper address boundary for data Range Pair x.

Code protection range registers are names as follows:

- CPRx_L - Defines the lower address boundary for code Range Pair x.
- CPRx_U - Defines the upper address boundary for code Range Pair x.

Note: x = implementation dependent.

10.3.6 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

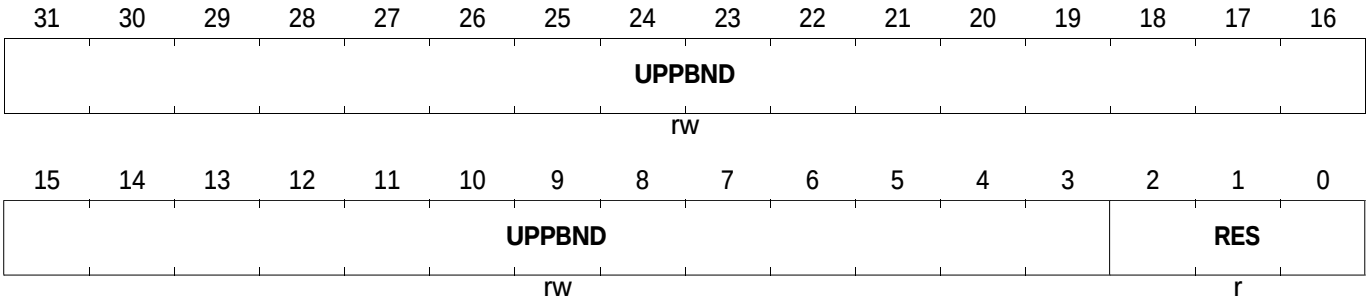
Within each of these registers range-x has permissions enabled if bit-x of the register is 1 else permission is disabled. As the number of code and data protection ranges is implementation dependent the number of bits in these registers is also implementation dependent.

10.4 Range Based Memory Protection Registers

Data Protection Range Register Upper Bound

DPRx_U (x=0-31)
Data Protection Range Register x Upper Bound

($C004_H$+x*8_H) **Reset Value: Implementation Specific**

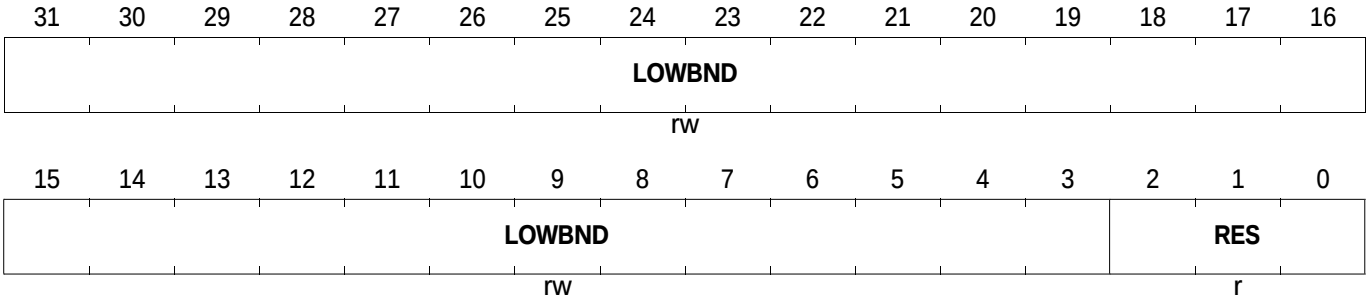


Field	Bits	Type	Description
UPPBND	[31:3]	rw	**DPRx_U Upper Boundary Address**
RES	[2:0]	r	**Reserved** The three least significant bits are not writeable and always return zero.

Data Protection Range Register Lower Bound

DPRx_L (x=0-31)
Data Protection Range Register x_0 Lower Bound

($C000_H+x*8_H$) **Reset Value: Implementation Specific**

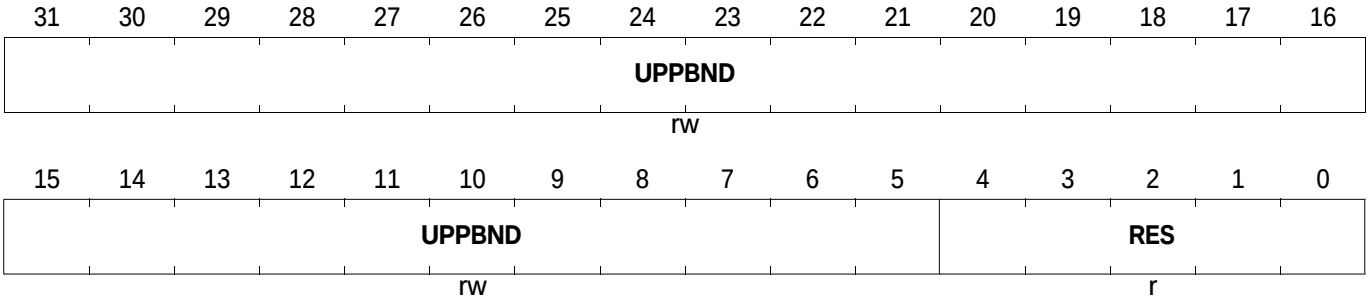


Field	Bits	Type	Description
LOWBND	[31:3]	rw	**DPRx_L Lower Boundary Address**
RES	[2:0]	r	**Reserved** The three least significant bits are not writeable and always return zero.

Code Protection Range Register Upper Bound

CPRx_U (x=0-31)
Code Protection Range Register x Upper Bound

($D004_H$+x*8_H) **Reset Value: Implementation Specific**

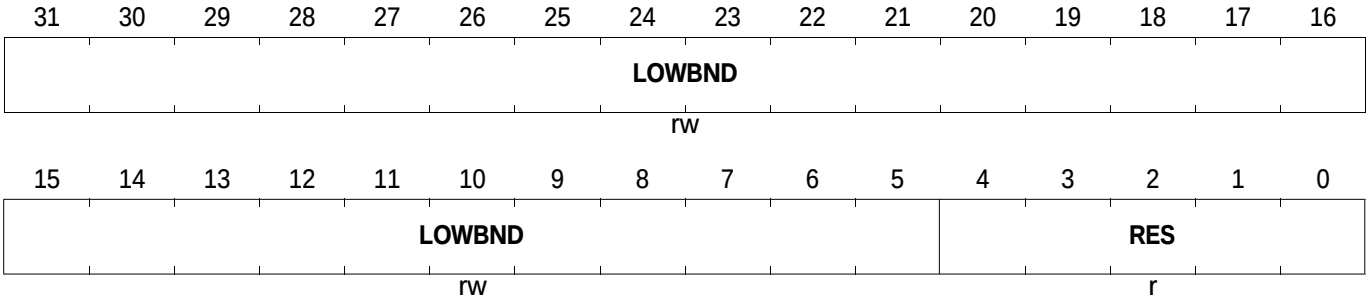


Field	Bits	Type	Description
UPPBND	[31:5]	rw	**CPRx_U Upper Boundary Address**
RES	[4:0]	r	**Reserved** The five least significant bits are not writeable and always return zero.

Code Protection Range Register Lower Bound

CPRx_L (x=0-31)
Code Protection Range Register x Lower Bound

($D000_H$+x*8_H) **Reset Value: Implementation Specific**

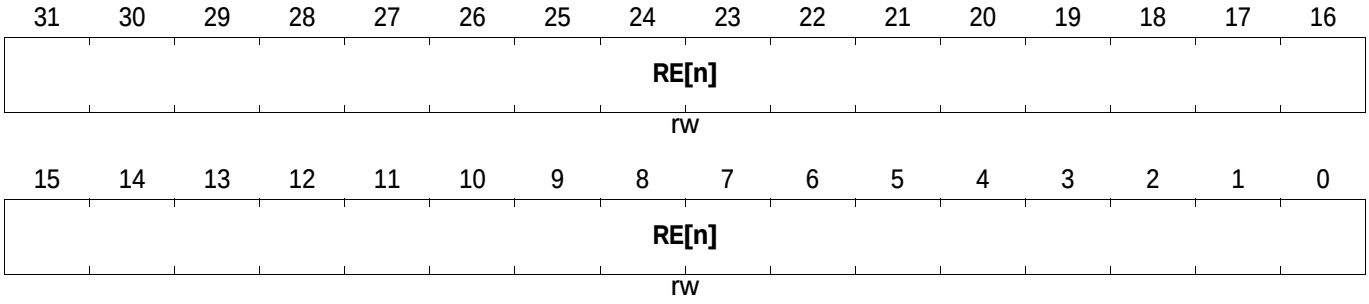


Field	Bits	Type	Description
LOWBND	[31:5]	rw	**CPRx_L Lower Boundary Address**
RES	[4:0]	r	**Reserved** The five least significant bits are not writeable and always returns zero.

Data Protection Read Enable Set Configuration Register

DPRE_x (x=0-7)
Data Protection Read Enable Set Configuration Register x

($E010_H$+x*4_H) **Reset Value: Implementation Specific**

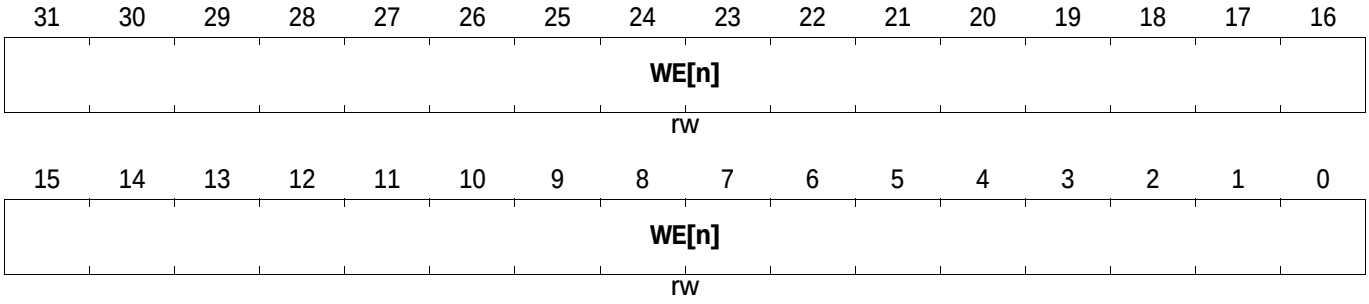


Field	Bits	Type	Description
RE[n]	[31:0]	rw	**Data protection Range Read Enable** 0 : Data read accesses to data protection range[n] not permitted for set x 1 : Data read accesses to data protection range[n] permitted for set x Note :- The number of protection ranges is implementation dependent. Enable bits for unimplemented ranges are read only and return 0 when read.

Data Protection Write Enable Set Configuration Register

DPWE_x (x=0-7)
Data Protection Write Enable Set Configuration Register x

($E020_H$+x*4_H) **Reset Value: Implementation Specific**



Field	Bits	Type	Description
WE[n]	[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.

Code Protection Execute Enable Set Configuration Register

CPXE_x (x=0-7)
Code Protection Execute Enable Set Configuration Register x

($E000_H$+x*4_H) **Reset Value: Implementation Specific**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
XE[n]															
rw															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
XE[n]															
rw															

Field	Bits	Type	Description
XE[n]	[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 consists of two primary mechanisms, the temporal protection timers and the exception 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 SYSCON 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 TEXP bit in the TPS_CON register is set. The TEXP bit is cleared by any write to the associated TPS_TIMERx register.

On setting any TEXP bit in the TPS_CON register, the TTRAP bit in the same register is set. A TAE trap is raised whenever the TTRAP bit transitions from zero to one.

The TTRAP bit is cleared by any write to the TPS_CON register. However attempting to clear the register while any TEXP bit is set will cause the TTRAP bit to be re-enabled and a new TAE trap is generated. This ensures that no time-out event is missed during the handling of another TAE trap.

11.2 Exception Timers

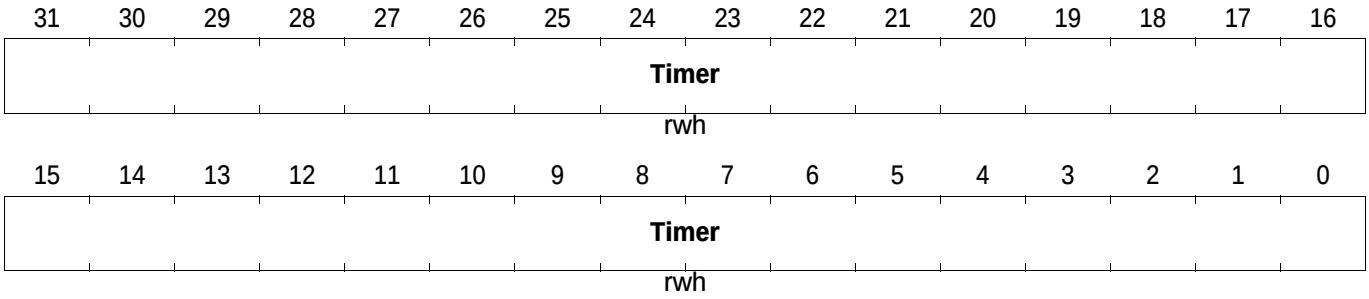
The exception timer system provides a method of detecting the overrun of exception handlers in the system. The TriCore™ architecture defines a set of register to be used with this system (TPS_EXTIM*). The deails of the system are implementation specific.

11.3 Temporal Protection System Registers

TPS Timer Register

Definition of the Temporal Protection System Timer register.

TPS_TIMERx (x=0-2)
TPS Timer Register x (E404+x*4H) **Reset Value: Implementation Specific**

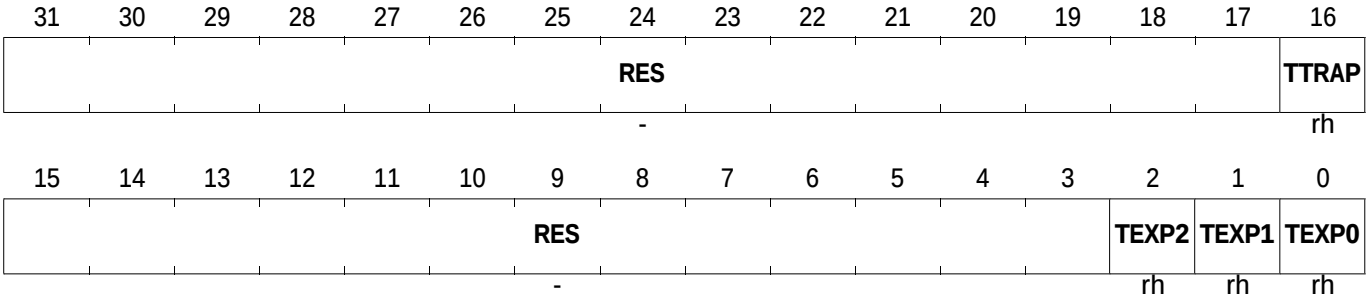


Field	Bits	Type	Description
Timer	[31:0]	rwh	**Temporal Protection Timer** Writing zero de-activates the Timer. Writing a non-zero value starts the Timer. Any write clears the corresponding TPS_CON.TEXP flag. Read returns the current Timer value.

TPS Control Register

Definition of the Temporal Protection System Control register.

TPS_CON
TPS Control Register (E400H) **Reset Value: Implementation Specific**

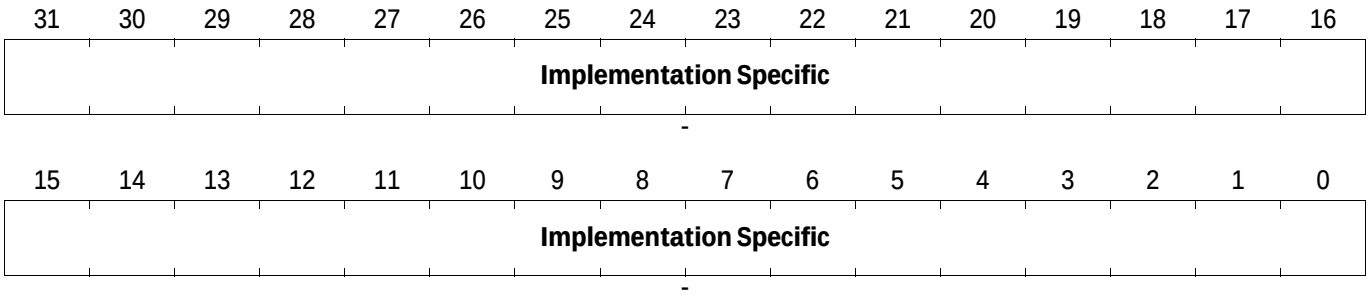


Field	Bits	Type	Description
RES	[31:17]	-	**Reserved**
TTRAP	16	rh	**Temporal Protection Trap** If set, indicates that a TAE trap has been requested. Any subsequent TAE traps are disabled. A write clears the flag and re-enables TAE traps.
RES	[15:2]	-	**Reserved**
TEXP2	2	rh	**Timer2 Expired flag** Set 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.

Exception Entry Timer Load value register

Implementation Specific Register.

TPS_EXTIM_ENTRY_LVAL
Exception Entry Timer Load value **($E440_H$)** **Reset Value: $0000\ 0000_H$**

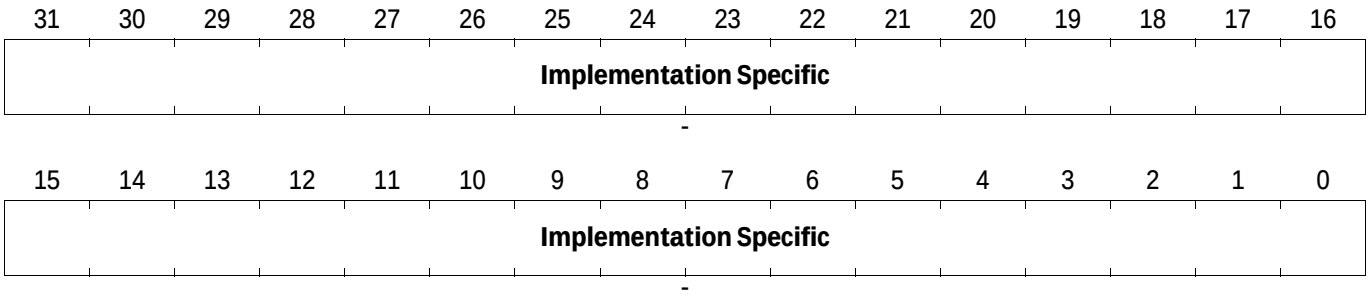


Field	Bits	Type	Description
Implementation Specific	[31:0]	-	**Implementation Specific**

Exception Exit Timer Load value register

Implementation Specific Register.

TPS_EXTIM_EXIT_LVAL
Exception Exit Timer Load value **($E448_H$)** **Reset Value: $0000\ 0000_H$**

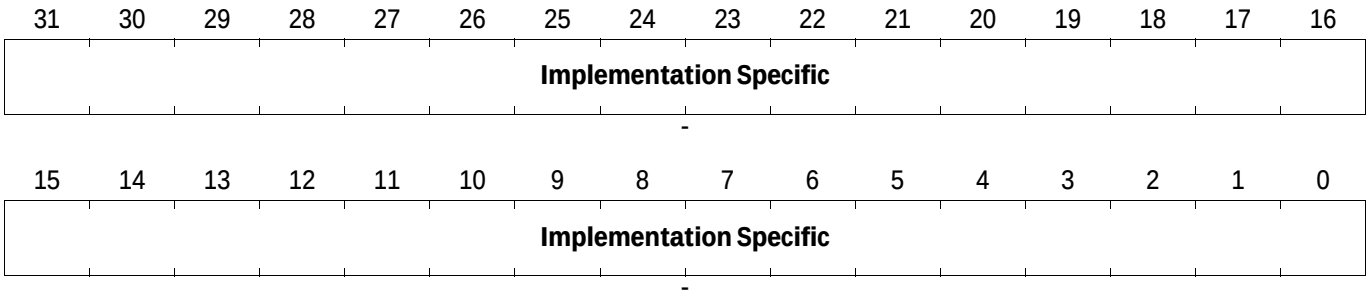


Field	Bits	Type	Description
Implementation Specific	[31:0]	-	**Implementation Specific**

Exception Entry Timer Current value register

Implementation Specific Register.

TPS_EXTIM_ENTRY_CVAL
Exception Entry Timer Current value **($E444_H$)** **Reset Value: $0000\ 0000_H$**

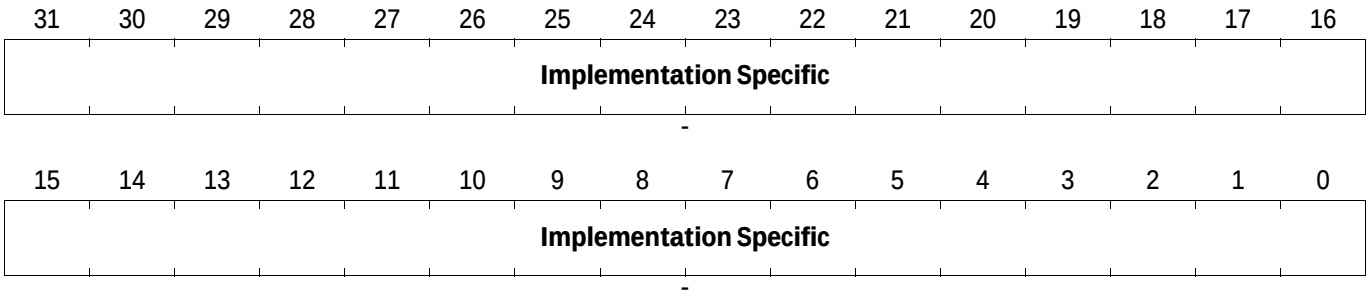


Field	Bits	Type	Description
Implementation Specific	[31:0]	-	**Implementation Specific**

Exception Exit Timer Load Current register

Implementation Specific Register.

TPS_EXTIM_EXIT_CVAL
Exception Exit Timer Current value **($E44C_H$)** **Reset Value: $0000\ 0000_H$**

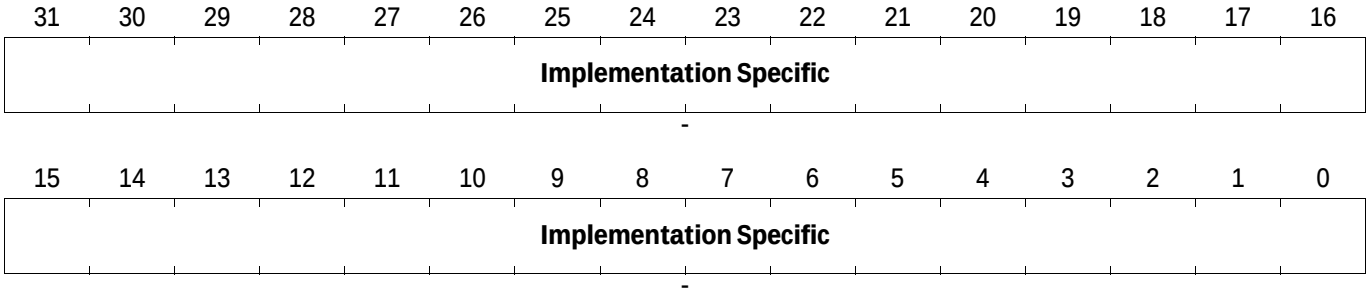


Field	Bits	Type	Description
Implementation Specific	[31:0]	-	**Implementation Specific**

Exception Timer Class Enable register

Implementation Specific Register.

TPS_EXTIM_CLASS_EN
Exception Timer Class Enable register (E450$_H$) Reset Value: 0000 0000$_H$

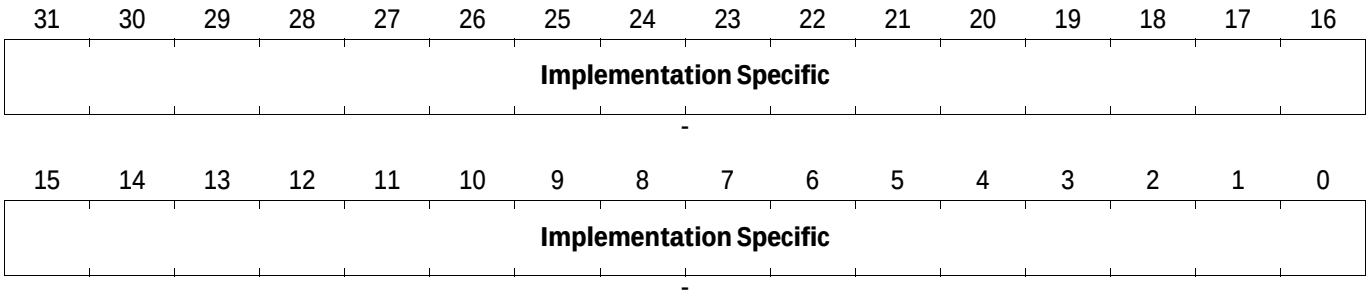


Field	Bits	Type	Description
Implementation Specific	[31:0]	-	**Implementation Specific**

Exception Timer Status register

Implementation Specific Register.

TPS_EXTIM_STAT
Exception Timer Status register **($E454_H$)** **Reset Value: $0000\ 0000_H$**

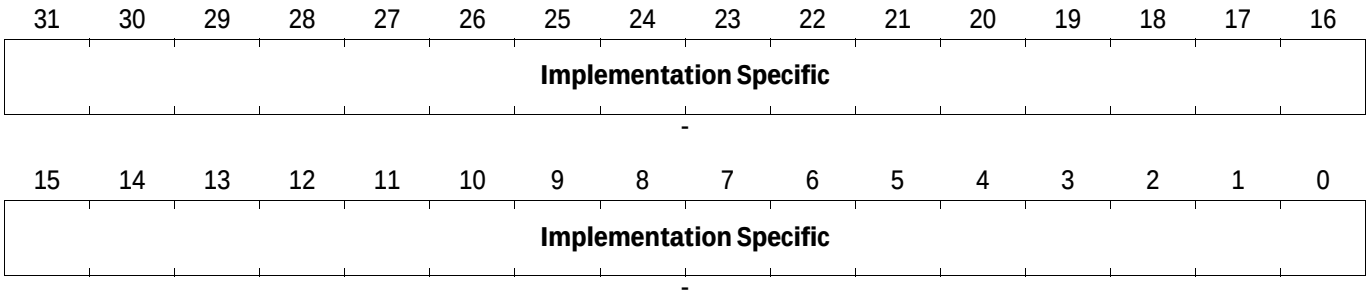


Field	Bits	Type	Description
Implementation Specific	[31:0]	-	**Implementation Specific**

Exception Timer FCX register

Implementation Specific Register.

TPS_EXTIM_FCX
Exception Timer FCX register **($E458_H$)** **Reset Value: $0000\ 0000_H$**



Field	Bits	Type	Description
Implementation Specific	[31:0]	-	**Implementation Specific**

12 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 via 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~~.

12.1 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) SUSP field (DBGSR.SUSP). This signal can be controlled by writes 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.

  - Events raised by the Trigger Event Unit (see **“Trigger Event Unit” on Page 4**).
- 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.

12.2 Debug Events

When the Core Debug Controller is enabled, a Debug Event can be generated by:

- Core Break-In signal.
  - See **"External Debug Event" on Page 3**.
- Execution of a DEBUG instruction.
  - See **"Debug Instruction" on Page 3**.
- Execution of the MTCR or MFCR instruction.
  - See **"MTCR and MFCR Instructions" on Page 3**.
- A hardware Event generation unit.
  - See **"Trigger Event Unit" on Page 4**.

12.2.1 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 EXEVT (External Event) register (see **"EXEVT" on Page 15**).

12.2.2 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 SWEVT (Software Debug Event) register (See **"SWEVT" on Page 19**).

12.2.3 MTCR and MFCR Instructions

A Debug Event is raised when a MTCR (Move To Core Register) or MFCR (Move From Core Register) instruction is used to read or modify a user Core Special Function Register (CSFR). This gives the debug software the ability to monitor, detect and modify changes to CSFRs. A Debug Event is not raised when a MTCR or MFCR 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 (**"DBGSR" on Page 13**).
- Core Register Access Event Register (**"CREVT" on Page 17**).
- Software Debug Event Register (**"SWEVT" on Page 19**).
- External Event Register (**"EXEVT" on Page 15**).
- Trigger Event Register (TRnEVT) (**"TRxEVT" on Page 21**).
- Debug Monitor Start Register (**"DMS" on Page 25**).
- Debug Context Pointer Register (**"DCX" on Page 26**).
- Debug Trap Control Register (**"DBGTCR" on Page 27**).
- Accumulated Trigger Information Register (**"TRIG_ACC" on Page 24**).

Additional Counter Registers

- Counter Control Register - **“Counter Control Register” on Page 32**.
- CPU Clock Count Register - **“CPU Clock Cycle Count Register” on Page 33**.
- Instruction Count Register - **“Instruction Count Register” on Page 34**.
- Multi-Count Register 1 - **“Multi-Count Register 1” on Page 35**.
- Multi-Count Register 2 - **“Multi-Count Register 2” on Page 36**.
- Multi-Count Register 3 - **“Multi-Count Register 3” on Page 37**.

The Debug Action taken when the Debug Event is raised is specified in the CREVT register (See **“CREVT” on Page 17**). Configuring the Debug Controller or accessing Performance counters will not cause a debug event.

12.2.4 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.

Note: Compared addresses are virtual addresses.

These Debug Triggers provide the inputs to a programmable block of logic which produces Debug Events as its output (see **Debug Triggers (pg 5)**).

The Debug Action taken when the Debug Event is raised, is specified in the Trigger Event register (TRnEVT). See **“Trigger Event Registers” on Page 21** for the register definition.

12.3 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.

12.3.1 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 event 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, etc.) is always reserved.

12.3.2 Task Specific Debug Triggers

In some instances it may be desirable to assert a debug trigger only when the target address is generated by a particular task. This is achieved by use of the Application Space Identifier (ASI) comparison feature.

If the ASI_EN bit in the Trigger Event register (TRnEVT) is set, then the trigger will only be asserted if both the address matches and the TRnEVT.ASI field matches the current task ASI (Programmed in the TASK_ASI register).

12.3.3 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.

12.4 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)" on Page 6**.
- **"Indicate on Core Break-Out Signal" on Page 6**.
- **"Indicate on Core Suspend-Out Signal" on Page 6**.
- **"Halt" on Page 6**.
- **"Breakpoint Trap" on Page 7**.
- **"Breakpoint Interrupt" on Page 8**.
- **"Suspend Out" on Page 9**.
- **"Performance Counter Start/Stop" on Page 9**.
- **"None" on Page 9**.
- **"Disabled" on Page 10**.
- **"Suspend In Halt" on Page 10**.

12.4.1 Update Debug Status Register (DBGSR)

When a Debug Event occurs the EVTSRC (Event Source), PEVT (Posted Event), PREVSUSP (Previous State of Suspend Signal) and SUSP (Current State of Suspend Signal) fields of the Debug Status Register (DBGSR) are always updated.

The PREVSUSP field is updated from the contents of the SUSP field.

SUSP is updated from the EVTA field of the register that prompted the Debug Event (EXEVT, CREVT, SWEVT or TRnEVT).

12.4.2 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.

12.4.3 Indicate on Core Suspend-Out Signal

On a Core Suspend-Out action, the value of the SUSP field in the Debug Status Register (DBGSR) is copied to the PREVSUSP field (DBGSR.PREVSUSP).

The DBGSR.SUSP field is updated with the contents of the SUSP field from the register that prompted the Debug Event (EXEVT, CREVT, SWEVT or TRnEVT).

Modification of the DBGSR.SUSP bit will be reflected in the Core Suspend-Out Signal. When writing to the DBGSR.SUSP bit, PREVSUSP is not updated.

When a debug event causes a breakpoint interrupt to be posted, DBGSR.SUSP, DBGSR.PREVSUSP and the Core Suspend-Out signal remain unchanged.

12.4.4 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 cancelled 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 DBGSTR.HALT field. This is treated as external event and will result in the DBGSTR fields being updated accordingly.

The DBGSTR.HALT field is cleared by reset and the CPU will resume normal operation.

12.4.5 Breakpoint Trap

The Breakpoint Trap enters a Debug Monitor without using any user resource. It relies upon the following emulator resources:

- A Debug Monitor 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 A10 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 which does not take any User 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 (i.e. DBGSR.DE == 0).

Note: The generation of breakpont 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],24'h000000}] will have MPX traps disabled for instruction fetches in debug mode.
- The 16 MByte region containing the DCX pointer (Base Address == {DCX[31:24],24'h000000}] 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 behaviour may be unconditionally disabled by setting the Emulator Space Disable bit in the SYSCON register (SYSCON.ESDIS).

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, A10, A11) 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.

12.4.6 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 dependant.

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 microcontrollers, where a Breakpoint stops the processor, by simply assigning the highest interrupt priority level to the Debug Monitor or by ensuring interrupts are disabled in the Debug Monitor. It also provides the flexibility for critical interrupts to be programmed with a higher priority than the Debug Monitor. The advantages of this are that:

- The Debug Monitor 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 is running.
- Any Debug Events posted in a critical routine are postponed until the CPU priority drops below that of the Debug Monitor.

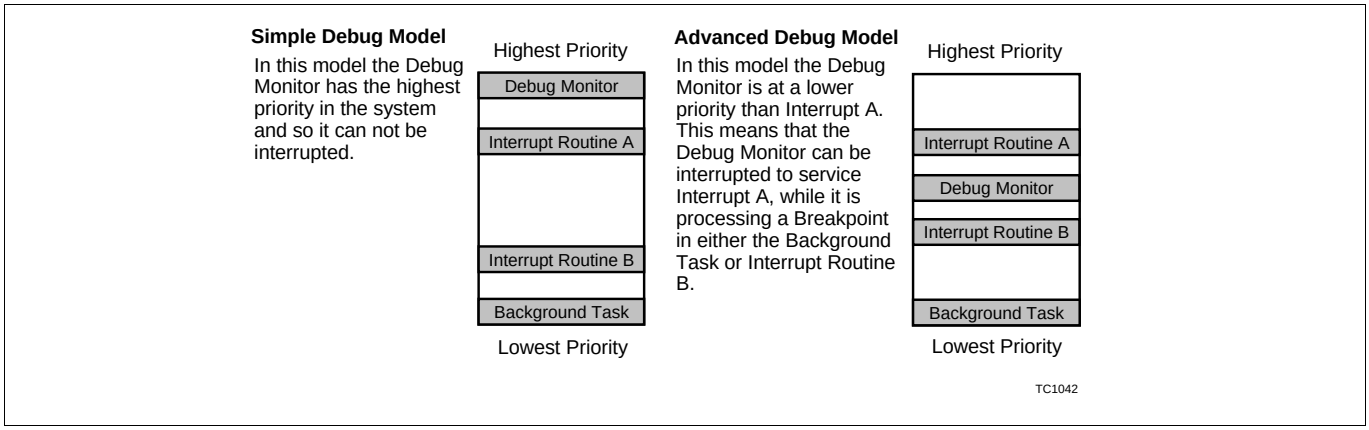


Figure 27 Debug Monitor - Simple and Advanced Models

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 **Figure 27** for example, if a Breakpoint Interrupt is set in Interrupt Routine 'A' it is a problem, because the Debug Monitor is programmed to be at a lower priority than the current Task.

This scenario is indicated by posting a software interrupt at the interrupt level associated with the Breakpoint. Therefore, when the CPU interrupt priority level falls below that of the Debug Monitor, 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 software 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.SUSP and DBGSR.PREVSUSP are not updated when a breakpoint interrupt is posted.

1. *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.

12.4.7 Suspend Out

The suspend out signal will either be asserted or negated when a debug event occurs. The previous state of the suspend out signal is recorded in DBGSR.PREVSUSP.

12.4.8 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.

12.4.9 None

No action is implemented through the EVTA field of the event's register, however the suspend out signal, performance count and DBGSR register updates still occur as normal for an event.

12.4.10 Disabled

The event is disabled and no actions occur: the suspend out signal, performance counter control and DBGSR register ignore the event.

12.4.11 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.

12.5 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.
- MTCR/MFCR Instruction.
- Debug Instruction.
- Programmable triggers on Address
  - When multiple triggers are active, 0 has the highest priority and 7 the lowest.

12.6 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).

12.7 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.

12.8 Debug Control Registers - Summary

Core Debug Controller Registers.

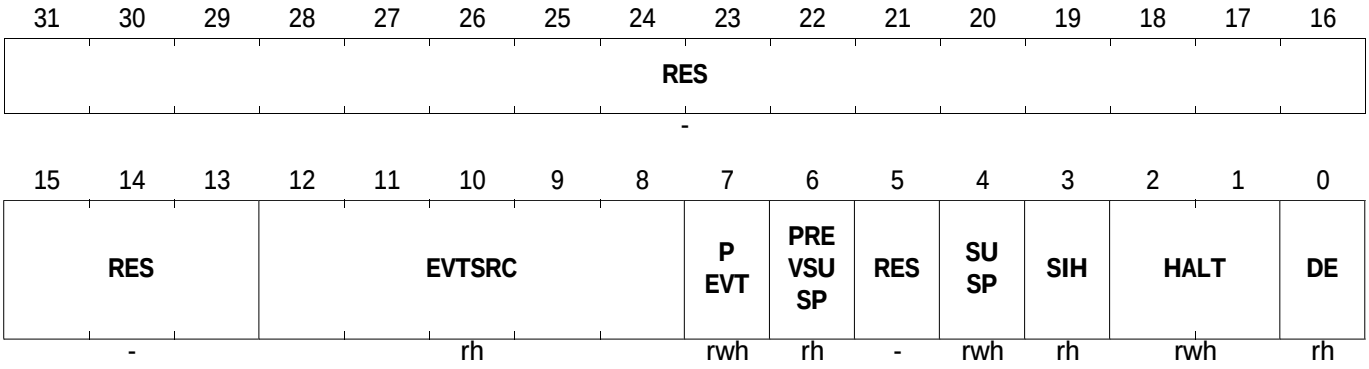
Table 19 Debug Control Registers Summary

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	$FD10_H$
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	$FD48_H$
TASK_ASI	Application Space Idenitifier Register	8004_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	$F03C_H$

12.9 Debug Control Registers

Debug Status Register

DBGSR
Debug Status Register **($FD00_H$)** **Reset Value: $0000\ 0000_H$**



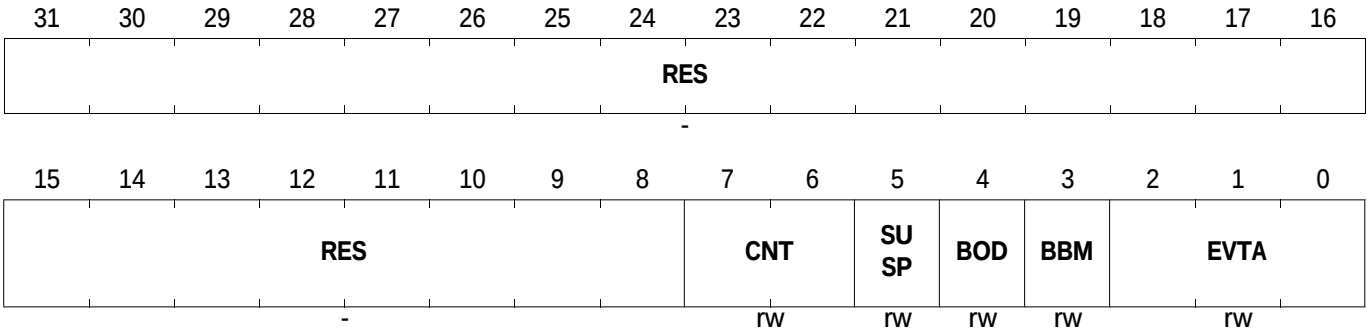
Field	Bits	Type	Description
RES	[31:13]	-	**Reserved**
EVTSRC	[12:8]	rh	**Event Source** 0 : EXEVT. 1 : CREVT. 2 : SWEVT. 16 + n TRnEVT (n = 0, 7). Other = Reserved.
PEVT	7	rwh	**Posted Event** 0 : No posted event. 1 : Posted event.
PREVSUSP	6	rh	**Previous State of Core Suspend-Out Signal** 0 : Previous core suspend-out inactive. 1 : Previous core suspend-out active. Updated when a Debug Event causes a hardware update of DBGSR.SUSP. This field is not updated for writes to DBGSR.SUSP.
RES	5	-	**Reserved**
SUSP	4	rwh	**Current State of the Core Suspend-Out Signal** 0 : Core suspend-out inactive. 1 : Core suspend-out active.
SIH	3	rh	**Suspend-in Halt** State of the Suspend-In signal. 1 : The Suspend-In signal is asserted. The CPU is in Halt Mode. 0 : 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).

Field	Bits	Type	Description
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, set HALT[0]. If DBGSR.DE == 0, HALT[0] is left unchanged.
DE	0	rh	**Debug Enable** Determines whether the core debug is enabled or not. 0 : The Core Debug is disabled. 1 : The Core Debug is enabled.

Core Debug Controller

External Event Register

EXEVT
External Event Register **($FD08_H$)** **Reset Value: $0000\ 0000_H$**



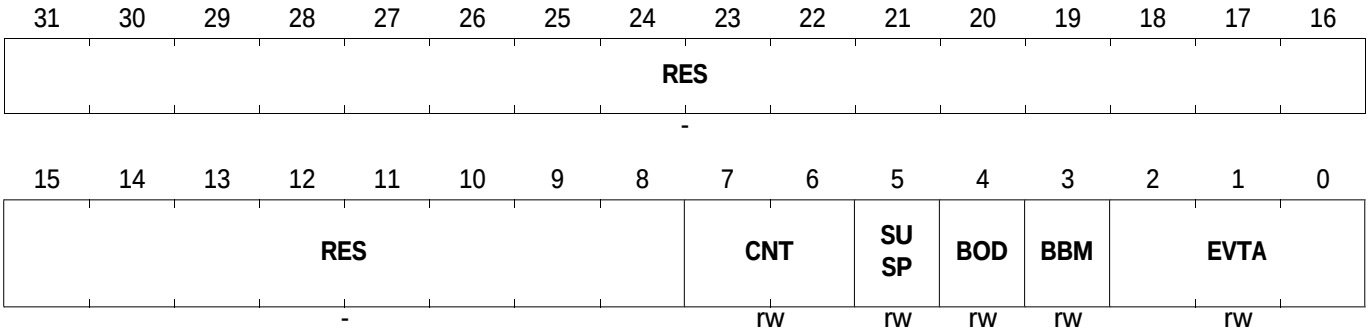
Field	Bits	Type	Description
RES	[31:8]	-	**Reserved**
CNT	[7:6]	rw	**Counter** When this event occurs adjust the control of the performance counters in task mode as follows: 00: No change. 01: Start the performance counters. 10: Stop the performance counters. 11: Toggle the performance counter control (i.e. start it if it is currently stopped, stop it if it is currently running).
SUSP	5	rw	**CCore Debug Suspend-Out Signal State** Value to be assigned to the Core Debug suspend-out signal when the Debug Event is raised.
BOD	4	rw	**Breakout Disable** 0 : BRKOUT signal asserted according to the Debug Action specified in the EVTA field. 1 : BRKOUT signal not asserted. This takes priority over any assertion generated by the EVTA field.
BBM	3	rw	**Break Before Make (BBM) or Break After Make (BAM) Selection** 0 : Break after make (BAM). 1 : Break before make (BBM).

Field	Bits	Type	Description
EVTA	[2:0]	rw	**Event Associated** Debug Action associated with the Debug Event: **When field BOD = 0** 000_B : Disabled. 001_B : Pulse BRKOUT Signal. 010_B : Halt and pulse BRKOUT Signal. 011_B : Breakpoint trap and pulse BRKOUT Signal. 100_B : Breakpoint interrupt 0 and pulse BRKOUT Signal. 101_B : If implemented, breakpoint interrupt 1 and pulse BRKOUT Signal[1]. 110_B : If implemented, breakpoint interrupt 2 and pulse BRKOUT Signal[1]. 111_B : If implemented, breakpoint interrupt 3 and pulse BRKOUT Signal[1]. **When field BOD = 1** 000_B : Disabled. 001_B : None. 010_B : Halt. 011_B : Breakpoint trap. 100_B : Breakpoint interrupt 0. 101_B : If implemented, breakpoint interrupt 1[1]. 110_B : If implemented, breakpoint interrupt 2[1]. 111_B : If implemented, breakpoint interrupt 3[1].

1) If not implemented, None

Core Register Access Event Register

CREVT
Core Register Access Event **($FD0C_H$)** **Reset Value: $0000\ 0000_H$**



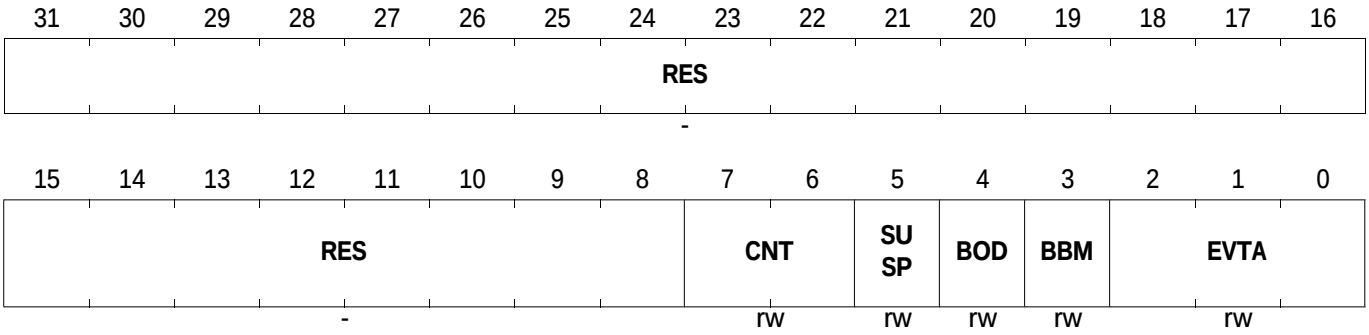
Field	Bits	Type	Description
RES	[31:8]	-	**Reserved**
CNT	[7:6]	rw	**Counter** When this event occurs adjust the control of the performance counters in task mode as follows: 00: No change. 01: Start the performance counters. 10: Stop the performance counters. 11: Toggle the performance counter control (i.e. start it if it is currently stopped, stop it if it is currently running).
SUSP	5	rw	**Core Debug Suspend-Out Signal State** Value to be assigned to the Core debug suspend-out signal when the Debug Event is raised.
BOD	4	rw	**Breakout Disable** 0 : BRKOUT signal asserted according to the action specified in the EVTA field. 1 : BRKOUT signal not asserted. This takes priority over any assertion generated by the EVTA field.
BBM	3	rw	**Break Before Make (BBM) or Break After Make (BAM) Selection** 0 : Break after make (BAM). 1 : Break before make (BBM).

Field	Bits	Type	Description
EVTA	[2:0]	rw	**Event Associated** Debug Action associated with the Debug Event: **When field BOD = 0** 000_B : Disabled. 001_B : Pulse BRKOUT Signal. 010_B : Halt and pulse BRKOUT Signal. 011_B : Breakpoint trap and pulse BRKOUT Signal. 100_B : Breakpoint interrupt 0 and pulse BRKOUT Signal. 101_B : If implemented, breakpoint interrupt 1 and pulse BRKOUT Signal[1]. 110_B : If implemented, breakpoint interrupt 2 and pulse BRKOUT Signal[1]. 111_B : If implemented, breakpoint interrupt 3 and pulse BRKOUT Signal[1]. **When field BOD = 1** 000_B : Disabled. 001_B : None. 010_B : Halt. 011_B : Breakpoint trap. 100_B : Breakpoint interrupt 0. 101_B : If implemented, breakpoint interrupt 1[1]. 110_B : If implemented, breakpoint interrupt 2[1]. 111_B : If implemented, breakpoint interrupt 3[1].

1) If not implemented, None

Software Debug Event Register

SWEVT
Software Debug Event **($FD10_H$)** **Reset Value: $0000\ 0000_H$**



Field	Bits	Type	Description
RES	[31:8]	-	**Reserved**
CNT	[7:6]	rw	**Counter** When this event occurs adjust the control of the performance counters in task mode as follows: 00: No change. 01: Start the performance counters. 10: Stop the performance counters. 11: Toggle the performance counter control (i.e. start it if it is currently stopped, stop it if it is currently running).
SUSP	5	rw	**Core Debug Suspend-Out Signal State** Value to be assigned to the Core Debug suspend-out signal when the event is raised.
BOD	4	rw	**Breakout Disable** 0 : BRKOUT signal asserted according to the action specified in the EVTA field. 1 : BRKOUT signal not asserted. This takes priority over any assertion generated by the EVTA field.
BBM	3	rw	**Break Before Make (BBM) or Break After Make (BAM) Selection** 0 : Break after make (BAM). 1 : Break before make (BBM).

Field	Bits	Type	Description
EVTA	[2:0]	rw	**Event Associated** Debug Action associated with the Debug Event: **When field BOD = 0** 000_B : Disabled. 001_B : Pulse BRKOUT Signal. 010_B : Halt and pulse BRKOUT Signal. 011_B : Breakpoint trap and pulse BRKOUT Signal. 100_B : Breakpoint interrupt 0 and pulse BRKOUT Signal. 101_B : If implemented, breakpoint interrupt 1 and pulse BRKOUT Signal[1]. 110_B : If implemented, breakpoint interrupt 2 and pulse BRKOUT Signal[1]. 111_B : If implemented, breakpoint interrupt 3 and pulse BRKOUT Signal[1]. **When field BOD = 1** 000_B : Disabled. 001_B : None. 010_B : Halt. 011_B : Breakpoint trap. 100_B : Breakpoint interrupt 0. 101_B : If implemented, breakpoint interrupt 1[1]. 110_B : If implemented, breakpoint interrupt 2[1]. 111_B : If implemented, breakpoint interrupt 3[1].

1) If not implemented, None

Trigger Event Registers

TRxEVT stores the configuration of each trigger.

TRxEVT will be duplicated as many times as there are comparators.

Note: The RNG bit of 'odd' numbered Trigger Event registers (TR1EVT, TR3EVT, etc.) is always reserved.

TRxEVT
Trigger Event x **(F0XX$_H$)** **Reset Value: 0000 0000$_H$**

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ASI_EN	RES	RNG	TYP	RES				CNT		SUSP	BOD	BBM	EVTA		
rw	-	rw	rw	-				rw		rw	rw	rw	rw		

Field	Bits	Type	Description
RES	[31:29]	-	**Reserved**
ALD	28	rw	**Address Load** Used in conjunction with TYP=0
AST	27	rw	**Address Store** Used in conjunction with TYP=0
RES	[26:21]	-	**Reserved**
ASI	[20:16]	rw	**Address Space Identifier** The ASI of the Debug Trigger process.
ASI_EN	15	rw	**Enable ASI Comparison** 0 : No ASI comparison performed. Debug Trigger is valid for all processes. 1 : Enable ASI comparison. Debug Events are only triggered when the current process ASI matches TRnEVT.ASI.
RES	14	-	**Reserved**
RNG	13	rw	**Compare Type** *Note: The RNG bit of 'odd' numbered Trigger Event registers (TR1EVT, TR3EVT, etc.) is always reserved. The following definition only applies to 'even' numbered Trigger Event registers (i.e. TR0EVT, TR2EVT, etc.).* 1_B Range 0_B Equality Once an even numbered comparator has been set to range, the EVTR settings of its associated upper neighbour will be ignored.
TYP	12	rw	**Input Selection** 0_B Address 1_B PC

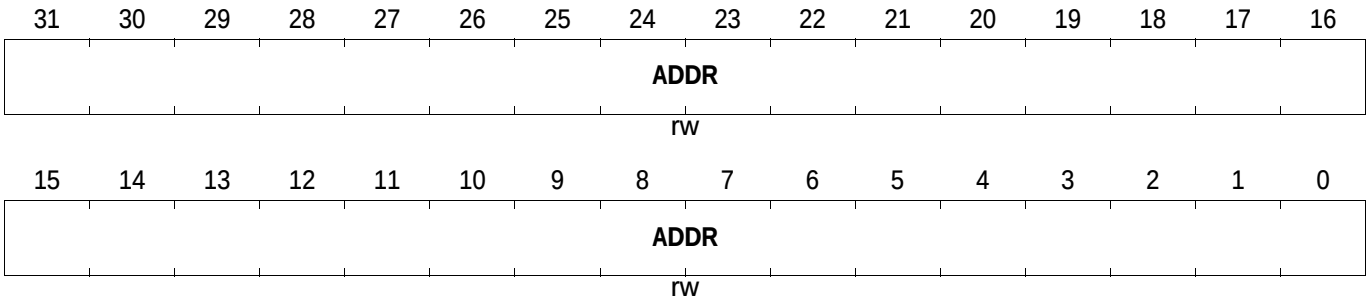
Field	Bits	Type	Description
RES	[11:8]	-	**Reserved**
CNT	[7:6]	rw	**Counter** When this event occurs adjust the control of the performance counters in task mode as follows: 00: No change. 01: Start the performance counters. 10: Stop the performance counters. 11: Toggle the performance counter control (i.e. start it if it is currently stopped, stop it if it is currently running).
SUSP	5	rw	**Core Debug Suspend-Out Signal State** Value to be assigned to the Core Debug suspend-out signal when the Debug Event is raised.
BOD	4	rw	**Breakout Disable** 0 : BRKOUT signal asserted according to the action specified in the EVTA field. 1 : BRKOUT signal not asserted. This takes priority over any assertion generated by the EVTA field.
BBM	3	rw	**Break Before Make (BBM) or Break After Make (BAM) Selection** Code triggers BBM or BAM selection. 0 : Code only triggers Break After Make (BAM). 1 : Code only triggers Break Before Make (BBM). Trigger BBM or BAM selection. 0 : Triggers is Break After Make (BAM). 1 : Triggers is Break Before Make (BBM).
EVTA	[2:0]	rw	**Event Associated** Specifies the Debug Action associated with the Debug Event: **When field BOD = 0** 000_B : Disabled. 001_B : Pulse BRKOUT Signal. 010_B : Halt and pulse BRKOUT Signal. 011_B : Breakpoint trap and pulse BRKOUT Signal. 100_B : Breakpoint interrupt 0 and pulse BRKOUT Signal. 101_B : If implemented, breakpoint interrupt 1 and pulse BRKOUT Signal[1]. 110_B : If implemented, breakpoint interrupt 2 and pulse BRKOUT Signal[1]. 111_B : If implemented, breakpoint interrupt 3 and pulse BRKOUT Signal[1]. **When field BOD = 1** 000_B : Disabled. 001_B : None. 010_B : Halt. 011_B : Breakpoint trap. 100_B : Breakpoint interrupt 0. 101_B : If implemented, breakpoint interrupt 1[1]. 110_B : If implemented, breakpoint interrupt 2[1]. 111_B : If implemented, breakpoint interrupt 3[1].

1) If not implemented, None

Trigger Address Register

TRxADR stores the comparison address value for each trigger.

TRxADR
Trigger Address x **($F0XX_H$)** **Reset Value: $0000\ 0000_H$**



Field	Bits	Type	Description
ADDR	[31:0]	rw	**Comparison Address** *Note: For PC comparison, bit[0] is always zero.*

Trigger Accumulator Register

TRIG_ACC stores the accumulated debug trigger state since the register was last cleared.

Note: This register is cleared by any read operation, write operations are ignored.

TRIG_ACC
Core Debug Trigger Accumulator (FD30$_H$) Reset Value: 0000 0000$_H$

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

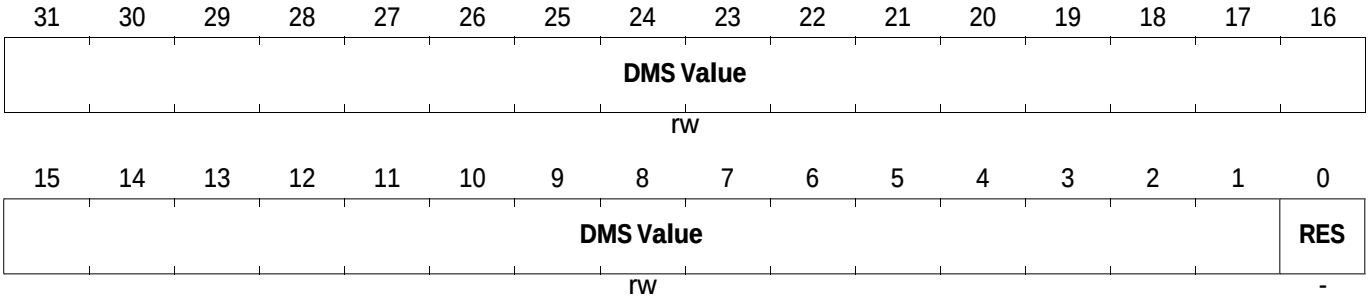
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES								T7	T6	T5	T4	T3	T2	T1	T0
-								rh	rh	rh	rh	rh	rh	rh	rh

Field	Bits	Type	Description
RES	[31:8]	-	**Reserved**
T7	7	rh	**Trigger-7 active since last cleared**
T6	6	rh	**Trigger-6 active since last cleared**
T5	5	rh	**Trigger-5 active since last cleared**
T4	4	rh	**Trigger-4 active since last cleared**
T3	3	rh	**Trigger-3 active since last cleared**
T2	2	rh	**Trigger-2 active since last cleared**
T1	[1	rh	**Trigger-1 active since last cleared**
T0	0	rh	**Trigger-0 active since last cleared**

Debug Monitor Start Address Register

The DMS reset value is {20'hA0000,3'B0001,CORE_ID,6'B000000}.

DMS
Debug Monitor Start Address **($FD40_H$)** **Reset Value: Implementation Specific**

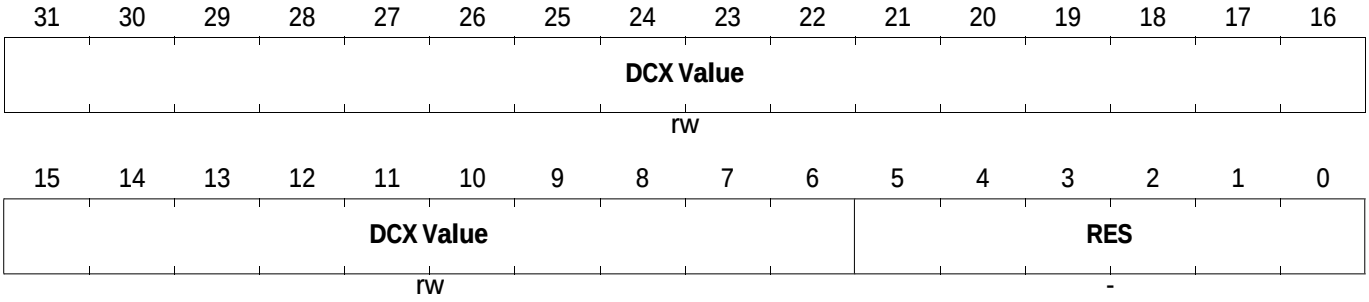


Field	Bits	Type	Description
DMS Value	[31:1]	rw	**Debug Monitor Start Address** The address at which monitor code execution begins when a breakpoint trap is taken.
RES	0	-	**Reserved**

Debug Context Save Area Pointer Register

The reset value of the DCX register is {20'hA0000,3'b010,core_id,6'b000000}.

DCX
Debug Context Save Area Pointer **($FD44_H$)** **Reset Value: Implementation Specific**



Field	Bits	Type	Description
DCX Value	[31:6]	rw	**Debug Context Save Area Pointer** Address where the debug context is stored following a breakpoint trap.
RES	[5:0]	-	**Reserved**

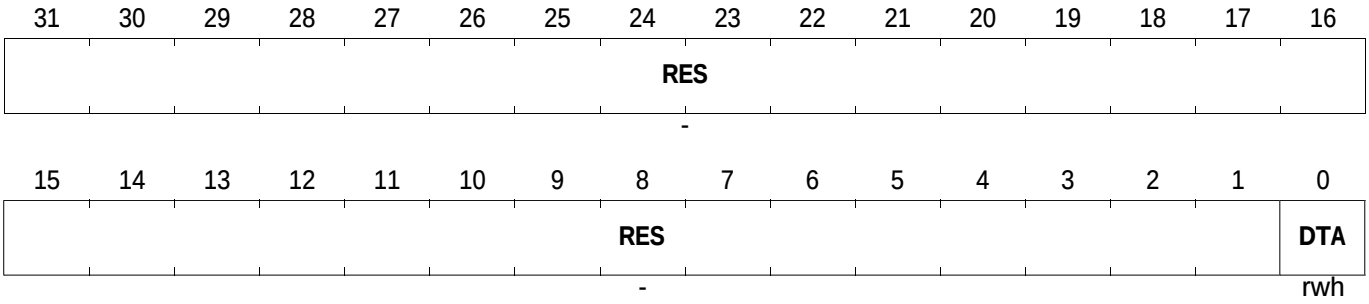
Debug Trap Control Register

The Debug Trap Control Register contains the DTA (Debug Trap Active) bit.

The DTA bit is defined as being cleared on an RFM instruction and set on a breakpoint trap. It may also be set and cleared by MTCR.

After an application reset the DTA bit is set to one. The register must therefore be cleared before a debug trap may be taken.

DBGTCR
Debug Trap Control Register **($FD48_H$)** **Reset Value: $0000\ 0001_H$**

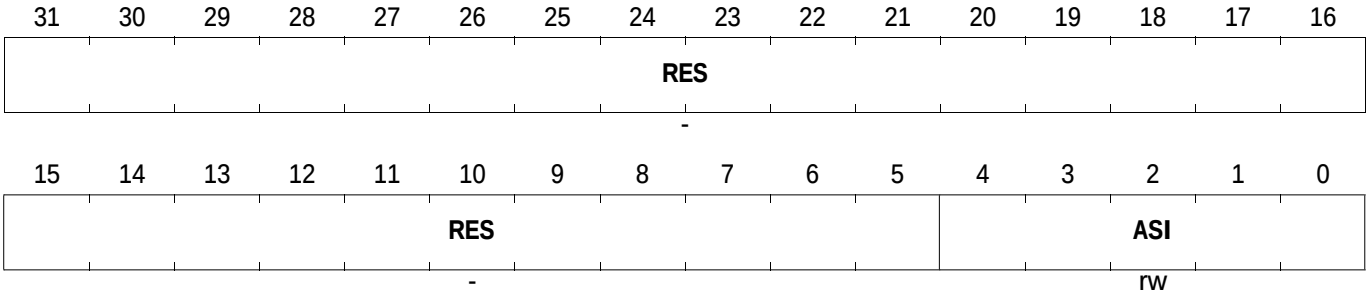


Field	Bits	Type	Description
RES	[31:1]	-	**Reserved**
DTA	0	rwh	**Debug Trap Active Bit** 1: A breakpoint Trap is active 0: No breakpoint trap is active. A breakpoint trap may only be taken in the condition DTA == 0. Taking a breakpoint trap sets the DTA bit to one. Further breakpoint traps are therefore disabled until such time as the breakpoint trap handler clears the DTA bit or until the breakpoint trap handler terminates with a RFM.

Address Space Identifier Register (TASK_ASI)

The Address Space Identifier (ASI) register description.

TASK_ASI
Address Space Identifier Register **(8004_H)** **Reset Value: Implementation Specific**



Field	Bits	Type	Description
RES	[31:5]	-	**Reserved**
ASI	[4:0]	rw	**Address Space Identifier** The ASI register contains the Address Space Identifier of the current process.

12.10 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 (i.e. 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.

The grouping of counter functions allows typical measurements to be clustered; i.e. 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 (i.e. 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.
- The mode is active (Normal mode is always active).
- 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 (i.e. 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 user 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 counter is sticky. It gets set when bits 30:0 wrap. It stays set until written by software.

12.11 Performance Counter Registers

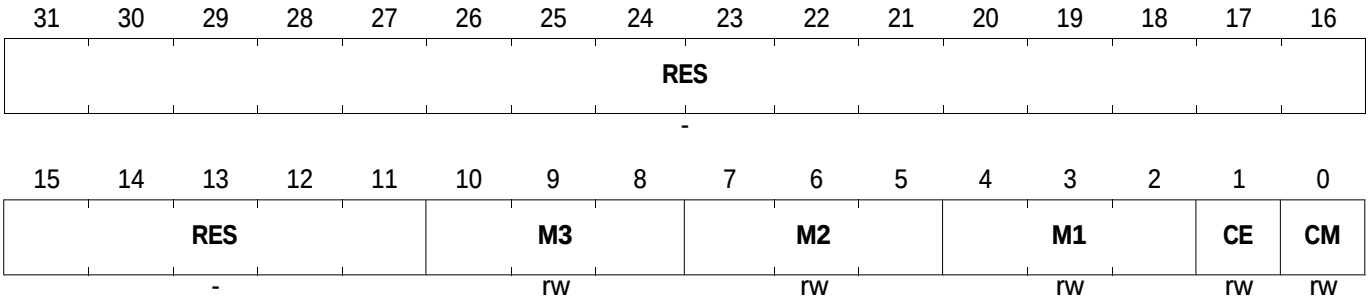
The performance counter registers are:

Register	Description	Offset Address	Reference
CCTRL	Counter Control Register.	$FC00_H$	Page 32
CCNT	CPU Clock Count Register.	$FC04_H$	Page 33
ICNT	Instruction Count Register.	$FC08_H$	Page 34
M1CNT	Multi Count Register 1.	$FC0C_H$	Page 35
M2CNT	Multi Count Register 2.	$FC10_H$	Page 36
M3CNT	Multi Count Register 3.	$FC14_H$	Page 37

Core Debug Controller

Counter Control Register

CCTRL
Counter Control **($FC00_H$)** **Reset Value: $0000\ 0000_H$**

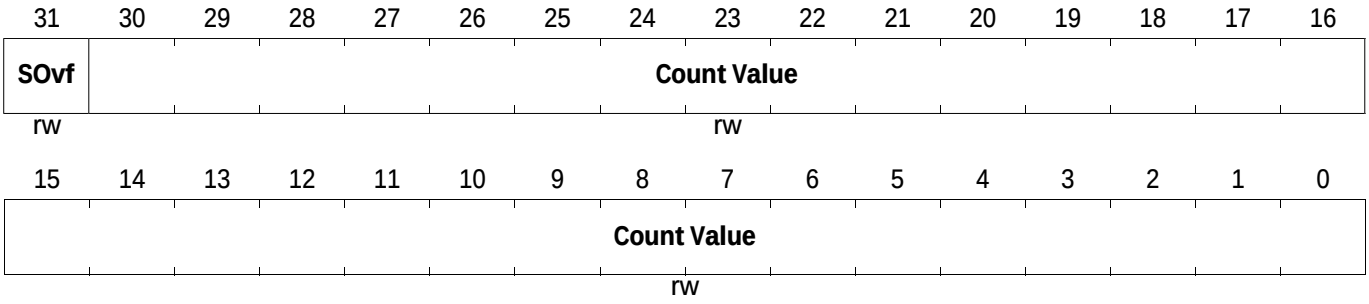


Field	Bits	Type	Description
RES	[31:11]	-	**Reserved**
M3	[10:8]	rw	**M3CNT configuration - Implementation Specific**
M2	[7:5]	rw	**M2CNT configuration - Implementation Specific**
M1	[4:2]	rw	**M1CNT configuration - Implementation Specific**
CE	1	rw	**Count Enable** 0 : Disable the counters: CCNT, ICNT, M1CNT, M2CNT, M3CNT. 1 : Enable the counters: CCNT, ICNT, M1CNT, M2CNT, M3CNT.
CM	0	rw	**Counter Mode** 0 : Normal Mode. 1 : Task Mode.

Core Debug Controller

CPU Clock Cycle Count Register

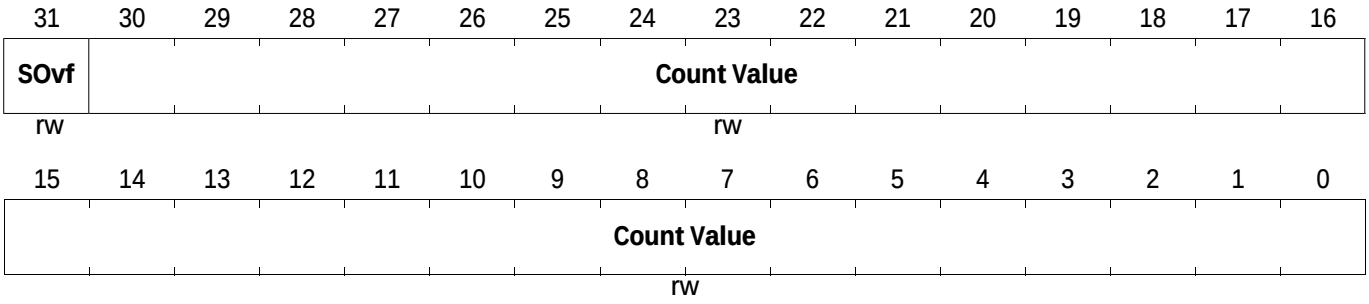
CCNT
CPU Clock Cycle Count **($FC04_H$)** **Reset Value: $0000\ 0000_H$**



Field	Bits	Type	Description
SOvf	31	rw	**Sticky Overflow bit** Set by hardware when count value [30:0] = 31'h7FFF_FFFF. It can only be cleared by software.
Count Value	[30:0]	rw	**Count Value** Current Count of the CPU Clock Cycles.

Instruction Count Register

ICNT
Instruction Count **($FC08_H$)** **Reset Value: $0000\ 0000_H$**

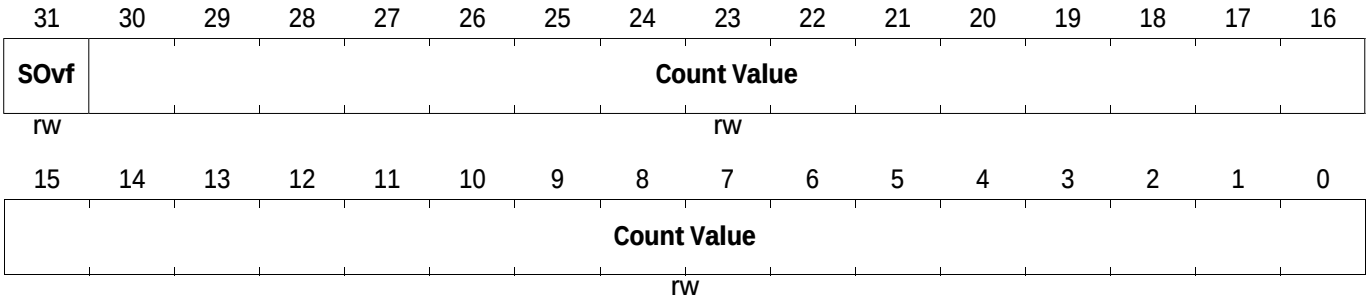


Field	Bits	Type	Description
SOvf	31	rw	**Sticky Overflow bit** Set by hardware when count value [30:0] = 31'h7FFF_FFFF. It can only be cleared by software.
Count Value	[30:0]	rw	**Count Value** Count of the Instructions Executed.

Core Debug Controller

Multi-Count Register 1

M1CNT
Multi-Count Register 1 **($FC0C_H$)** **Reset Value: $0000\ 0000_H$**

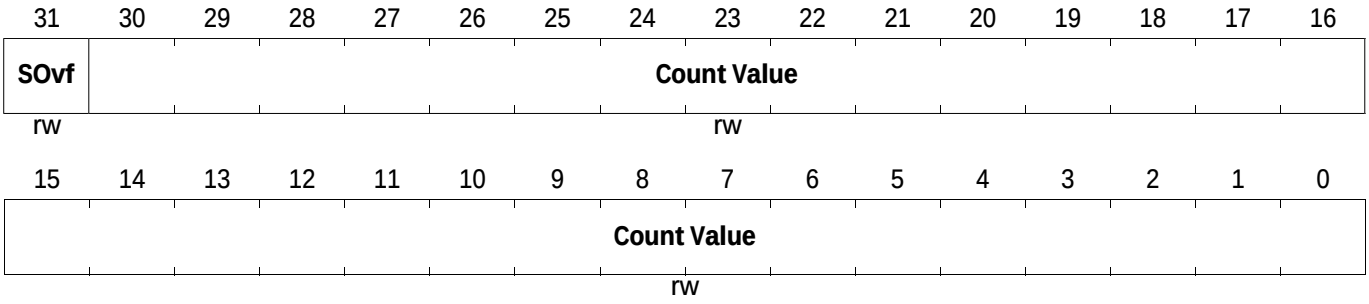


Field	Bits	Type	Description
SOvf	31	rw	**Sticky Overflow bit** Set by hardware when count value [30:0] = 31'h7FFF_FFFF. It can only be cleared by software.
Count Value	[30:0]	rw	**Count Value** Count of the Selected Event.

Core Debug Controller

Multi-Count Register 2

M2CNT
Multi-Count Register 2 **($FC10_H$)** **Reset Value: $0000\ 0000_H$**

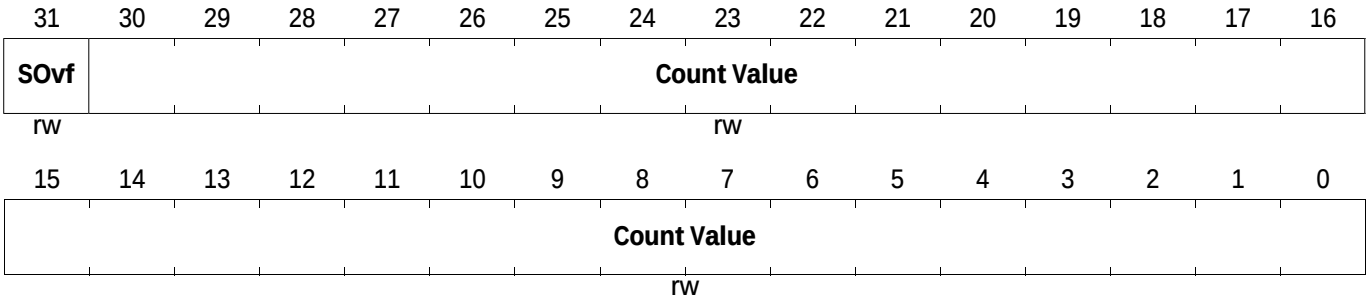


Field	Bits	Type	Description
SOvf	31	rw	**Sticky Overflow bit** Set by hardware when count value [30:0] = 31'h7FFF_FFFF. It can only be cleared by software.
Count Value	[30:0]	rw	**Count Value** Count of the Selected Event.

Core Debug Controller

Multi-Count Register 3

M3CNT
Multi-Count Register 3 **($FC14_H$)** **Reset Value: $0000\ 0000_H$**



Field	Bits	Type	Description
SOvf	31	rw	**Sticky Overflow bit** Set by hardware when count value [30:0] = 31'h7FFF_FFFF. It can only be cleared by software.
Count Value	[30:0]	rw	**Count Value** Count of the Selected Event.

13 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 20 General Purpose Registers (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.	$FF80_H$ [1]
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 instruction.

Table 21 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.	$FE04_H$

Table 21 Core Special Function Registers (CSFR) (cont'd)

Register Name	Description	Address Offset
PC	Program Counter Register.	$FE08_H$
SYSCON[2)]	System Configuration Register.	$FE14_H$
CPU_ID	CPU Identification Register (Read Only).	$FE18_H$
CORE_ID	Core Identification Register	$FE1C_H$
BIV[1)]	Base Address of Interrupt Vector Table Register.	$FE20_H$
BTV[1)]	Base Address of Trap Vector Table Register.	$FE24_H$
ISP[1)]	Interrupt Stack Pointer Register.	$FE28_H$
ICR	ICU Interrupt Control Register.	$FE2C_H$
FCX	Free Context List Head Pointer Register.	$FE38_H$
LCX	Free Context List Limit Pointer Register.	$FE3C_H$
COMPAT[1)2)]	Compatibility Mode Register.	9400_H
Memory Protection Registers		
DPR0_L	Data Segment Protection Range 0, Lower.	$C000_H$
DPR0_U	Data Segment Protection Range 0, Upper.	$C004_H$
DPR1_L	Data Segment Protection Range 1, Lower.	$C008_H$
DPR1_U	Data Segment Protection Range 1, Upper.	$C00C_H$
DPR2_L	Data Segment Protection Range 2, Lower.	$C010_H$
DPR2_U	Data Segment Protection Range 2, Upper.	$C014_H$
DPR3_L	Data Segment Protection Range 3, Lower.	$C018_H$
DPR3_U	Data Segment Protection Range 3, Upper.	$C01C_H$
DPR4_L	Data Segment Protection Range 4, Lower.	$C020_H$
DPR4_U	Data Segment Protection Range 4, Upper.	$C024_H$
DPR5_L	Data Segment Protection Range 5, Lower.	$C028_H$
DPR5_U	Data Segment Protection Range 5, Upper.	$C02C_H$
DPR6_L	Data Segment Protection Range 6, Lower.	$C030_H$
DPR6_U	Data Segment Protection Range 6, Upper.	$C034_H$
DPR7_L	Data Segment Protection Range 7, Lower.	$C038_H$
DPR7_U	Data Segment Protection Range 7, Upper.	$C03C_H$
DPR8_L	Data Segment Protection Range 8, Lower.	$C040_H$
DPR8_U	Data Segment Protection Range 8, Upper.	$C044_H$
DPR9_L	Data Segment Protection Range 9, Lower.	$C048_H$
DPR9_U	Data Segment Protection Range 9, Upper.	$C04C_H$
DPR10_L	Data Segment Protection Range 10, Lower.	$C050_H$
DPR10_U	Data Segment Protection Range 10, Upper.	$C054_H$
DPR11_L	Data Segment Protection Range 11, Lower.	$C058_H$
DPR11_U	Data Segment Protection Range 11, Upper.	$C05C_H$

Table 21 Core Special Function Registers (CSFR) (cont'd)

Register Name	Description	Address Offset
DPR12_L	Data Segment Protection Range 12, Lower.	$C060_H$
DPR12_U	Data Segment Protection Range 12, Upper.	$C064_H$
DPR13_L	Data Segment Protection Range 13, Lower.	$C068_H$
DPR13_U	Data Segment Protection Range 13, Upper.	$C06C_H$
DPR14_L	Data Segment Protection Range 14, Lower.	$C070_H$
DPR14_U	Data Segment Protection Range 14, Upper.	$C074_H$
DPR15_L	Data Segment Protection Range 15, Lower.	$C078_H$
DPR15_U	Data Segment Protection Range 15, Upper.	$C07C_H$
CPR0_L	Code Segment Protection Range 0, Lower.	$D000_H$
CPR0_U	Code Segment Protection Range 0, Upper.	$D004_H$
CPR1_L	Code Segment Protection Range 1, Lower.	$D008_H$
CPR1_U	Code Segment Protection Range 1, Upper.	$D00C_H$
CPR2_L	Code Segment Protection Range 2, Lower.	$D010_H$
CPR2_U	Code Segment Protection Range 2, Upper.	$D014_H$
CPR3_L	Code Segment Protection Range 3, Lower.	$D018_H$
CPR3_U	Code Segment Protection Range 3, Upper.	$D01C_H$
CPR4_L	Code Segment Protection Range 4, Lower.	$D020_H$
CPR4_U	Code Segment Protection Range 4, Upper.	$D024_H$
CPR5_L	Code Segment Protection Range 5, Lower.	$D028_H$
CPR5_U	Code Segment Protection Range 5, Upper.	$D02C_H$
CPR6_L	Code Segment Protection Range 6, Lower.	$D030_H$
CPR6_U	Code Segment Protection Range 6, Upper.	$D034_H$
CPR7_L	Code Segment Protection Range 7, Lower.	$D038_H$
CPR7_U	Code Segment Protection Range 7, Upper.	$D03C_H$
CPR8_L	Code Segment Protection Range 8, Lower.	$D040_H$
CPR8_U	Code Segment Protection Range 8, Upper.	$D044_H$
CPR9_L	Code Segment Protection Range 9, Lower.	$D048_H$
CPR9_U	Code Segment Protection Range 9, Upper.	$D04C_H$
CPR10_L	Code Segment Protection Range 10, Lower.	$D050_H$
CPR10_U	Code Segment Protection Range 10, Upper.	$D054_H$
CPR11_L	Code Segment Protection Range 11, Lower.	$D058_H$
CPR11_U	Code Segment Protection Range 11, Upper.	$D05C_H$
CPR12_L	Code Segment Protection Range 12, Lower.	$D060_H$
CPR12_U	Code Segment Protection Range 12, Upper.	$D064_H$
CPR13_L	Code Segment Protection Range 13, Lower.	$D068_H$
CPR13_U	Code Segment Protection Range 13, Upper.	$D06C_H$
CPR14_L	Code Segment Protection Range 14, Lower.	$D070_H$
CPR14_U	Code Segment Protection Range 14, Upper.	$D074_H$
CPR15_L	Code Segment Protection Range 15, Lower.	$D078_H$
CPR15_U	Code Segment 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 21 Core Special Function Registers (CSFR) (cont'd)

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	$E40C_H$
TPS_EXTIM_ENTRY_CVAL	Temporal Protection Exception Timer Register	$E440_H$
TPS_EXTIM_ENTRY_LVAL	Temporal Protection Exception Timer Register	$E444_H$
TPS_EXTIM_EXIT_CVAL	Temporal Protection Exception Timer Register	$E448_H$
TPS_EXTIM_EXIT_LVAL	Temporal Protection Exception Timer Register	$E44C_H$
TPS_EXTIM_CLASS_EN	Temporal Protection Exception Timer Register	$E450_H$
TPS_EXTIM_STAT	Temporal Protection Exception Timer Register	$E454_H$
TPS_EXTIM_FCX	Temporal Protection Exception Timer Register	$E458_H$
Memory Management Registers (If implemented)		
MMU_CON	Memory Management Unit Configuration Register.	8000_H
MMU_ASI	MMU Address Space Identifier Register.	8004_H
MMU_TVA	MMU Translation Virtual Address Register.	$800C_H$
MMU_TPA	MMU Translation Physical Address Register.	8010_H

Table 21 Core Special Function Registers (CSFR) (cont'd)

Register Name	Description	Address Offset
MMU_TPX	MMU Translation Physical Index Register.	8014_H
MMU_TFA	MMU Translation Fault Address Register.	8018_H
MMU_TFAS	MMU Translation Fault Address Status Register.	8020_H
PMA0[1)]	Physical Memory Attributes Register 0.	$801C_H$
PMA0[1)]	Physical Memory Attributes Register 0.	8100_H
PMA1[1)]	Physical Memory Attributes Register 1.	8104_H
PMA2[1)]	Physical Memory Attributes Register 2.	8108_H
DCON2	Data Memory Configuration Register-2.	9000_H
DCON1	Data memory Configuration Register-1.	9008_H
SMACON[1)2)]	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
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 Registers		
DBGSR	Debug Status Register.	$FD00_H$
EXEVT	External Event Register.	$FD08_H$
CREVT	Core Register Event Register.	$FD0C_H$
SWEVT	Software Event Register.	$FD10_H$
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.	$F024_H$

Table 21 Core Special Function Registers (CSFR) (cont'd)

Register Name	Description	Address Offset
TR5EVT	Trigger Event 5 Register	$F028_H$
TR5ADR	Trigger Address 5 Register.	$F02C_H$
TR6EVT	Trigger Event 6 Register	$F030_H$
TR6ADR	Trigger Address 6 Register.	$F034_H$
TR7EVT	Trigger Event 7 Register	$F038_H$
TR7ADR	Trigger Address 7 Register.	$F03C_H$
TRIG_ACC	Trigger Accumulator Register.	$FD30_H$
DMS	Debug Monitor Start Address Register.	$FD40_H$
DCX	Debug Context Save Address Register.	$FD44_H$
TASK_ASI	TASK Address Space Identifier Register.	8004_H
DBGTCR	Debug Trap Control Register.	$FD48_H$
CCTRL	Counter Control Register	FC00
CCNT	CPU Clock Count Register	FC04
ICNT	Instruction Count Register	FC08
M1CNT	Multi Count Register 1	FC0C
M2CNT	Multi Count Register 2	FC10
M3CNT	Multi Count Register 3	FC14
Floating Point Registers		
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	Trapping Instruction SRC1 Operand Register.	$A010_H$
FPU_TRAP_SRC2	Trapping Instruction SRC2 Operand Register.	$A014_H$
FPU_TRAP_SRC3	Trapping Instruction SRC3 Operand Register.	$A018_H$

1) These registers are ENDINIT protected.
2) These registers are SAFETY_ENDINIT protected.

Index

Register index

Revision history

Revision history	
Page or Item	**Subjects (major changes since previous revision)**
V1.2.2, 2020-01-15	Minor typographic (typo) corrections
V1.1, 2017-24-08	Minor typographic (typo) corrections in the following chapters: • Temporal Protection System • General Purpose and System registers • Core Debug Controller
V1.0, 2016-01-11	
	Emulator Space Disable added to SYSCON register
	CDC amiguity removed, now Core Debug Controller
	Endinit register list updated
	Added “Any pending asynchronous exception may be lost when FCU condition occurs”
	Re-ordered MPP trap priority
V1.0 Draft 1, 2015-11-06	
	PSW.PRS support for 8 sets
	Number of Code and data protetion sets extended to 32
	Register definitions for TPS exception timer system
	Boot halt moved to SYSCON register

Trademarks of Infineon Technologies AG
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2020-01-15
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
Z8F54561158

IMPORTANT NOTICE

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics ("Beschaffenheitsgarantie").

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer's compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer's products and any use of the product of Infineon Technologies in customer's applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on technology, delivery terms and conditions and prices, please contact the nearest Infineon Technologies Office (**www.infineon.com**).

WARNINGS

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.