TriCore™ 1

32-bit Unified Processor Core

Volume 1:
v1.3 Core Architecture

32bit

Microcontrollers

Infineon technologies

Never stop thinking.

Edition 2005-02

Published by Infineon Technologies AG,
St.-Martin-Strasse 53,
81669 München

Attention please!

The information herein is given to describe certain components and shall not be considered as a guarantee of characteristics.

Terms of delivery and rights to technical change reserved.

We hereby disclaim any and all warranties, including but not limited to warranties of non-infringement, regarding circuits, descriptions and charts stated herein.

Information

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

Warnings

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

Infineon Technologies Components may only be used in life-support devices or systems with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

TriCore™ 1

32-bit Unified Processor Core

Volume1:
v1.3 Core Architecture

Microcontrollers

Infineon technologies

Never stop thinking.

Previous Version

Version	Subjects (major changes since last revision)
1.3.0	First release v1.3 architecture (Jan 2000).
1.3.2	General release of v1.3 architecture (August 2000).
1.3.3	Revision of chapters 3, 6, 8 and 9. Addition of appendices.
1.3.5	New information, general revision and restructuring of instruction set. Information split into two volumes: Core Architecture and Instruction Set.

We Listen to Your Comments

Is there any information in this document that you feel is wrong, unclear or missing? Your feedback will help us to continuously improve the quality of our documentation. Please send feedback (including a reference to this document) to:

ipdoc@infineon.com

Contents

Contents

Contents

1 Preface

This manual describes the TriCore architecture, Infineon's ground breaking 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. The document has been split into two volumes:

- 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 MMU and 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 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.

Additional Information

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

http://www.infineon.com/TriCore

For information and links to documentation for Infineon products that use TriCore, visit:

http://www.infineon.com/32-bit-microcontrollers

1.1 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.
- 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: $\overline{\text{BRKOUT}}$.
- Where the phrase 'Reserved Value' is used, NEVER write to this value.

In tables where register bit fields are defined, the conventions shown in **Table 1** 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 '-' in the Field and Type column.

2 Architecture Overview

This chapter gives an overview of the TriCore™ architecture.

2.1 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.

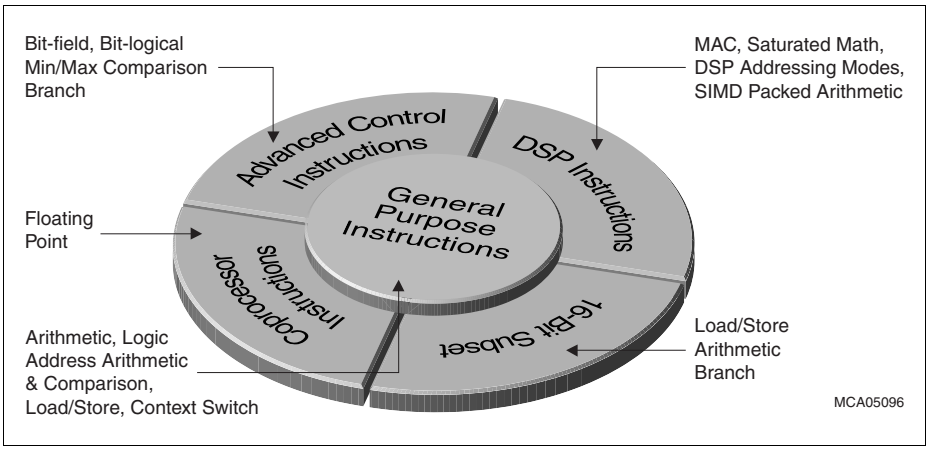


Figure 1 TriCore Architecture Overview

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.

2.1.1 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 co-processors to allow the addition of customised instructions.
- Zero overhead loop capabilities.
- Dual single-clock-cycle 16×16-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 (2×16-bit or 4×8-bit operands).
- Flexible interrupt prioritization scheme.
- Byte and bit addressing.
- Little-endian byte ordering for data memory and CPU registers.
- Memory protection.
- Debug support.

2.2 Programming Model

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

- Architectural Registers **page 2-3**.
- Data Types **page 2-4**.
- Memory Model **page 2-5**.
- Addressing Modes **page 2-5**.

The Programming Model is described in detail in the chapter **Programming Model, page 3-1**.

2.2.1 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).

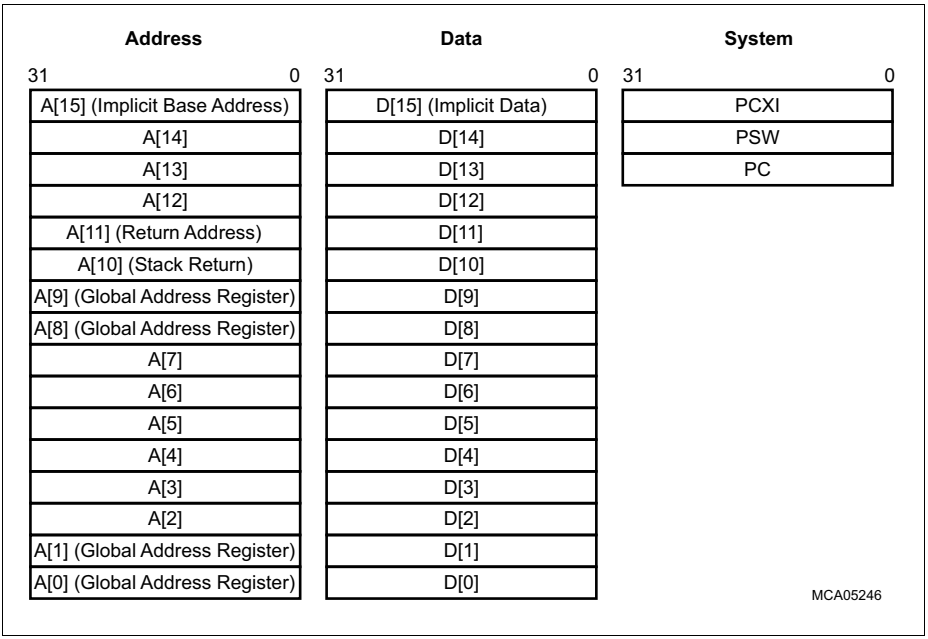


Figure 2 Architectural Registers

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).
- A[11] is the Return Address (RA) register.
- A[15] is the Implicit Address register.

Registers [0 - 7] 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, page 5-1**) and are not saved and restored across calls or interrupts. They are normally used by the operating system to reduce system overhead.

In addition to the General Purpose Registers (GPRs), the core registers are composed of a certain number of Core Special Function Registers (CSFRs). See **Core Special Function Register (CSFR) Definitions, page 4-9**.

2.2.2 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.

2.2.3 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 - F_H], each of 256 MBytes. The upper four bits of an address select the specific segment. The first 16 KBytes of each segment can be accessed directly using absolute addressing.

The diagram which follows shows the TriCore architecture address space mapping.

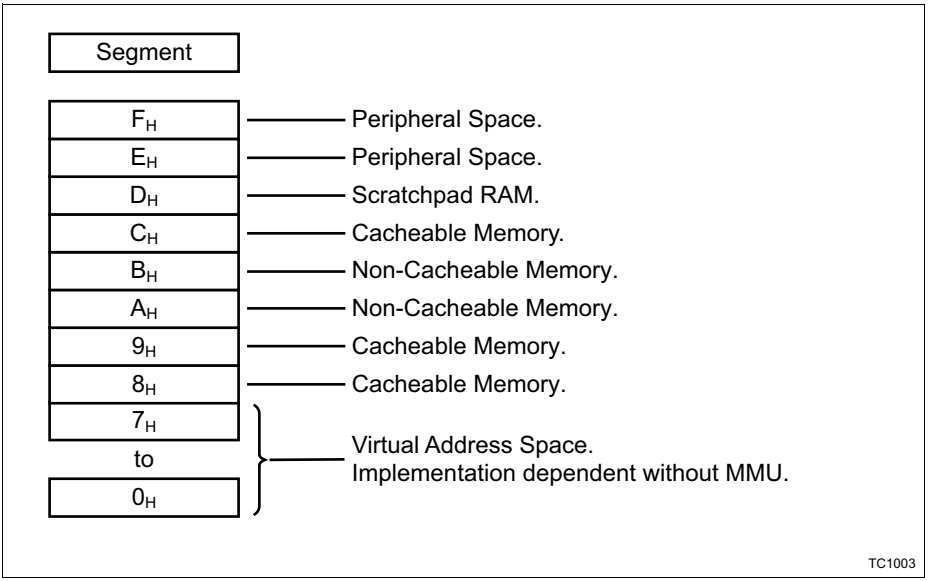


Figure 3 Address Map and Memory Model

2.2.4 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. Simple data elements are 8-bits, 16-bits, 32-bits, or 64-bits.

The TriCore 1 architecture supports seven addressing modes. 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, page 3-12**.

2.3 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.
- **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, page 5-1**.

2.4 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 however, 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.

2.4.1 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). Even the 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, page 6-1**.

2.5 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, page 7-1**.

2.6 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 unauthorized tasks from accessing critical system peripherals. The protection system also facilitates debugging. It detects and traps errors that might otherwise go unnoticed until it was too late to identify the cause of the error.

The overall protection system is composed of three main subsystems:

1. **The Trap System**: Described briefly in this chapter, **Section 2.5**, **page 2-8**, but covered in detail in **Trap System, page 7-1**.
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.
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.

For TriCore v1.3 and later architecture revisions, there are actually two independent memory protection systems. 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. The MMU is detailed in **Memory Management Unit (MMU), page 10-1**.

For the smaller and lower cost applications there is a range-based memory protection system. This is designed to provide course-grained memory protection for systems that do not require virtual memory and which do not want to incur the die area and performance cost of address translation in an MMU. The range-based memory protection system and its interaction with I/O privilege level for access to peripherals, is detailed in **Memory Protection System, page 9-1**.

2.7 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 or 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 enabled.

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.
- **Page Table Entry (PTE) Translation**: 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. There are six memory-mapped Core Special Function Registers (CSFRs), two of which control the memory management system.

See **Memory Management Unit (MMU), page 10-1**.

2.8 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 mechanism and registers is detailed in **Core Debug Controller (CDC), page 11-1**.

3 Programming Model

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

- Supported data types **page 3-1**.
- Data formats in registers and memory **page 3-2**.
- The Memory model **page 3-6**.
- Addressing modes **page 3-7**.

3.1 Data Types

The instruction set supports operations on the following Data Types:

- Boolean **page 3-1**.
- Bit String **page 3-1**.
- Byte **page 3-1**.
- Signed Fraction **page 3-2**.
- Address **page 3-2**.
- Signed and Unsigned Integers **page 3-2**.
- IEEE-754 Single-precision Floating-point Number **page 3-2**.

Most instructions operate on a specific Data Type, while others are useful for manipulating several Data Types.

3.1.1 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.

3.1.2 Bit String

A bit string is a packed field of bits.

Bit strings are produced and used by logical, shift, and bit field instructions.

3.1.3 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.

3.1.4 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).

3.1.5 Address

An address is a 32-bit unsigned value.

3.1.6 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.

3.1.7 IEEE-754 Single-Precision Floating-Point Number

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

3.2 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.
- Word: signed, unsigned, fraction, floating-point.
- 48-bit: signed, unsigned, fraction, MAC accumulator.
- Double-word: signed, unsigned, fraction.

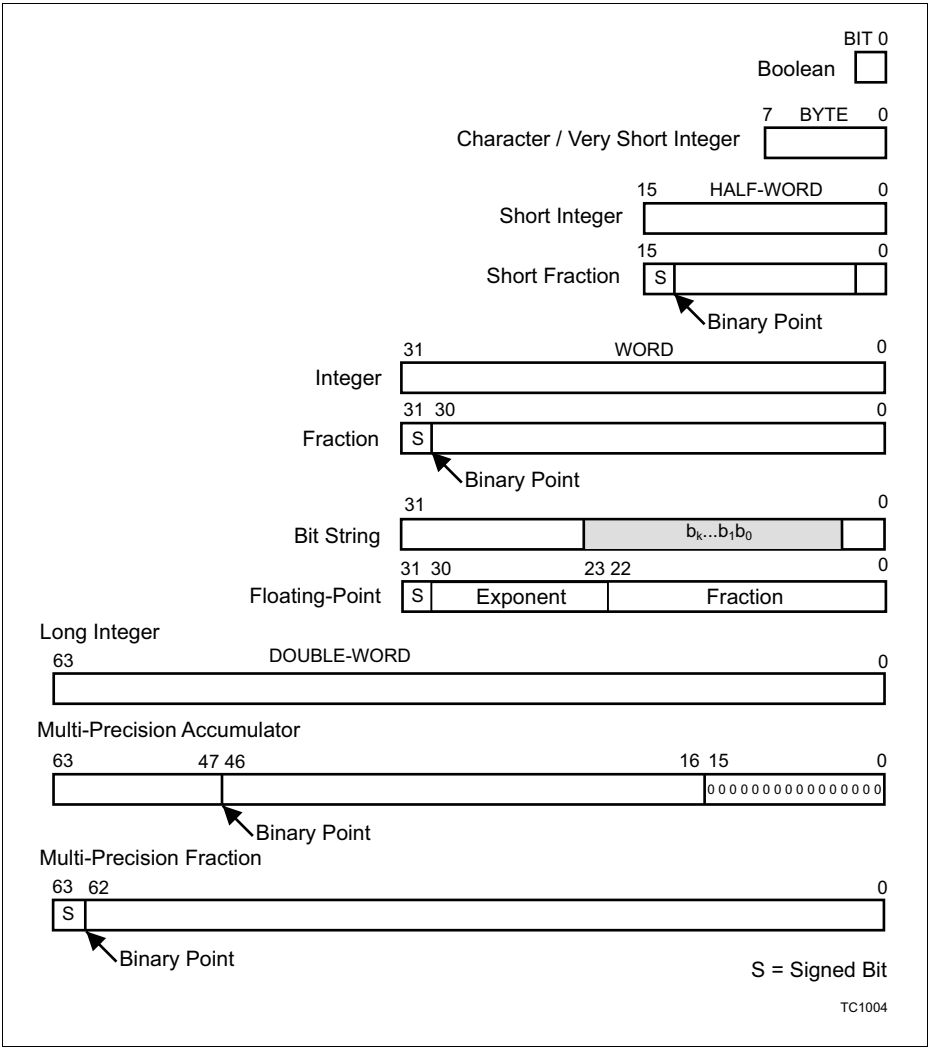


Figure 4 Supported Data Formats

3.2.1 Alignment Requirements

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

Data can be aligned on any half-word boundary, regardless of size. 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.

There are some restrictions of which programmers must be aware, specifically:

- The LDMST and SWAP instructions require their operands to be word-aligned.
- Half-word alignment for LD.D and ST.D is only allowed when the source or destination address is targeted at cached memory or data scratchpad RAM (see **Scratchpad RAM, page 8-4**). For all other addresses double-word accesses must be word-aligned.
- Byte operations LD.B, ST.B, LD.BU, ST.T may be byte aligned.

Table 2 Alignment Rules

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)
ST.T	Byte	Byte (1_H)
Context Load / Store / Restore / Save	16 x 32-bit registers	64 bytes (40_H)

3.2.2 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.

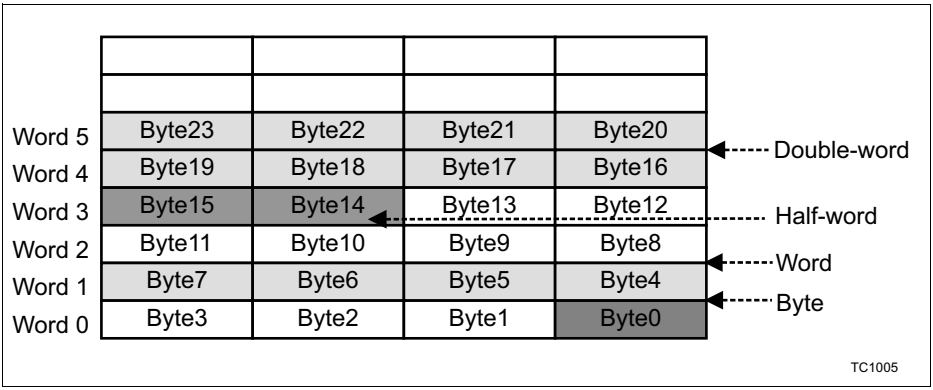


Figure 5 Byte Ordering

3.3 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 - 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 either absolute addressing or absolute bit 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, page 7-1** 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 - 7] 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 have the peripheral space attribute and therefore all accesses are non-speculative and are not accessible to User-0 mode. This segment can therefore be used for mapping peripheral registers.

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), page 8-1**.

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.

3.4 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. Simple data elements are 8, 16, 32 or 64 bits wide.

The TriCore 1 architecture supports seven addressing modes. The addressing modes were selected to 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

The instruction formats were chosen to 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 should be noted that 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.

3.4.1 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 6**). Other bits are zero-filled.

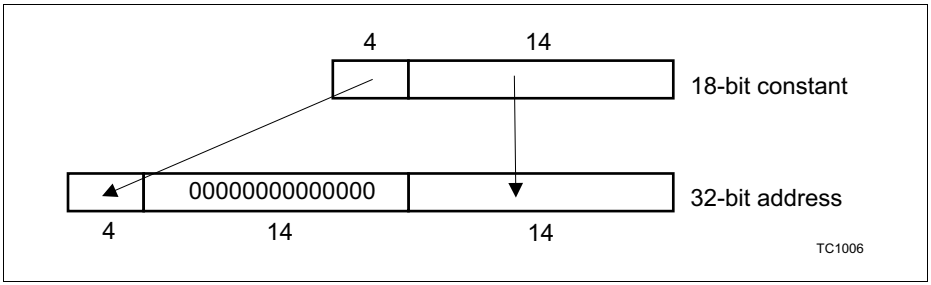


Figure 6 Translation of Absolute Address to Full Effective Address

3.4.2 Base + Offset Addressing

Base + offset 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.

3.4.3 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.

3.4.4 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.

3.4.5 Circular Addressing

The primary use of circular addressing (**Figure 7**) is for accessing data values in circular buffers while performing filter calculations.

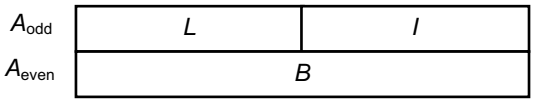


Figure 7 Circular Addressing Mode

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 8 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 9**), 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.

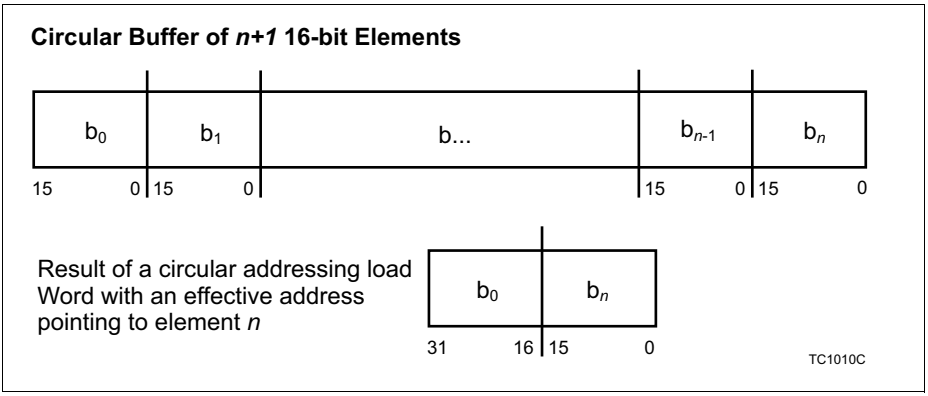


Figure 9 Circular Buffer End Case

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).

3.4.6 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, page 3-11**).

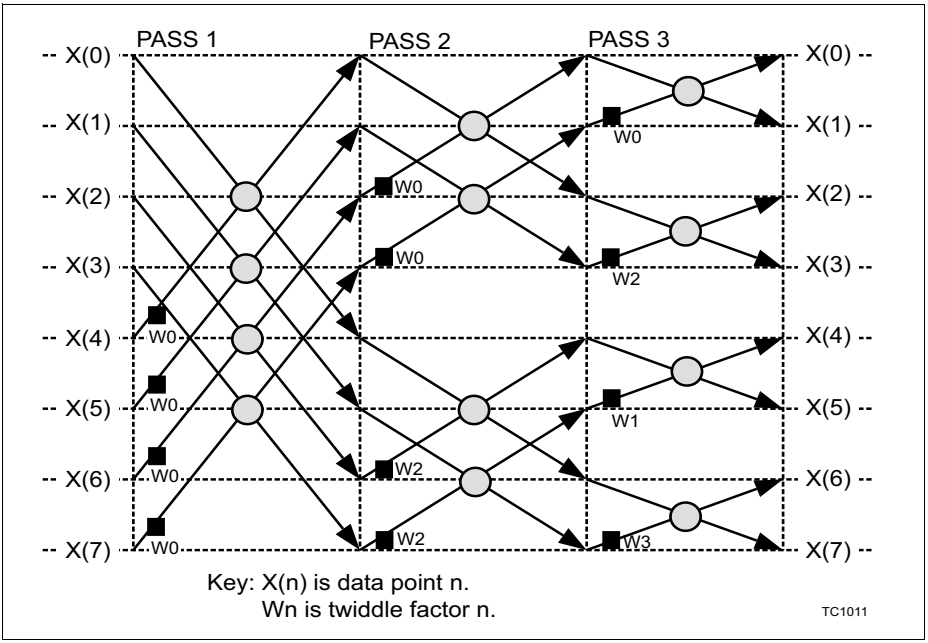


Figure 10 Bit-Reverse Addressing

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

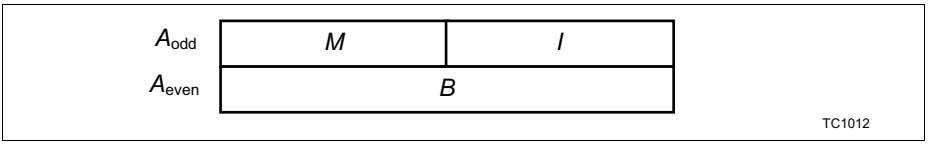


Figure 11 Register Pair for Bit-Reverse Addressing

- 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
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.

3.4.7 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

Indexed addressing 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.

4 Core 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. GPRs are described in **General Purpose Registers (GPRs), page 4-7**.
- The CSFRs control the operation of the core and provide status information about the core. CSFRs are described in **Core Special Function Register (CSFR) Definitions, page 4-9**.

4.1 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 (BTV, BIV and ISR), by only allowing them to be changed in the initialisation state.

4.2 Core Register Table

The following table lists all the TriCore™ CSFRs and GPRs.

The memory protection system is modular and the actual number of registers is implementation-specific.

Table 6 Core Register Map

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.	$FFBC_H$
PCXI	Previous Context Information Register.	$FE00_H$
PSW	Program Status Word Register.	$FE04_H$

Table 6 Core Register Map (Continued)

Register Name	Description	Address Offset
PC	Program Counter Register.	$FE08_H$
SYSCON	System Configuration Register.	$FE14_H$
CPU_ID	CPU Identification Register (Read Only).	$FE18_H$
BIV [2)]	Base Address of Interrupt Vector Table Register.	$FE20_H$
BTV [2)]	Base Address of Trap Vector Table Register.	$FE24_H$
ISP [2)]	Interrupt Stack Pointer Register.	$FE28_H$
ICR	Interrupt Control Register.	$FE2C_H$
FCX	Free Context List Head Pointer Register.	$FE38_H$
LCX	Free Context List Limit Pointer Register.	$FE3C_H$
DPR0_0L	Data Segment Protection Register 0, Set 0, Lower.	$C000_H$
DPR0_0U	Data Segment Protection Register 0, Set 0, Upper.	$C004_H$
DPR0_1L	Data Segment Protection Register 1, Set 0, Lower.	$C008_H$
DPR0_1U	Data Segment Protection Register 1, Set 0, Upper.	$C00C_H$
DPR0_2L	Data Segment Protection Register 2, Set 0, Lower.	$C010_H$
DPR0_2U	Data Segment Protection Register 2, Set 0, Upper.	$C014_H$
DPR0_3L	Data Segment Protection Register 3, Set 0, Lower.	$C018_H$
DPR0_3U	Data Segment Protection Register 3, Set 0, Upper.	$C01C_H$
DPR1_0L	Data Segment Protection Register 0, Set 1, Lower.	$C400_H$
DPR1_0U	Data Segment Protection Register 0, Set 1, Upper.	$C404_H$
DPR1_1L	Data Segment Protection Register 1, Set 1, Lower.	$C408_H$
DPR1_1U	Data Segment Protection Register 1, Set 1, Upper.	$C40C_H$
DPR1_2L	Data Segment Protection Register 2, Set 1, Lower.	$C410_H$
DPR1_2U	Data Segment Protection Register 2, Set 1, Upper.	$C414_H$
DPR1_3L	Data Segment Protection Register 3, Set 1, Lower.	$C418_H$
DPR1_3U	Data Segment Protection Register 3, Set 1, Upper.	$C41C_H$
DPR2_0L	Data Segment Protection Register 0, Set 2, Lower.	$C800_H$
DPR2_0U	Data Segment Protection Register 0, Set 2, Upper.	$C804_H$
DPR2_1L	Data Segment Protection Register 1, Set 2, Lower.	$C808_H$
DPR2_1U	Data Segment Protection Register 1, Set 2, Upper.	$C80C_H$
DPR2_2L	Data Segment Protection Register 2, Set 2, Lower.	$C810_H$
DPR2_2U	Data Segment Protection Register 2, Set 2, Upper.	$C814_H$
DPR2_3L	Data Segment Protection Register 3, Set 2, Lower.	$C818_H$
DPR2_3U	Data Segment Protection Register 3, Set 2, Upper.	$C81C_H$

Table 6 Core Register Map (Continued)

Register Name	Description	Address Offset
DPR3_0L	Data Segment Protection Register 0, Set 3, Lower.	$CC00_H$
DPR3_0U	Data Segment Protection Register 0, Set 3, Upper.	$CC04_H$
DPR3_1L	Data Segment Protection Register 1, Set 3, Lower.	$CC08_H$
DPR3_1U	Data Segment Protection Register 1, Set 3, Upper.	$CC0C_H$
DPR3_2L	Data Segment Protection Register 2, Set 3, Lower.	$CC10_H$
DPR3_2U	Data Segment Protection Register 2, Set 3, Upper.	$CC14_H$
DPR3_3L	Data Segment Protection Register 3, Set 3, Lower.	$CC18_H$
DPR3_3U	Data Segment Protection Register 3, Set 3, Upper.	$CC1C_H$
CPR0_0L	Code Segment Protection Register 0, Set 0, Lower.	$D000_H$
CPR0_0U	Code Segment Protection Register 0, Set 0, Upper.	$D004_H$
CPR0_1L	Code Segment Protection Register 1, Set 0, Lower.	$D008_H$
CPR0_1U	Code Segment Protection Register 1, Set 0, Upper.	$D00C_H$
CPR0_2L	Code Segment Protection Register 2, Set 0, Lower.	$D010_H$
CPR0_2U	Code Segment Protection Register 2, Set 0, Upper.	$D014_H$
CPR0_3L	Code Segment Protection Register 3, Set 0, Lower.	$D018_H$
CPR0_3U	Code Segment Protection Register 3, Set 0, Upper.	$D01C_H$
CPR1_0L	Code Segment Protection Register 0, Set 1, Lower.	$D400_H$
CPR1_0U	Code Segment Protection Register 0, Set 1, Upper.	$D404_H$
CPR1_1L	Code Segment Protection Register 1, Set 1, Lower.	$D408_H$
CPR1_1U	Code Segment Protection Register 1, Set 1, Upper.	$D40C_H$
CPR1_2L	Code Segment Protection Register 2, Set 1, Lower.	$D410_H$
CPR1_2U	Code Segment Protection Register 2, Set 1, Upper.	$D414_H$
CPR1_3L	Code Segment Protection Register 3, Set 1, Lower.	$D418_H$
CPR1_3U	Code Segment Protection Register 3, Set 1, Upper.	$D41C_H$
CPR2_0L	Code Segment Protection Register 0, Set 2, Lower.	$D800_H$
CPR2_0U	Code Segment Protection Register 0, Set 2, Upper.	$D800_H$
CPR2_1L	Code Segment Protection Register 1, Set 2, Lower.	$D804_H$
CPR2_1U	Code Segment Protection Register 1, Set 2, Upper.	$D80C_H$
CPR2_2L	Code Segment Protection Register 2, Set 2, Lower.	$D810_H$
CPR2_2U	Code Segment Protection Register 2, Set 2, Upper.	$D814_H$
CPR2_3L	Code Segment Protection Register 3, Set 2, Lower.	$D818_H$
CPR2_3U	Code Segment Protection Register 3, Set 2, Upper.	$D81C_H$

Table 6 Core Register Map (Continued)

Register Name	Description	Address Offset
CPR3_0L	Code Segment Protection Register 0, Set 3, Lower.	$DC00_H$
CPR3_0U	Code Segment Protection Register 0, Set 3, Upper.	$DC04_H$
CPR3_1L	Code Segment Protection Register 1, Set 3, Lower.	$DC08_H$
CPR3_1U	Code Segment Protection Register 1, Set 3, Upper.	$DC0C_H$
CPR3_2L	Code Segment Protection Register 2, Set 3, Lower.	$DC10_H$
CPR3_2U	Code Segment Protection Register 2, Set 3, Upper.	$DC14_H$
CPR3_3L	Code Segment Protection Register 3, Set 3, Lower.	$DC18_H$
CPR3_3U	Code Segment Protection Register 3, Set 3, Upper.	$DC1C_H$
DPM0	Data Protection Mode Register 0.	$E000_H$
DPM1	Data Protection Mode Register 1.	$E080_H$
DPM2	Data Protection Mode Register 2.	$E100_H$
DPM3	Data Protection Mode Register 3.	$E180_H$
CPM0	Code Protection Mode Register 0.	$E200_H$
CPM1	Code Protection Mode Register 1.	$E280_H$
CPM2	Code Protection Mode Register 2.	$E300_H$
CPM3	Code Protection Mode Register 3.	$E380_H$
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
MMU_TPX	MMU Translation Physical Index Register.	8014_H
MMU_TFA	MMU Translation Fault Address Register.	8018_H
CPU_SRC0	CPU Service Request Control Register 0.	$FFFC_H$
CPU_SRC1	CPU Service Request Control Register 1.	$FFF8_H$
CPU_SRC2	CPU Service Request Control Register 2.	$FFF4_H$
CPU_SRC3	CPU Service Request Control Register 3.	$FFF0_H$
CPU_SBSRC0	CPU Software Break Service Request Control Register 0.	$FFBC_H$

Table 6 Core Register Map (Continued)

Register Name	Description	Address Offset
CPU_SBSRC1	CPU Software Break Service Request Control Register 1.	$FFB8_H$
CPU_SBSRC2	CPU Software Break Service Request Control Register 2.	$FFB4_H$
CPU_SBSRC3	CPU Software Break Service Request Control Register 3.	$FFB0_H$
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.	$FD10_H$
TR0EVT	Trigger Event 0 Register.	$FD20_H$
TR1EVT	Trigger Event 1 Register.	$FD24_H$
DMS	Debug Monitor Start Address Register.	$FD40_H$
DCX	Debug Context Save Area Pointer Register.	$FD44_H$

1) These Address Offsets are not used by the MTCR instruction.

2) These registers are ENDINIT protected.

4.3 General Purpose Registers (GPRs)

Figure 12, page 4-8, shows the 32-bit wide GPRs. The GPRs are split evenly into:

- 16 Data registers (DGPRs), D[0] to D[15].
- 16 Address registers (AGPRs), A[0] to A[15].

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, page 4-21**.

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.

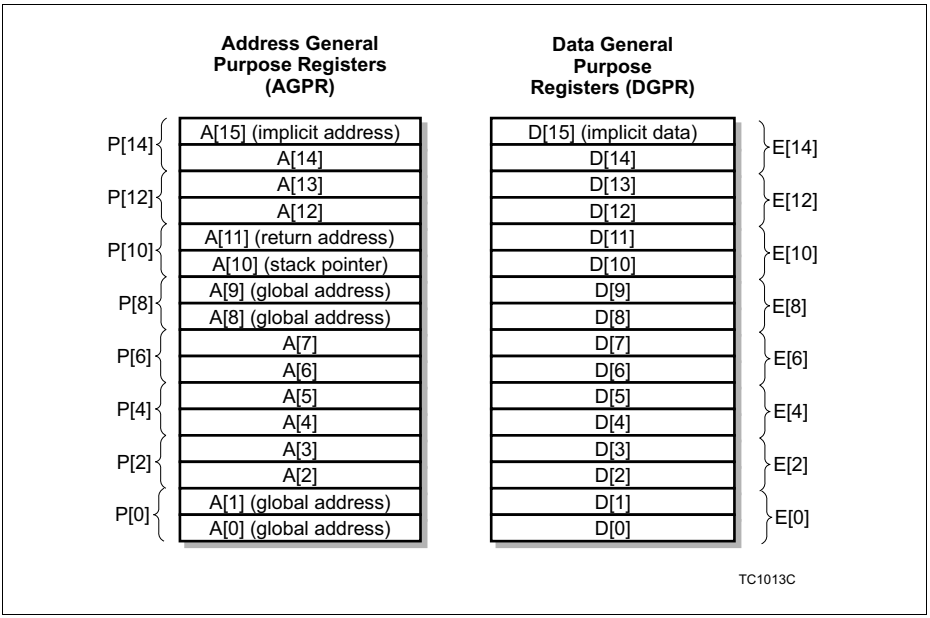


Figure 12 General Purpose Registers (GPRs)

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 5 Tasks and Functions****.*

4.4 Core Special Function Register (CSFR) Definitions

The remainder of this chapter consists of CSFR register definitions.

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.

The CSFRs described are:

- Program State Information **page 4-2**.
  – PC, PSW, and PCXI.
- Context Management **page 4-17**.
  – FCX, PCX, and LCX.
- Stack Management **page 4-21**.
  – A[10](SP), and ISP.
- Interrupt and Trap Control **page 4-23**.
  – ICR, BIV, and BTV.
- System Control - SYSCON **page 4-27**.
- CPU Identification **page 4-28**.
- Memory Protection **page 4-29**.
- Memory Management Unit (MMU) **page 4-29**.
- Core Debug Controller (CDC) **page 4-30**.

4.5 Program State Information

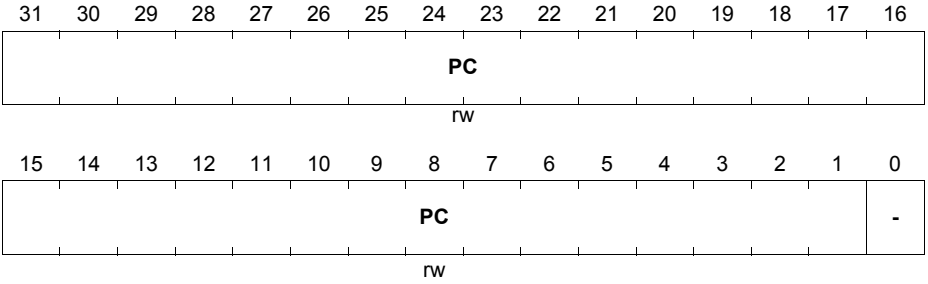
The PC, PSW, and PCXI registers hold and reflect program state information. These registers are an important part of storing and restoring a task's context, when the contents are stored, restored or modified during this process.

- PC: Program Counter page 4-10.
- PSW: Program Status Word page 4-11.
- PCXI: Previous Context Information page 4-16.

4.5.1 Program Counter (PC)

The 32-bit Program Counter (PC) shown below, holds the address of the instruction that is currently running. The Program Counter is part of a task's state information.

PC
Program Counter Register **Reset Value: Implementation Specific**



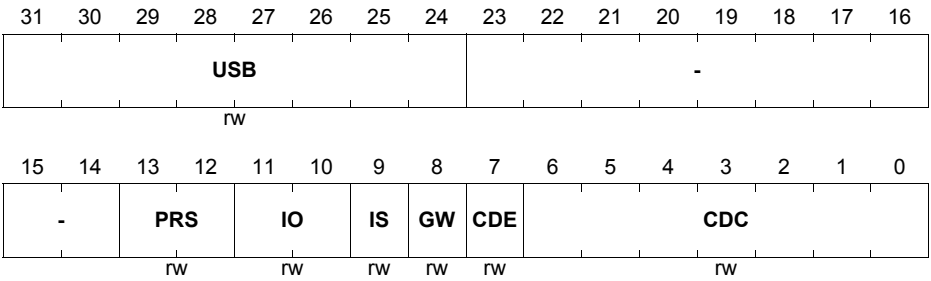
Field	Bits	Type	Description
PC	[31:1]	rw	**Program Counter**
-	0	-	**Reserved Field**

4.5.2 Program Status Word (PSW)

The Program Status Word (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 **Reset Value: 0000 0B80$_H$**



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.
-	[23:14]	-	**Reserved Field**
PRS	[13:12]	rw	**Protection Register Set** 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. 00_B : Protection Register Set 0. 01_B : Protection Register Set 1. 10_B : Protection Register Set 2. 11_B : Protection Register Set 3.

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. 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).

Field	Bits	Type	Description
GW	8	rw	**Global Address Register Write Permission** Determines whether the current execution thread has permission to modify the global address registers. Most tasks and ISRs use the global address registers as 'read only' registers, pointing to the global literal pool and key data structures. However a task or ISR can be designated as the 'owner' of a particular global address register, and is allowed to modify it. The system designer must determine which global address variables are used with sufficient frequency and/or in sufficiently time-critical code to justify allocation to a global address register. By compiler convention, global address register A[0] is reserved as the base register for short form loads and stores. Register A[1] is also reserved for compiler use. Registers A[8] and A[9] are not used by the compiler, and are available for holding critical system address variables. 0 : Write permission to global registers A[0], A[1], A[8], A[9] is disabled. 1 : Write permission to global registers A[0], A[1], A[8], A[9] is enabled.
CDE	7	rw	**Call Depth Count Enable** Enables call-depth counting, provided that the PSW.CDC mask field is not all set to 1. PSW.CDE is set to 1 by default, but should be cleared by the SYSCALL instruction Trap Service Routine to allow a trapped SYSCALL instruction to execute without producing another trap upon return from the trap handler. It is then set again when the next SYSCALL instruction is executed. 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.

Field	Bits	Type	Description
CDC	[6:0]	rw	**Call Depth Counter Overflow** Consists of two variable-width subfields. The first subfield is a mask field, consisting of a string of zero or more initial 1 bits, terminated by the first 0 bit. The remaining bits comprise the subfield, which constitutes the Call Depth Counter (CDC). $0ccccc_B$: 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 counter overflows, a trap is generated. Depending on the width of the mask field, the call depth counter can be set to overflow at any power of two boundary, from 1 to 64. Setting the mask field to 1111110_B allows no bits for the counter, and causes every call to be trapped. This is used for call tracing. Setting the mask field 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 7 PSW User Status Bits

Field	Bits	Type	Description
C	31	rw	Carry.
V	30	rw	Overflow.
SV	29	rw	Sticky Overflow.
AV	28	rw	Advance Overflow.
SAV	27	rw	Sticky Advance Overflow.
-	[26:24]	-	Reserved Field.

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

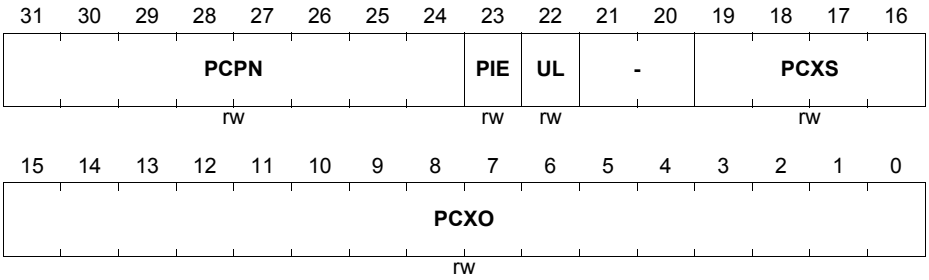
- Arithmetic instructions that may produce carry and overflow results.
- Implementation-specific coprocessor instructions which may use any or all of the eight bits, in a manner that is entirely implementation specific.

Bits [23:16] of the PSW are reserved bits 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.

4.5.3 Previous Context Information Register (PCXI)

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

PCXI
Previous Context Information **Reset Value: Implementation Specific**



Field	Bits	Type	Description
PCPN	[31:24]	rw	**Previous CPU Priority Number** Contains the priority level number of the interrupted task.
PIE	23	rw	**Previous Interrupt Enable** Indicates the state of the interrupt enable bit (ICR.IE) for the interrupted task.
UL	22	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.
-	[21:20]	-	**Reserved Field**
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.

4.6 Context Management Registers

The context management registers comprise of three pointers that handle context management and are used during context save/restore operations.

- FCX: Free CSA List Head Pointer **page 4-18**.
- PCX: Previous Context Pointer **page 4-19**.
- LCX: Free CSA List Limit Pointer **page 4-20**.

Each pointer consists of two fields:

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

Table 13 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.

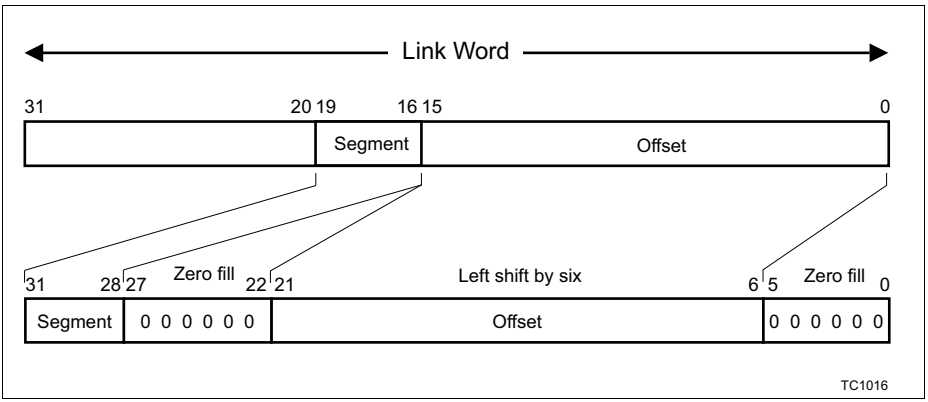


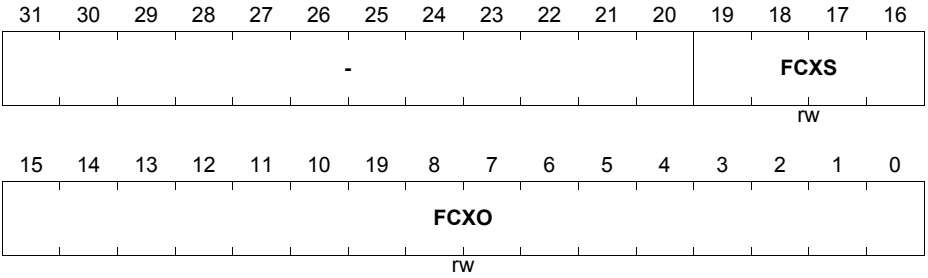
Figure 13 Generation of the Effective Address of a Context Save Area (CSA)

*Note: See **Context Save Area, page 5-3** for additional constraints on the Effective Address (EA).*

4.6.1 Free CSA List Head Pointer (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 **Reset Value: Implementation Specific**

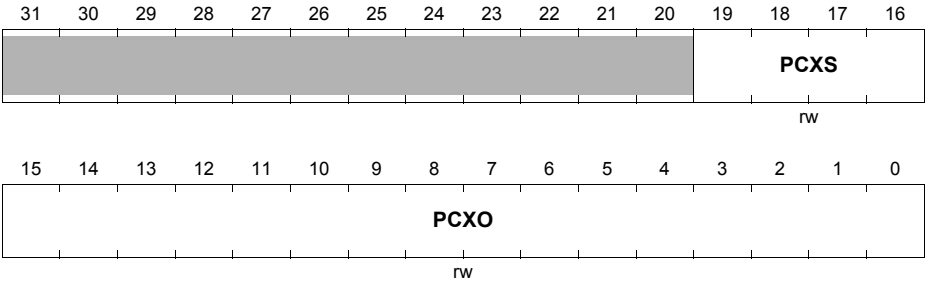


Field	Bits	Type	Description
-	[31:20]	-	**Reserved Field**
FCXS	[19:16]	rw	**FCX Segment Address Field** Used in conjunction with the FCXO field.
FCXO	[15:0]	rw	**FCX Offset Address Field** The FCXO and FCXS fields together form the FCX pointer, which points to the next available CSA.

4.6.2 Previous Context Pointer (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 **Reset Value: Implementation Specific**



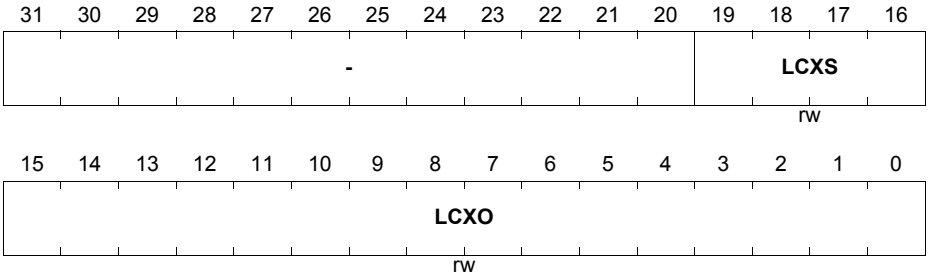
Field	Bits	Type	Description
	[31:20]		These bits (shaded) are not relevant to the pointer function and so are not described here.
PCXS	[19:16]	rw	**PCX Segment Address Field** This field is used in conjunction with the PCXO field.
PCXO	[15:0]	rw	**Previous Context Pointer Offset Field** The PCXO and PCXS fields form the pointer PCX, which points to the CSA of the previous context.

4.6.3 Free CSA List Limit Pointer (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), page 7-10****.*

LCX
Free CSA List Limit Pointer **Reset Value: Implementation Specific**



Field	Bits	Type	Description
-	[31:20]	-	**Reserved Field**
LCXS	[19:16]	rw	**LCX Segment Address** This field is used in conjunction with the LCXO field.
LCXO	[15:0]	rw	**LCX Offset Field** The LCXO and LCXS fields form the pointer LCX, which points to the last available CSA.

4.7 Stack Management

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](SP): A[10](Stack Pointer) **page 4-22**.
- ISP: Interrupt Stack Pointer **page 4-22**.

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 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.

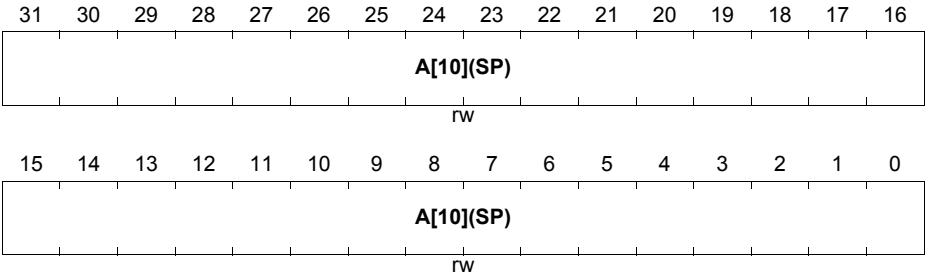
Usually it is only necessary to initialize the ISP once during the initialization routine. However, depending on application needs, the ISP can be modified during execution. Note that there is nothing preventing an ISR or system service routine from executing on a private stack.

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

4.7.1 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) **Reset Value: Implementation Specific**

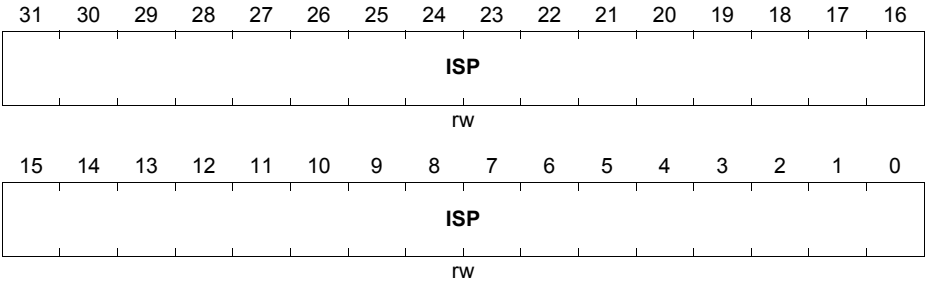


Field	Bits	Type	Description
A[10](SP)	[31:0]	rw	**Address Register A[10] (Stack Pointer)**

4.7.2 Interrupt Stack Pointer (ISP)

The Interrupt Stack Pointer is defined as follows:

ISP
Interrupt Stack Pointer **Reset Value: Implementation Specific**



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

4.8 Interrupt and Trap Control

Three CSFRs support interrupt and trap handling:

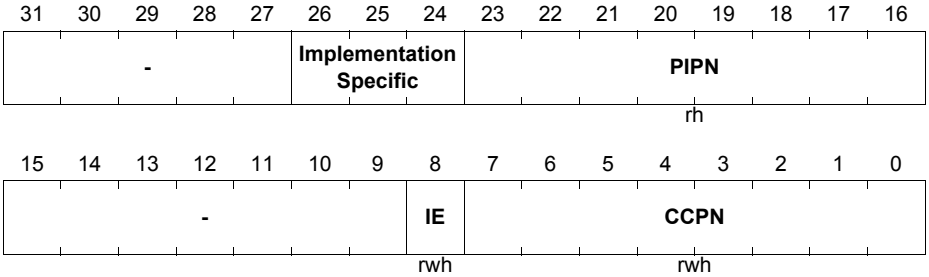
- ICR: Interrupt Control Register **page 4-23**.
- BIV: Base Interrupt Vector Table Pointer **page 4-25**.
- BTV: Base Trap Vector Table Pointer **page 4-26**.

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 other two registers hold the base addresses for the interrupt (BIV) and trap vector tables (BTV). Special instructions control the enabling and disabling of the interrupt system. For more information see **Interrupt System, page 6-1**.

4.8.1 Interrupt Control Register (ICR)

The Interrupt Control register is defined as follows:

ICR
Interrupt Control **Reset Value: 0000 0000$_H$**



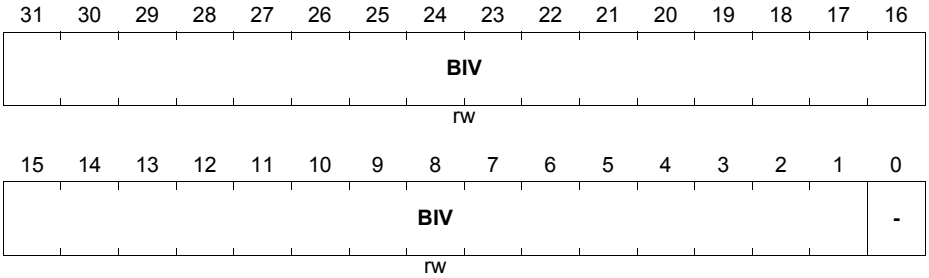
Field	Bits	Type	Function
-	[31:27]	-	**Reserved Field**
	[26:24]	-	**Implementation Specific** Control of the arbitration. See the relevant documentation for a specific TriCore product implementation.

Field	Bits	Type	Function
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.
-	[15:9]	-	**Reserved Field**
IE	8	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.
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.

4.8.2 Base Interrupt Vector Table Pointer (BIV)

The BIV register contains the base address of the interrupt vector table. When an interrupt is accepted, the entry address into the interrupt vector table is generated from the priority number (taken from the PIPN) of that interrupt, left shifted by 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 8 words (32 bytes) between the individual entries in the vector table.

BIV
Base Interrupt Vector Table Pointer **Reset Value: Implementation Specific**

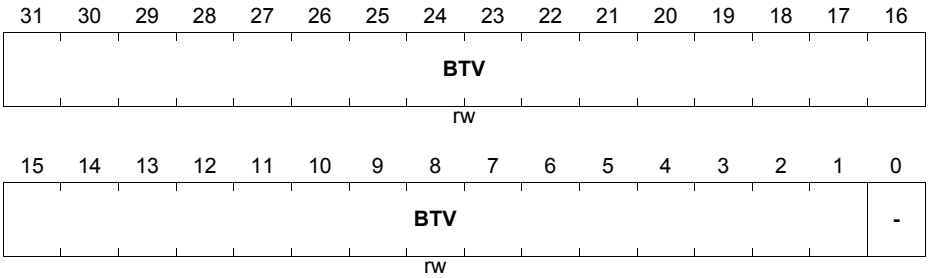


Field	Bits	Type	Description
BIV	[31:1]	rw	**Base Address of Interrupt Vector Table** The address in the BIV register must be aligned to an even byte address (half-word address). Because of the simple ORing of the left-shifted priority number and the contents of the BIV register, the alignment of the base address of the vector table must be to a power of two boundary, dependent on the number of interrupt entries used. For the full range of 256 interrupt entries an alignment to an 8 KByte boundary is required. If fewer sources are used, the alignment requirements are correspondingly relaxed.
-	0	-	**Reserved Field**

4.8.3 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.

BTV
Base Trap Vector Table Pointer **Reset Value: Implementation Specific**

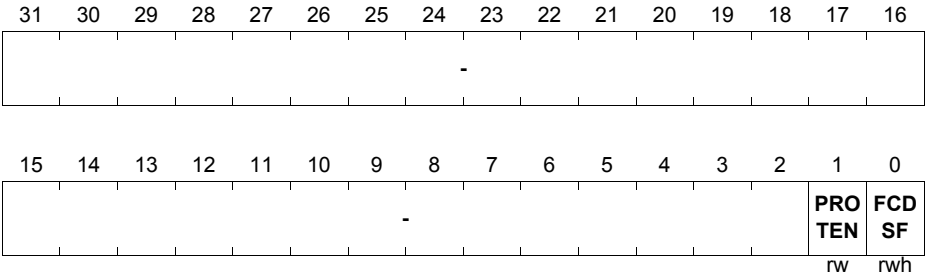


Field	Bits	Type	Description
BTV	[31:1]	rw	**Base Address of Trap Vector Table** The address in the BTV register must be aligned to an even byte address (half-word address). Also, due to the simple ORing of the left-shifted trap identification number and the contents of the BTV register, the alignment 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).
-	0	-	**Reserved Field**

4.9 System Control Registers (SYSCON)

The System Configuration Register provides the enable/disable bit for the memory protection system and a status flag for the Free Context List Depletion condition.

SYSCON
System Configuration Register **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
-	[31:2]	-	**Reserved Field**
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.

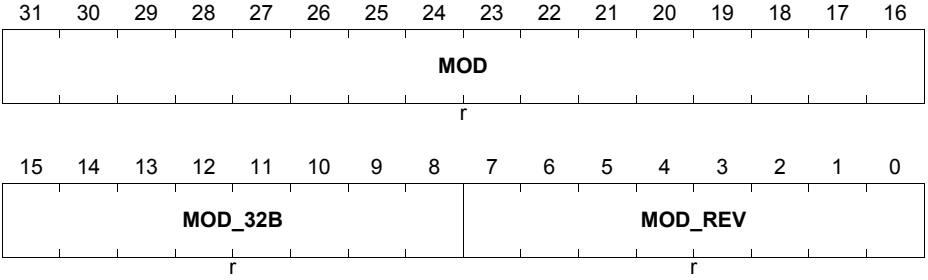
4.10 ID Registers

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.

4.10.1 CPU Identification Register (CPU_ID)

The CPU Identification Register identifies the CPU type and revision.

CPU_ID
CPU Module Identification **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.

4.11 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 CSFRs which control the Interrupts are described in **Interrupt System, page 6-1**.

- SRCn: Service Request Control **page 6-3**.
- ICR: Interrupt Control **page 4-23**.

4.12 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 four 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 CSFRs which control the Memory Protection Registers are described in **Memory Protection System, page 9-1**.

- DPRx_nU: Data Segment Protection Register Pair - Upper Bound **page 9-4**.
- DPRx_nL: Data Segment Protection Register Pair - Lower Bound **page 9-4**.
- CPRx_nU: Code Segment Protection Register Pair- Upper Bound **page 9-5**.
- CPRx_nL: Code Segment Protection Register Pair- Lower Bound **page 9-5**.
- DPMx: Data Protection Mode Register **page 9-6**.
- CPMx: Code Protection Mode Register **page 9-8**.

4.13 Memory Management Unit Registers

The optional Memory Management Unit (MMU) supports virtual memory and page-based memory access protection.

The CSFRs which control the optional MMU are described in **Memory Management Unit (MMU), page 10-1**.

- MMU_CON: Configuration Register **page 10-13**.
- MMU_ASI: Address Space Identifier **page 10-15**.
- MMU_TVA: Translation Virtual Address Register **page 10-16**.
- MMU_TPA: Translation Physical Address Register **page 10-17**.
- MMU_TPX: Translation Page Index Register **page 10-19**.
- MMU_TFA: Translation Fault Page Address Register **page 10-20**.

4.14 Core Debug Controller Registers

The core registers that support debugging define the conditions under which a Debug Event is generated and the Debug Actions taken on the assertion of a Debug Event, as well as providing status information on the Core Debug Controller (CDC). Some of these functions are implementation specific.

These registers are described in **Core Debug Controller (CDC), page 11-1**.

- DBGSR: Debug Status Register **DBGSR, page 11-13**.
- CREVT: Core Register Access Event Register **CREVT, page 11-16**.
- SWEVT: Software Debug Event Register **SWEVT, page 11-17**.
- EXEVT: External Event Register **EXEVT, page 11-15**.
- TRnEVT: Trigger Event Register **TR0EVT, page 11-18**.
- DMS: Debug Monitor Start Address Register **DMS, page 11-21**.
- DCX: Debug Context Save Area Pointer **DCX, page 11-21**.
- SBSRCn: Software Breakpoint Service Request Control Register **page 11-22**.

5 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 however, 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. These are discussed together in this chapter.

5.1 Upper and Lower Contexts

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.

There are three sub-context types:

- 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 (i.e. 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).

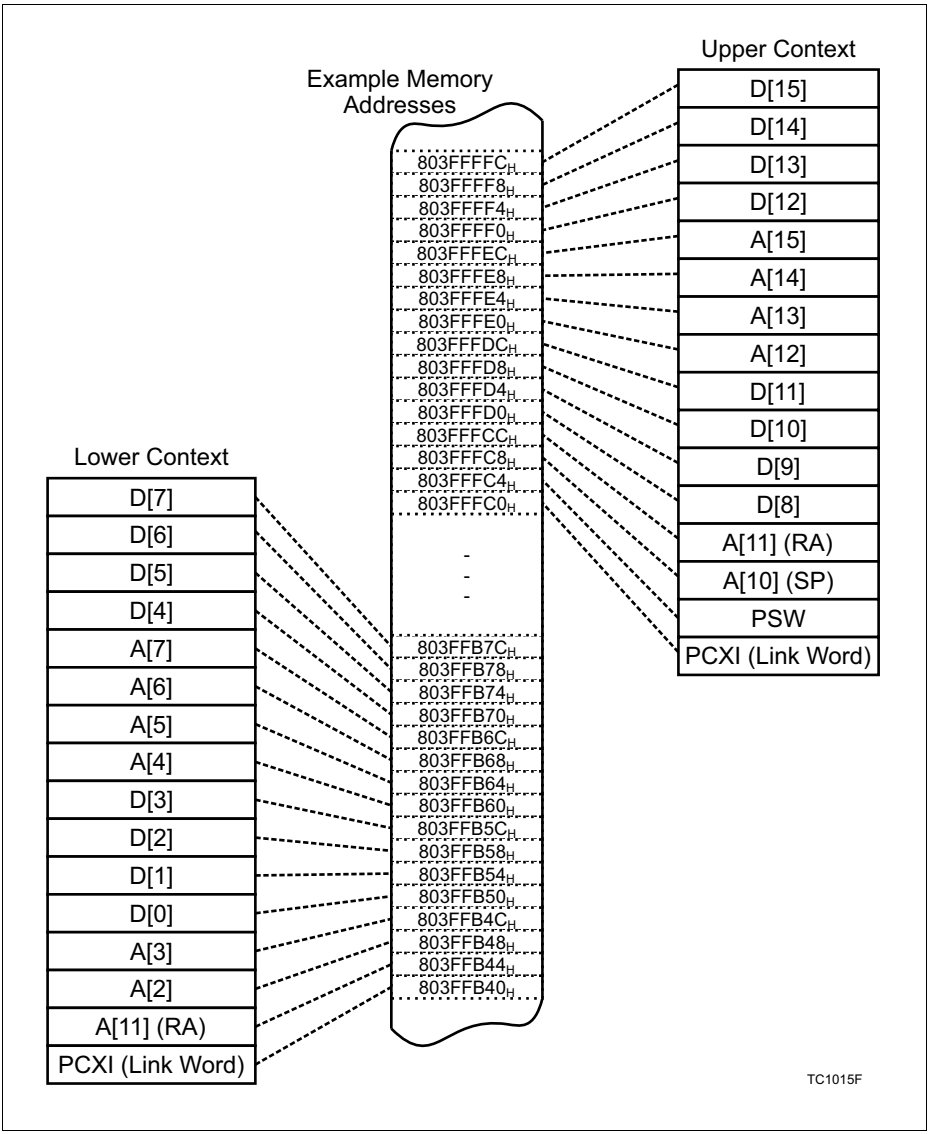


Figure 14 Upper and Lower Contexts

5.1.1 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 15**.

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.

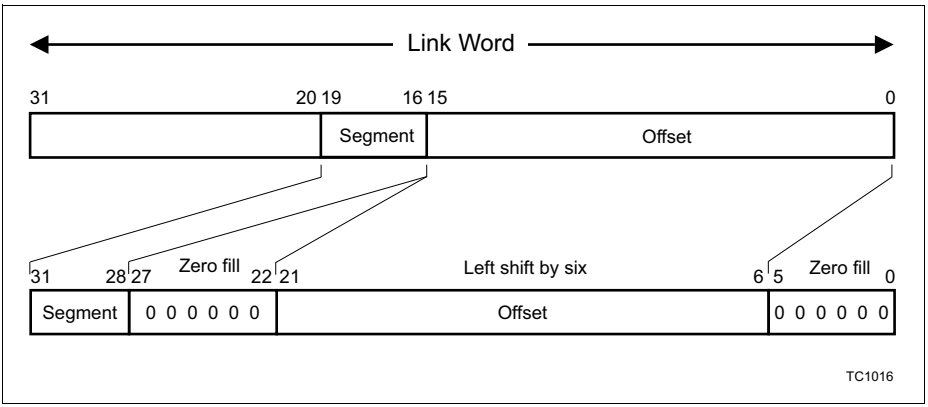


Figure 15 Generation of the Effective Address of a Context Save Area (CSA)

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, page 5-5**.

5.2 Task Switching Operation

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

There is an essential difference in the treatment of registers in the upper and lower contexts, in terms of how their contents are maintained. The lower context registers are similar to global registers in the sense that 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 8 Context Related Events and Instructions

Event / Instruction	Context Operation	Complement Instruction	Context Operation
Interrupt	Save Upper	RFE - Return from Exception	Restore Upper
Trap	Save Upper	RFE - Return from Exception	Restore Upper
CALL - Function Call	Save Upper	RET - Return from Call	Restore Upper
BISR - Begin Interrupt Service Routine	Save Lower	RSLCX - Restore Lower Context	Restore Lower
SVLCX - Save Lower Context	Save Lower	RSLCX - Restore Lower Context	Restore Lower

Table 8 Context Related Events and Instructions (Continued)

Event / Instruction	Context Operation	Complement Instruction	Context Operation
STLCX - Store Lower Context	Store Lower	LDLCX - Load Lower Context	Load Lower
STUCX - Store Upper Context	Store Upper	LDUCX - Load Upper Context	Load Upper

5.2.1 Save and Restore Context Operations

The Effective Address of all context related operations must be a physical memory address which maps to cached memory or data scratchpad RAM (of the processor performing the access). Using address ranges not covered by physical memories will lead to undefined results.

5.3 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. CSAs that contain saved upper or lower contexts are linked together in the previous context list. The following figure (**Figure 16**) shows a simple configuration of CSAs within both context lists.

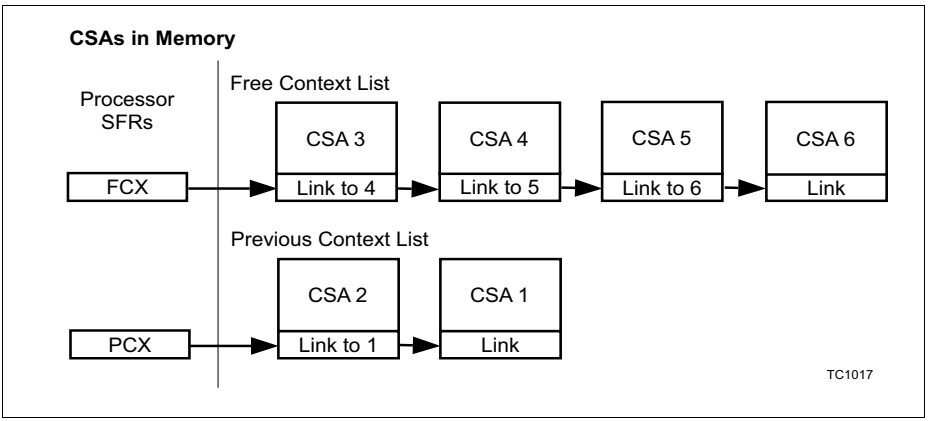


Figure 16 CSAs in Context Lists

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), page 7-10**.

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.

5.4 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. See **Interrupt Control Register (ICR), page 4-23**.

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.

5.5 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 can co-operate on the use of the lower context registers.

5.6 Context Save and Restore Examples

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

5.6.1 Context Save

Figure 17 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.

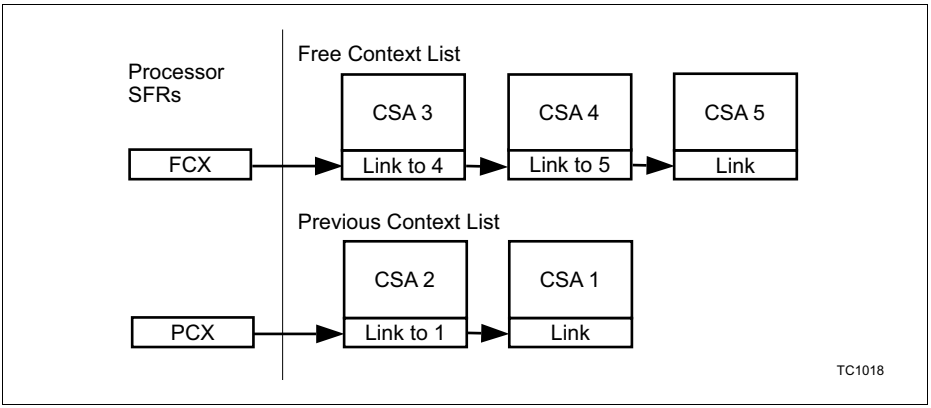


Figure 17 CSAs and Processor State Prior to Context Save

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

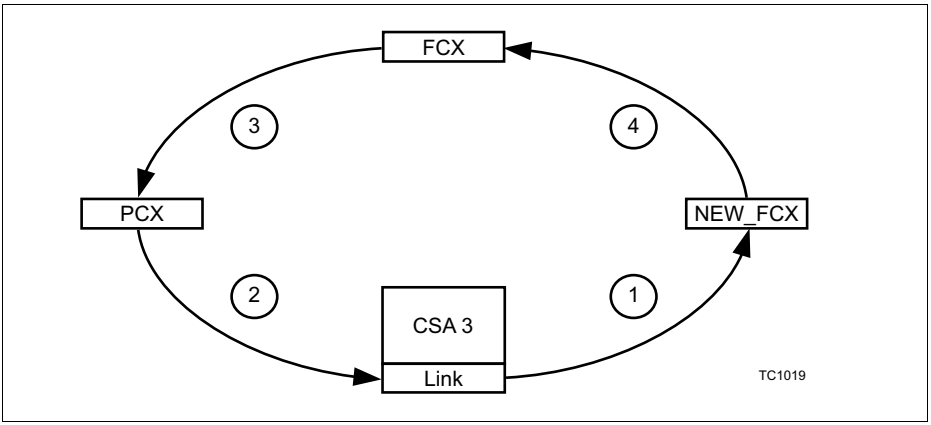


Figure 18 CSA and Processor SFR Updates on a Context Save Process

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

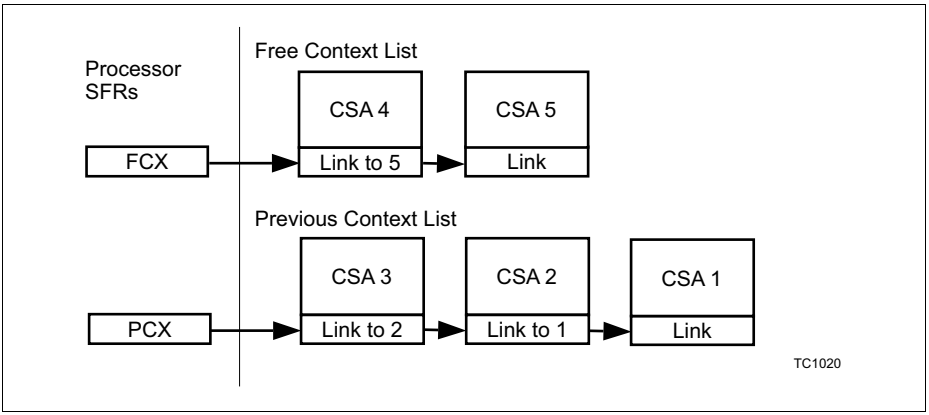


Figure 19 CSAs and Processor State After Context Save

5.6.2 Context Restore

The example in **Figure 20**, 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.

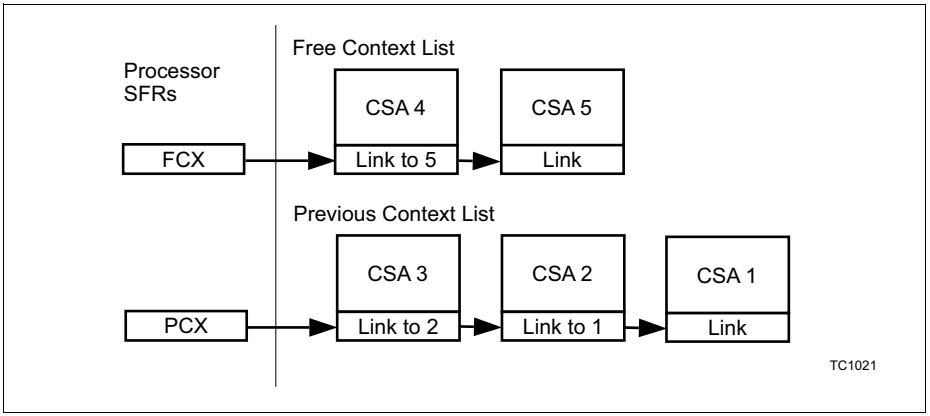


Figure 20 CSAs and Processor State Prior to Context Restore

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 21 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.

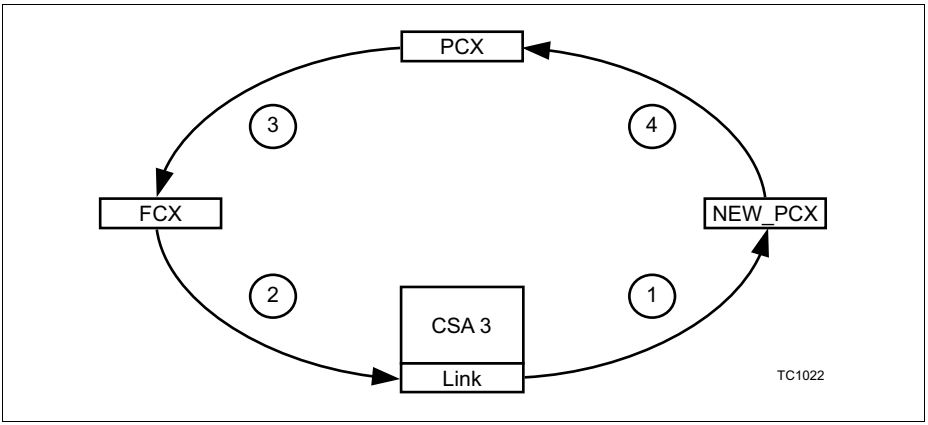


Figure 21 CSA and Processor SFR Updates on a Context Restore Process

The processor SFRs and CSAs now look as shown in **Figure 22**. The restored context is then written into the upper or lower context registers.

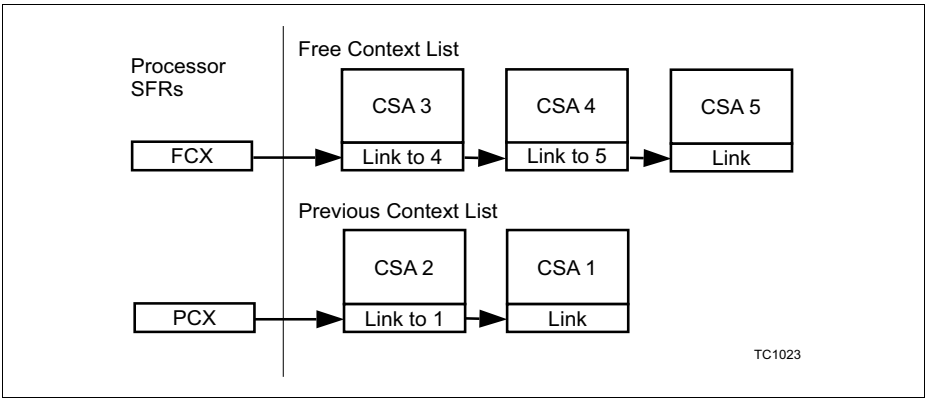


Figure 22 CSAs and Processor State After Context Restore

6 Interrupt System

This chapter describes the interrupt system, including arbitration, the priority level scheme, and access to the vector table.

In a TriCore™ system, multiple sources such as peripherals or external inputs can generate an interrupt signal to the CPU to request for service. The interrupt system also supports the implementation of additional units which are capable of handling interrupt requests, such as a second CPU, a standard DMA (Direct Memory Access) unit, or a PCP (Peripheral Control Processor). In the context of this chapter such units are known as ‘service providers’. Interrupt requests are often therefore referred to as ‘service requests’.

Besides the main CPU, up to three additional service providers can be handled with an interrupt Service Request Node (SRN). The actual number of additional service providers implemented in a given device is implementation dependent.

Each interrupt or service request from a module connects to a Service Request Node, containing a Service Request Control Register (SRC). Interrupt arbitration busses connect the SRNs with the interrupt control units of the service providers. These control units handle the interrupt arbitration and communication with the service provider.

Figure 23, page 6-2 shows an overview of a typical TriCore interrupt system.

6.1 Service Request Node (SRN)

Each Service Request Node contains a Service Request Control Register (SRC) and the necessary logic for communication with the requesting source and the interrupt arbitration busses. A peripheral or other module can have several service request lines, with each one of them connecting to its own individual SRN.

To support software-posting of interrupts for RTOS code, the TriCore architecture defines four Service Request Nodes (SRNs) which are not attached to a peripheral or any other module on the chip. The interrupt request bit can only be set by software. These SRNs are called the CPU Service Request Nodes. It should be noted however, that the interrupt request can also be set through an external bus master for example.

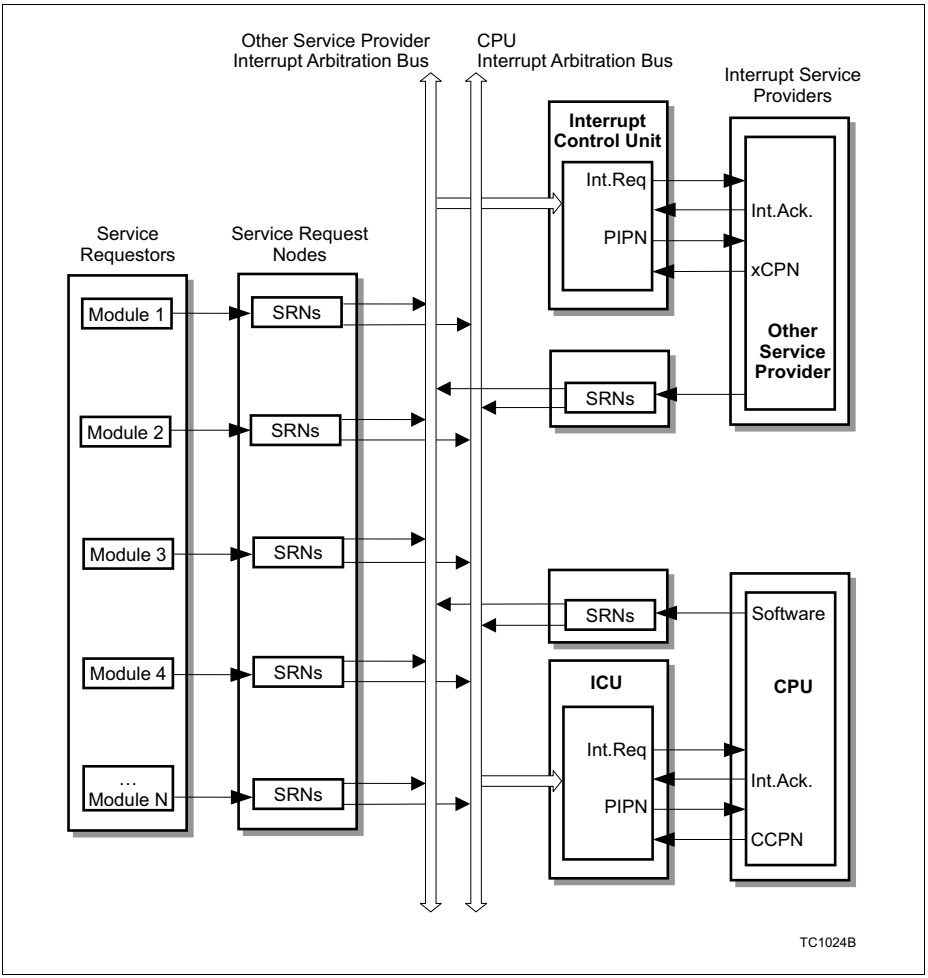


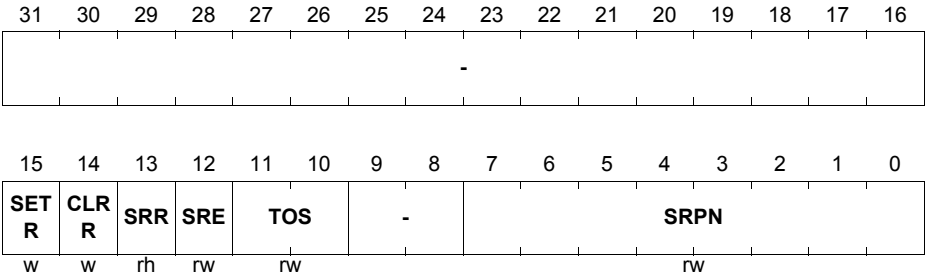
Figure 23 Block Diagram of a Typical TriCore Interrupt System

6.1.1 Service Request Control Register (SRC)

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. A request status bit shows whether or not the request is active. Besides being activated by the associated module through hardware, each request can also be set or reset through software.

The generic format and description of a Service Request Control register (SRC) is given below.

mod*_SRC*n
Service Request Control (*0 to 3*) **Reset Value: Implementation Specific**



Field	Bit	Type	Description
-	[31:16]	-	**Reserved Field**
SETR	15	w	**Service Request Set Bit** 0 : No action. 1 : Set SRR (no action if CLRR == 1). Written value is not stored. Read returns 0. No action if CLRR is also set. See **Service Request Set and Clear Bits (SETR, CLRR)** description.
CLRR	14	w	**Service Request Clear Bit** 0 : No action. 1 : Clear SRR (no action if SETR == 1). Written value is not stored. Read returns 0. No action if SETR is also set. See **Service Request Set and Clear Bits (SETR, CLRR)** description.

Field	Bit	Type	Description
SRR	13	rh	**Service Request Flag** 0 : No Service Request pending. 1 : Service Request is pending. See **Service Request Flag (SRR)** description.
SRE	12	rw	**Service Request Enable Control** 0 : Service Request is disabled. 1 : Service Request is enabled. See **Service Request Enable Control (SRE)** description.
TOS	[11:10]	rw	**Type-of-Service Control** 00_B : Service Provider 0. Typically CPU service is initiated. 01_B : Request Service Provider 1. Implementation specific. 10_B : Request Service Provider 2. Implementation specific. 11_B : Request Service Provider 3. Implementation specific. See **Type-of-Service Control (TOS)** description.
-	[9:8]	-	**Reserved Field**
SRPN	[7:0]	rw	**Service Request Priority Number** 00_H : A Service Request on this priority is never serviced. 01_H : Service Request, lowest priority. … FF_H : Service Request, highest priority. See **Service Request Priority Number (SRPN)** description.

Service Request Set and Clear Bits (SETR, CLRR)

These bits enable software to set or clear the actual service request bit SRR.

- Writing 1 to the SETR bit causes the SRR bit to be set to 1.
- Writing 1 to the CLRR bit causes the SRR bit to be cleared to 0.

If hardware attempts to modify SRR during an atomic read-modify-write software operation (such as store) the software operation succeeds and the hardware operation has no effect.

The value written to SETR or CLRR is not stored. Writing zero to these bits has no effect and these bits always return zero when read. If both SETR and CLRR are written to 1 at the same time, the SRR bit is not affected.

Service Request Flag (SRR)

The SRR bit is directly set or reset by the associated hardware. For example, an associated trigger event in a peripheral sets this bit to one and the acknowledgment of the service request by the Service Provider causes this bit to be cleared.

Bit SRR can be set or reset by software via bits SETR or CLRR, respectively. Writing directly to SRR via software has no effect.

SRR can be set or cleared (either by hardware or by software) regardless of the state of the enable bit SRE.

If SRE == 1, a pending service request takes part in the interrupt arbitration of the service provider selected via the TOS bit field. Bit SRR is automatically reset by hardware when the service request is acknowledged and serviced.

If SRE == 0, a pending service request is excluded from interrupt arbitrations. Software can poll SRR to check for a pending service request. SRR must be reset by software in this case (write 1 to CLRR).

Service Request Enable Control (SRE)

The SRE bit controls whether an active interrupt request is passed to the designated interrupt service provider (See the **Type-of-Service Control (TOS)** description, which follows). If SRE == 1, then the interrupt source associated with this SRN is enabled; i.e. if SRE is set to 1 and the value of the SRR bit moves to 1 (a service request is pending), the Service Request Node (SRN) will participate in interrupt arbitration rounds until the bit is cleared by software or until the interrupt is accepted for presentation to the interrupt service provider indicated by the TOS field. If the SRE bit is set to 0, then the associated interrupt source is disabled.

Disabling an interrupt source by clearing its SRE bit does not affect the setting or clearing of the SRR bit. The SRR bit can still be set by hardware or software (via the SETR bit), and can be read by software, but if the interrupt source is disabled it will not cause a hardware interrupt to be asserted. Users can therefore choose whether to handle the event associated with an individual SRN as an interrupt or through software polling.

Type-of-Service Control (TOS)

The interrupt system is designed to manage up to four Service Providers for service requests from peripherals or other sources. The TOS bit field is used to select the service provider for a request, indicating whether the service request takes part in the interrupt arbitration of the selected service provider. The number of service providers for a given device is implementation specific.

Service Request Priority Number (SRPN)

The 8-bit Service Request Priority Number (SRPN) of a service request, indicates its priority with respect to other sources requesting an interrupt to the same service provider, and to the priority of the service provider itself.

Each SRPN used by active sources requesting the same service provider must be unique at a given time. No active sources can use the same SRPN at the same time, except for the default SRPN of 00_H which excludes an SRN from taking part in the arbitration. This means that no two or more active sources (requesting CPU service for example) are allowed to use the same SRPN, although they can use the same SRPNs as sources which are requesting another service provider. The term active source in this context means a source which has its request enable bit SRE set to 1, to allow the request to participate in interrupt arbitrations. If a source is not active, meaning its service request enable bit is cleared (SRE == 0), no restrictions are applied to the Service Request Priority Number.

Implementations may look at a subrange of SRPN fields. In such an implementation or configuration the SRPN examined fields must be unique within the examined field.

The SRPN also identifies the entry into the interrupt vector table (or similar structures depending on the nature of the service provider). Unlike other interrupt systems the TriCore vector table provides an entry for each priority number, not for a specific interrupt source. In this way the vector table is de-coupled from the peripherals and a single peripheral can have multiple entry points for different purposes depending on its priority at a given time.

The range for the Service Request Numbers used in a system depends on the number of active service requests and the user-definable organization of the vector table. With the 8-bit SRPN, the interrupt arbitration scheme permits up to 255 sources to be active at one time. More information on the range of SRPNs can be found in **Interrupt Priority Groups, page 6-12**.

6.2 Interrupt Control Unit (ICU)

The Interrupt Control Unit (ICU) manages the interrupt system and arbitrates incoming interrupt requests to find the one with the highest priority and to determine whether or not to interrupt the service provider. The number of Interrupt Control Units depends on the number of service providers implemented in a TriCore device. Each ICU controls its associated interrupt arbitration bus and manages the communication with its service provider. The ICU is closely coupled with the CPU and its Interrupt Control Register (ICR). This register and the operation of the ICU is described in the sections which follow. In this document, only the CPU Interrupt Control Unit is detailed.

6.2.1 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 Pending Interrupt Priority Number (PIPN), as well as implementation-specific bits to control the interrupt arbitration cycles.

For a definition of the register, see **Interrupt Control Register (ICR), page 4-23**.

6.2.2 Interrupt Control Unit Operation

When an interrupt service is requested by one or more enabled sources, these requests are serviced depending on their priority ranking. The interrupt system must therefore determine which request has the highest priority each time multiple requests are received. The interrupt system uses a scheme that performs the arbitration in parallel to normal CPU operation. The Interrupt Control Unit (ICU) controls this scheme, which takes place in one or more cycles using the interrupt arbitration bus. The detailed arbitration scheme is implementation specific.

The ICU automatically starts an arbitration when a new interrupt request is detected. At the end of the arbitration the ICU has determined the service request with the highest priority number. This number is stored in the PIPN field of register ICR and generates an interrupt request to the CPU.

The CPU checks the state of the global interrupt enable bit ICR.IE, and compares the current CPU priority number CCPN in register ICR, 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; it reads the PIPN to determine the vector entry and acknowledges the ICU, which in turn sends acknowledgement back to the pending interrupt request (the ‘winner’ of this arbitration round), to inform it that it will be serviced. This node then resets its service request flag (SRR).

After sending the acknowledge, the ICU sets PIPN to 00_H (no valid pending request) and automatically starts a new arbitration to check whether there is another pending interrupt request. If there is then the priority number of this request is written to PIPN at the end of this arbitration. If there is no pending interrupt request then PIPN remains at 00_H and the ICU enters an idle state, waiting for the next interrupt request.

Note: Further CPU interrupt service actions are described in ***Entering an Interrupt Service Routine (ISR), page 6-8***.

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.

An arbitration is performed when a new service request is detected, regardless of whether the interrupt system is globally enabled or not, and regardless of whether there are other conditions preventing the CPU from servicing interrupts. In this way the PIPN field therefore reflects the pending service request with the highest priority. This can for example, be used for software polling techniques to determine high priority requests while keeping the interrupt system globally disabled.

If a new service request is generated by an SRN while an arbitration is in progress, this request has to wait until at least the end of that arbitration.

6.2.3 Arbitration Scheme

The arbitration scheme is implementation specific and is detailed in the documentation accompanying a specific TriCore product.

6.3 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, and A[11] (Return Address) 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.
- Memory protection using the interrupt memory protection map is enabled: PSW.PRS = 00_B.
- The Call Depth Counter (PSW.CDC) is cleared, and the call depth limit selector is set for 64: PSW.CDC = 1000000_B.
- 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. The effective address is the contents of the BIV register, ORd with the PIPN number left-shifted by 5.

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, page 6-12**.

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.

6.4 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.

6.5 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 entry points. The Interrupt Vector Table is stored in code memory.

When the CPU takes an interrupt, it calculates an address in the Interrupt Vector Table that corresponds with the priority of the interrupt (the ICR.PIPN bit field). This address is loaded 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. The PIPN is left-shifted by five bits 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 [12:5] of register BIV are set to 0. Note that bit 0 of the BIV register is always 0 and cannot be written to (instructions have to be aligned on even byte boundaries).

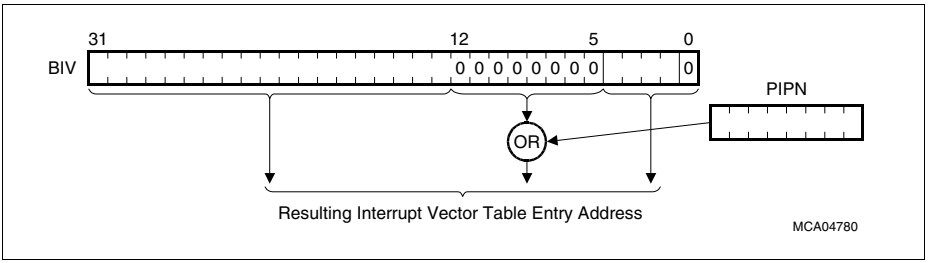


Figure 24 Interrupt Vector Table Entry Address Calculation

Left-shifting the PIPN by 5 bits creates entries in the vector table which are evenly spaced by 8 words. If an interrupt handler is very short it may fit entirely within the 8 words available in the vector code segment. Otherwise the code stored at the entry location can either span several vector entries[1)], or should contain some initial instructions followed by a jump to the rest of the handler.

Interrupt Vector Table, page 6-11 gives an overview of the interrupt vector table.

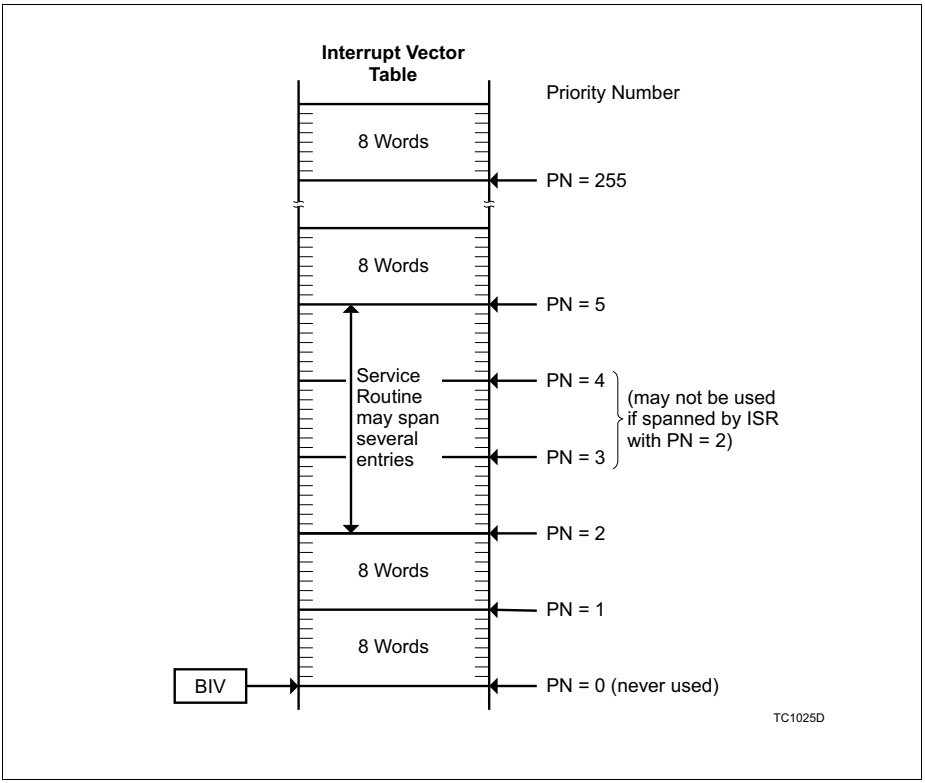


Figure 25 Interrupt Vector Table

The BIV register allows the interrupt vector table to be located anywhere in the available code memory. The default on power-up is fixed to 0000 0000_H, however 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.

1) See **Spanning Interrupt Service Routines across Vector Entries, page 6-12**.

6.6 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.

6.6.1 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 25, page 6-11**. 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 25, page 6-11**, 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.

6.6.2 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 26** shows an example.

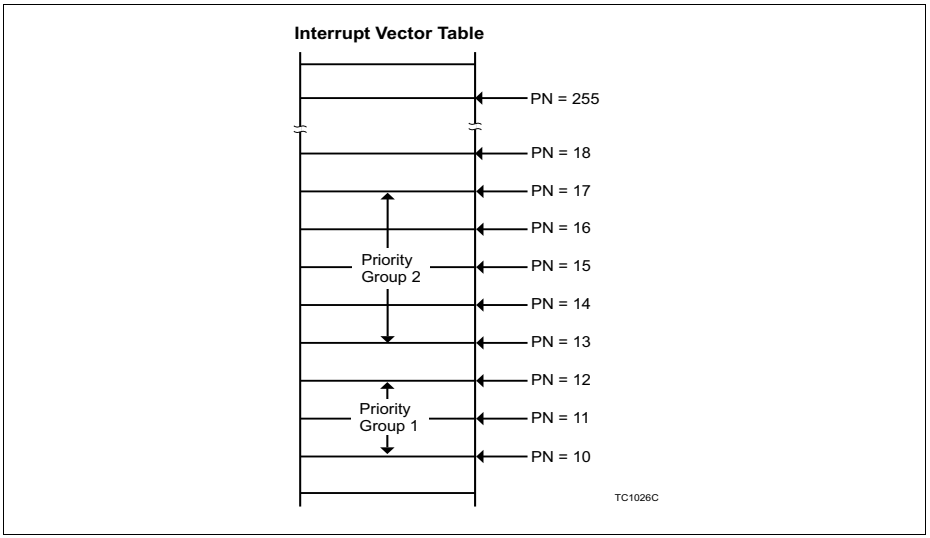


Figure 26 Interrupt Priority Groups

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.

6.6.3 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.

6.6.4 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.

6.6.5 Software-Posted Interrupts

A software-posted interrupt is a true hardware interrupt, carrying an interrupt priority that is processed through the regular interrupt subsystem when the interrupt is taken. The only difference is that the interrupt request is generated by explicitly setting the service request bit in a Service Request Node (SRN), through a software update of the node's control register.

Once the interrupt request bit in a service request control register is set, there is no way to distinguish between a software-posted interrupt request and a hardware interrupt request. For that reason it is generally advisable to use Service Request Nodes and interrupt priority numbers for software-posted interrupts that are not used for hardware interrupts, such as interrupts which are triggered by a peripheral module. However the number of hardware SRNs available in a given system for such purposes depends on the application requirements. An RTOS can not therefore rely on a certain number of 'free' SRNs for software-posting of interrupts.

To support the use of software-posted interrupts, principally for RTOS code, the architecture provides a number of Service Request Nodes which are intended solely for the purpose of software-posting. They are not connected to any peripheral or any other module on the chip, and the service request flag can only be set by software. This guarantees that there are SRNs available for the RTOS and user code which are not used by hardware modules.

Note: Current implementations contain four CPU Service Request Nodes.

6.6.6 Interrupt Priority Level One

Interrupt one is the first and lowest-priority entry in the interrupt vector and is best used for ISRs performing task management.

ISRs whose actions affect the launching of software-managed tasks post a software interrupt request at priority level one to signal the change. This posting is normally from RTOS code in a service function called directly from the ISR. The ISR can then execute a normal return from interrupt, rather than jumping to an ISR exit function in the kernel. There is no need for an exit function to check whether the ISR is returning to the background task level or to a lower priority ISR that it interrupted, in order to determine when to invoke the task dispatch function.

When there is a pending interrupt at a priority higher than the return context for the current interrupt, this interrupt will then be serviced. When a return to the background task level is performed the software-posted interrupt at priority level one will automatically be recognized and serviced.

7 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.

7.1 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 9 Supported Traps

TIN	Name	Synch. / Asynch.	HW / SW	Definition	Page
Class 0 — MMU					
0	VAF	Synch.	HW	Virtual Address Fill.	page 7-7
1	VAP	Synch.	HW	Virtual Address Protection.	page 7-7
Note: For VAF and VAP, see also **MMU Traps, page 10-5**.					page 10-5
Class 1 — Internal Protection Traps					
1	PRIV	Synch.	HW	Privileged Instruction.	page 7-7
2	MPR	Synch.	HW	Memory Protection Read.	page 7-7
3	MPW	Synch.	HW	Memory Protection Write.	page 7-8
4	MPX	Synch.	HW	Memory Protection Execution.	page 7-8
5	MPP	Synch.	HW	Memory Protection Peripheral Access.	page 7-8
6	MPN	Synch.	HW	Memory Protection Null Address.	page 7-8
7	GRWP	Synch.	HW	Global Register Write Protection.	page 7-8

Table 9 Supported Traps (Continued)

TIN	Name	Synch. / Asynch.	HW / SW	Definition	Page
Class 2 — Instruction Errors					
1	IOPC	Synch.	HW	Illegal Opcode.	page 7-8
2	UOPC	Synch.	HW	Unimplemented Opcode.	page 7-8
3	OPD	Synch.	HW	Invalid Operand specification.	page 7-9
4	ALN	Synch.	HW	Data Address Alignment.	page 7-9
5	MEM	Synch.	HW	Invalid Local Memory Address.	page 7-9
Class 3 — Context Management					
1	FCD	Synch.	HW	Free Context List Depletion (FCX = LCX).	page 7-10
2	CDO	Synch.	HW	Call Depth Overflow.	page 7-11
3	CDU	Synch.	HW	Call Depth Underflow.	page 7-11
4	FCU	Synch.	HW	Free Context List Underflow (FCX = 0).	page 7-11
5	CSU	Synch.	HW	Call Stack Underflow (PCX = 0).	page 7-11
6	CTYP	Synch.	HW	Context Type (PCXI.UL wrong).	page 7-11
7	NEST	Synch.	HW	Nesting Error: RFE with non-zero call depth.	page 7-12
Class 4 — System Bus and Peripheral Errors					
1	PSE	Synch.	HW	Program Fetch Synchronous Error.	page 7-12
2	DSE	Synch.	HW	Data Access Synchronous Error.	page 7-12
3	DAE	Asynch.	HW	Data Access Asynchronous Error.	page 7-12
Class 5— Assertion Traps					
1	OVF	Synch.	SW	Arithmetic Overflow.	page 7-13
2	SOVF	Synch.	SW	Sticky Arithmetic Overflow.	page 7-13
Class 6 — System Call[1)]					
	SYS	Synch.	SW	System Call.	page 7-13
Class 7 — Non-Maskable Interrupt					
0	NMI	Asynch.	HW	Non-Maskable Interrupt.	page 7-13

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.

7.1.1 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.

7.1.2 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.

7.1.3 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). See **MMU Traps, page 10-5** for more information.

7.1.4 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), page 7-13**.

7.1.5 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), page 7-11** for more information.

7.2 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, page 7-14**.

7.2.1 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.

7.2.2 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.

7.2.3 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), page 7-10**.

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.

7.2.4 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 27**). 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.

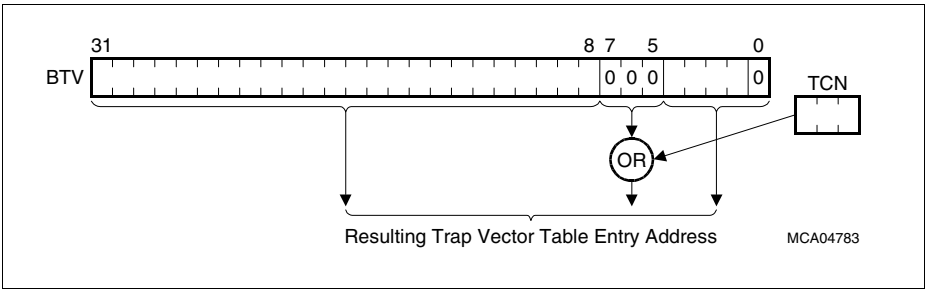


Figure 27 Trap Vector Table Entry Address Calculation

7.2.5 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 = 00_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.
- 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 = 00_B.
- 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.

7.3 Trap Descriptions

The following sub-sections describe the trap classes and specific traps listed in **Table 9 Supported Traps, page 7-1**.

7.3.1 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.

7.3.2 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 **Memory Protection System, page 9-1** for more details.

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

The following internal Protection Traps are defined:

PRIV - Privilege Violation (TIN 1)

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

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

MPR - Memory Protection Read (TIN 2)

The MPR trap is generated when the memory protection system is enabled and the effective address of a load, LDMST or SWAP 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 or SWAP instruction does not lie within any range with write permissions enabled.

This trap is not generated when an access violation occurs during a context save/restore operation.

MPX - Memory Protection Execute (TIN 4)

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

MPP - Memory Protection Peripheral Access (TIN 5)

A program executing in User-0 mode attempted a load or store access to a segment that has the privileged peripheral property. See **Physical Memory Attributes (PMA), page 8-1**.

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.

7.3.3 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, page 3-4**, for more information on these rules.

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 contraint 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 contraints 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 contraints 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*).

7.3.4 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 Registers (SYSCON), page 4-27**.

CDO - Call Depth Overflow (TIN 2)

A program attempted to make a call while executing with Call Depth Counting enabled, and the Call Depth Counter (CDC) was already at its maximum value. Call Depth Counting guards against context list depletion, by enabling the OS to detect ‘runaway recursion’ in executing tasks. See **Program Counter (PC), page 4-10**.

CDU - Call Depth Underflow (TIN 3)

A program attempted to execute a RET (Return) instruction while Call Depth Counting was enabled, and the Call Depth Counter was 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 operation, which presumably indicates a corrupted free list. The context save cannot be completed. Instead a forced jump is made to the FCU trap handler.

In failing to complete the context save, 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 or otherwise invalid. 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)

An RFE (Return From Exception) instruction was attempted when the Call Depth Counter was 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.

7.3.5 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), page 8-1**.
- A code fetch operation from Program scratchpad RAM (PSPR) (See **Scratchpad RAM, page 8-4**) where the access is beyond the end of the memory range.

DSE - Data Access Synchronous Error (TIN 2)

The DSE trap is raised when:

- A data access is attempted to a segment that does not have the data access property. (See **Physical Memory Attributes (PMA), page 8-1**).
- Whenever a bus error occurred because of a data load operation. It is also raised in the case of a data load operation from Data scratchpad RAM (DSPR) (**Scratchpad RAM, page 8-4**) where the access is beyond the end of the memory range.

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 trap is raised whenever a bus error occurred because of a data store operation, or when:

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.

- There is a data store operation to local scratch memory but the access is beyond the end of the memory range.
- There is an error caused by a cache management instruction.

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.

7.3.6 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).

7.3.7 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.

7.3.8 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.

7.3.9 Debug Traps

BBM - Break Before Make / BAM - Break After Make

Please refer to the Core Debug Controller chapter for information on debug traps. See **Chapter 11 Core Debug Controller (CDC), page 11-1**.

7.4 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 10 Synchronous Trap Priorities

Priority	Type of Trap
Instruction Fetch Traps	
1	Breakpoint (Virtual address, BBM)
2	VAF-P
3	VAP-P
4	MPX
5	PSE
Instruction Format Traps	
6	IOPC
7	OPD
8	UOPC
Instruction Traps	
9	Breakpoint trap (Instruction, BBM)
10	PRIV
11	GRWP
12	SYS
Context Traps	
13	FCD
14	FCU (Synchronous)

Table 10 Synchronous Trap Priorities (Continued)

15	CSU
16	CDO
17	CDU
18	NEST
19	CTYP
Data Memory Access Traps	
20	MEM (Data address)
21	ALN
22	MPN
23	VAF-D
24	VAP-D
25	MPR
26	MPW
27	MPP
28	DSE
General Data Traps	
29	SOVF
30	OVF
31	Breakpoint trap (BAM)

Table 11 Asynchronous Trap Priorities

Priority	Asynchronous Traps
1	NMI
2	DAE[1)]

1) DAE is used for both load and store errors.

8 Physical Memory Attributes (PMA)

This chapter describes the Physical Memory Attributes (PMA) that regions of the TriCore™ physical address map may or may not have. These attributes are defined by groups of physical memory properties.

8.1 Physical Memory Properties (PMP)

The TriCore architecture defines properties which physical memory addresses may or may not possess. These properties are:

- Privileged Peripheral (P).
- Cacheable (C).
- Speculative (S).
- Code Fetch (F).
- Data Access (D).

Each property defines a characteristic of the accesses that are possible to a physical memory region. For example, an address that does not have the cacheable property C, would be described as Non-cacheable $\overline{C}$.

In the following definitions the concept of necessary and speculative accesses is introduced. 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.

Privileged Peripheral (P)

Only Supervisor and User-1 mode data accesses are possible. No User-0 mode data access is possible. User-0 mode data accesses result in an MPP (Memory Protection Peripheral access) trap. All accesses are exempt from the protection system settings. PTE translation where the physical address targets a region with this property results in undefined behaviour.

Cacheable (C)

It is possible for data and code fetch accesses to the region to be cached by the CPU if a data cache or code cache is respectively present and enabled.

Speculative (S)

It is possible to perform speculative data accesses to the memory. A speculative data access is a read access to memory addresses that are not strictly necessary for correct program execution. The processor never performs speculative write accesses which are visible in a memory region.

Code Fetch (F)

Fetch accesses are possible to this region. The fetch property allows full speculation on all fetch accesses to the region. The cacheable property has no affect on the amount or range of speculation of code fetches. If a necessary fetch access is directed by program flow to a physical memory region that does not have the fetch property then a PSE (Program fetch Synchronous Error) trap occurs.

Data Access (D)

Data accesses are possible to this region. If a data access is directed by necessary program flow to a physical memory region that does not have the Data Access property, then a DSE (Data access Synchronous Error) trap occurs.

For data accesses, the interpretation of the combinations of the Privileged Peripheral, Cacheable and Speculative properties for a memory region are defined in **Table 12**. All other combinations of these three properties not present in this table, are reserved.

Table 12 Data Access - Cacheable and Speculative Properties

Name	Privileged Peripheral Property	Cacheable Property	Speculative Property	Behaviour of Physical Memory Region
Precise data access	P or $\overline{P}$	$\overline{C}$	$\overline{S}$	The processor only performs necessary accesses, in order, to the region.
Non-Cached access	$\overline{P}$	$\overline{C}$	S	The processor may read an entire cache line[1] containing the address of a necessary access and place it in a buffer for subsequent accesses. The order of accesses is not guaranteed [2].
Full Speculation	$\overline{P}$	C	S	The processor may perform speculative read accesses to entire cache lines in physical memory and place them in the cache. The order of accesses is not guaranteed.

1) The size of a cache line is implementation dependant. Examples of implemented cache lines are 16-bytes and 32-bytes, but may be smaller or larger.

2) The order of non-cached data accesses can be guaranteed by inserting a DSYNC instruction (ref to DSYNC) after each load or store instruction.

8.2 Physical Memory Attributes (PMA)

A physical memory attribute is a defined set of physical memory properties. The architecture defines four attributes:

Peripheral Space = $\overline{PCSFD}$.

Emulator Space = $\overline{PCSFD}$.

Cacheable Memory = $\overline{P}CSFD$.

Non-Cacheable Memory = $\overline{PCSFD}$.

All accesses to physical memory that has the Emulator Space attribute are directly translated (See **Memory Management Unit (MMU), page 10-1**) and are not subject to the protection constraints imposed by the protection system (See **Memory Protection System, page 9-1**); i.e. It is not possible to generate an MPX, MPR or MPW trap with a memory access to Emulator Space.

8.2.1 Physical Memory Attributes of the Address Map

The TriCore 4 GBytes (32-bit) of physical address space is divided into 16 equally sized segments. Each segment has its own physical memory attribute.

Segment F_H is constrained to be Peripheral Space and the lower 15 segments have implementation defined physical memory attributes, although Segment D_H is constrained to be either Cacheable or Non-Cacheable Memory. The standard implementation attributes are shown in the following table:

Table 13 TriCore Default Physical Memory Attributes for all Segments

Segment	Attributes
F_H [1)]	Peripheral Space.
E_H	Peripheral Space.
D_H [2)]	Non-cacheable Memory.
C_H	Cacheable Memory.
B_H	Non-cacheable Memory.
A_H	Non-cacheable Memory.
9_H	Cacheable Memory.
8_H	Cacheable Memory.
7_H - 0_H	Cacheable Memory.

1) F_H is constrained to be Peripheral Space.

2) See **Data Scratchpad Register (DSPR), page 6-2**.

The Emulator Space attribute is assigned to an implementation defined region of memory when the Debug Mode is enabled.

All physical memory accesses are subject to the constraints imposed by the PMA attributes before being permitted to execute.

8.3 Scratchpad RAM

Segment D contains the scratchpad RAM. There are two different scratchpad RAMs:

- DSPR - Data scratchpad RAM.
- PSPR - Program scratchpad RAM.

Table 14 Scratchpad RAM

Segment D Regions	Properties
$DFFFFFFF_H$ – $D8000000_H$	Implementation Dependent
$D7FFFFFF_H$ – $D4000000_H$	PSPR
$D3FFFFFF_H$ – $D0000000_H$	DSPR

8.4 Permitted versus Valid Accesses

A memory access can be permitted without necessarily being valid. There are three sources of permission for a memory access, the Protection System (PS), the Memory Management Unit (MMU) and the Physical Memory Attributes (PMA). If a memory access is permitted by the MMU or PS, then it must also be permitted by the PMA for the access to proceed, as shown in **Figure 28**. A memory access is not valid if the address of the access is to an unimplemented region of memory or is misaligned; therefore an access can be permitted but not valid.

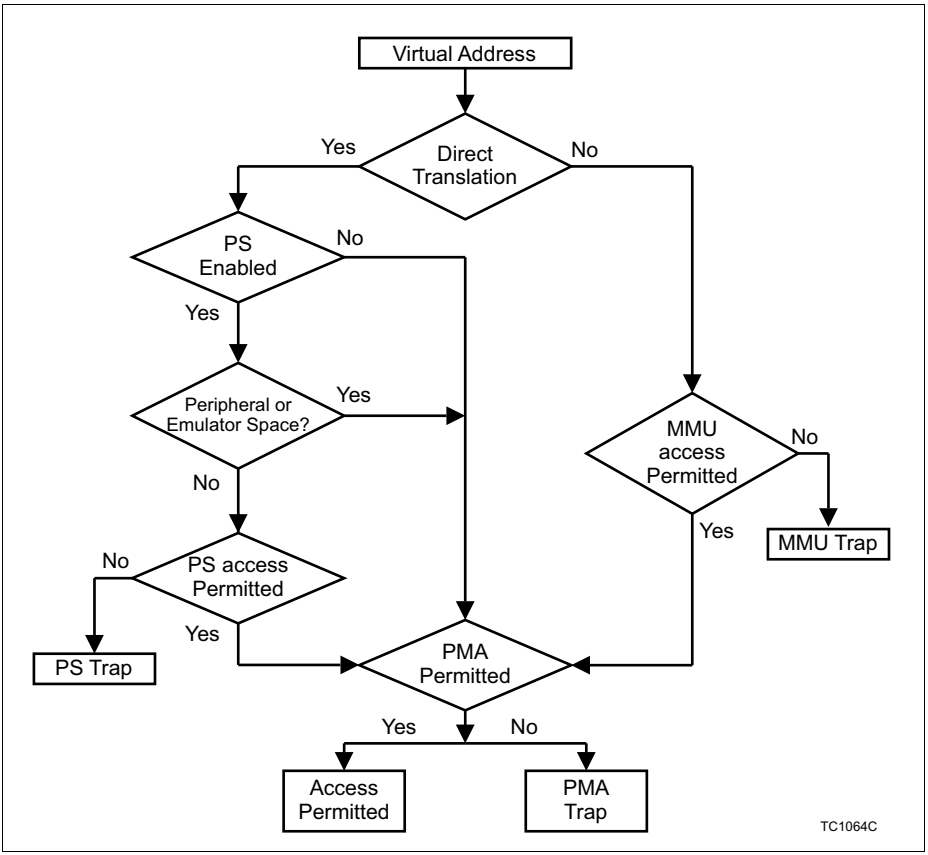


Figure 28 Translation Paths

The PS and MMU act upon the direct translation and virtual translation paths respectively, therefore the permission for a memory access that undergoes virtual translation lies only with the MMU, not the PS, and vice-versa.

9 Memory Protection System

The TriCore™ protection system provides the essential features needed to isolate errors and facilitate debugging. 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 in addition, be used to generate signals to the Core Debug Controller (CDC) (See **Core Debug Controller (CDC), page 11-1**).

This chapter describes the hardware operation of the protection system and introduces the use of the protection features by software in real-time systems. The protection system differentiates between peripheral protection and memory protection. The majority of this chapter is concerned with memory protection, which does not apply to memory regions that have the peripheral space or emulator space attribute. See the chapter **Physical Memory Attributes (PMA), page 8-1**.

Effective addresses are translated into physical addresses using one of two translation mechanisms:

- Direct translation.
- Page Table Entry (PTE) based translation.

These translation mechanisms are both defined in the chapter **Memory Management Unit (MMU), page 10-1**.

Memory protection for addresses that undergo direct address translation is enforced using the range-based memory protection system described in this chapter.

9.1 Memory Protection Registers

TriCore contains register sets that specify the address range and the access permissions for a number of memory ranges. The PSW.PRS field is used to select which register set is active at a given time. Two register sets are selected simultaneously:

- One Data Memory Protection.
- One Code Memory Protection.

The PSW.PRS field allows selection of up to four such register sets; four for data and four for code. See **Program Status Word (PSW), page 4-11** for more details on the PSW.

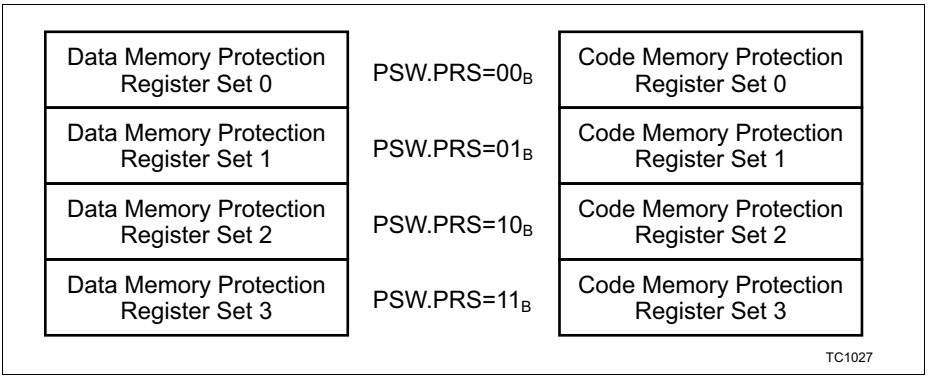


Figure 29 Memory Protection Register Sets

The number of register sets provided for memory protection is specific to each TriCore implementation, limited to a minimum of two and a maximum of four. This document only describes the generic format of these register sets. Unimplemented register set addresses are reserved and undefined.

Each register set is made up of several range registers (also called Range Table Entries). The number of range registers in a Data or Code Memory Protection Set is implementation defined. Each Range Table Entry (**Figure 30, page 9-3**) 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.

Lower Bound <= address < Upper Bound.

The Mode register contains the access permission and debug control bits. The control options are different for the data and the code memory protection.

For load and store operations, data address values are checked against the entries in the data range table.

On instruction fetches, the PC value for the fetch is checked against the entries in the code range table. When an address is found to fall outside of all ranges defined in the appropriate range table, then permission for the access is denied. When an address is found to fall within a range defined in the appropriate range table, the associated mode table entry is checked for access permissions. An instruction fetch cannot occur from a byte aligned address, and so the least significant bit of the Code Segment Protection upper and lower bound registers (CPRx_*n*) is not writeable and always returns zero.

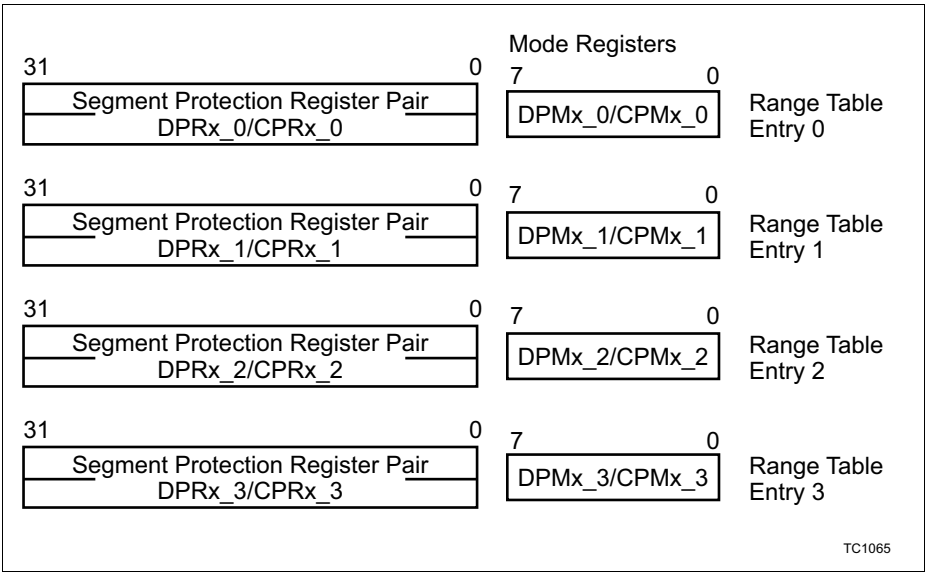
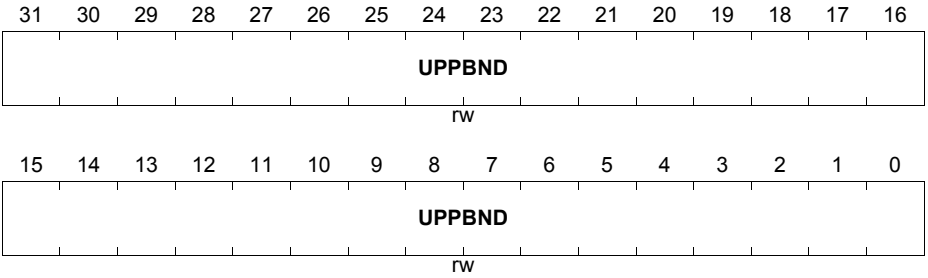


Figure 30 Example of Four Range Table Entries in a Protection Register Set

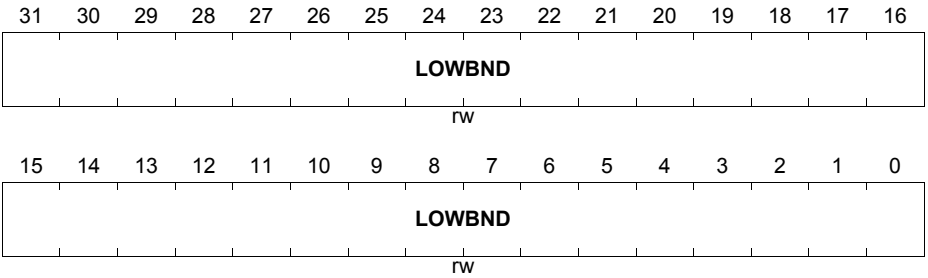
9.1.1 Memory Protection Registers

DPR*x*_*n*U
Data Segment Protection Register *x*_*n* Upper Bound
(x = set number 0 to 3; n = range number) Reset Value: Implementation Specific



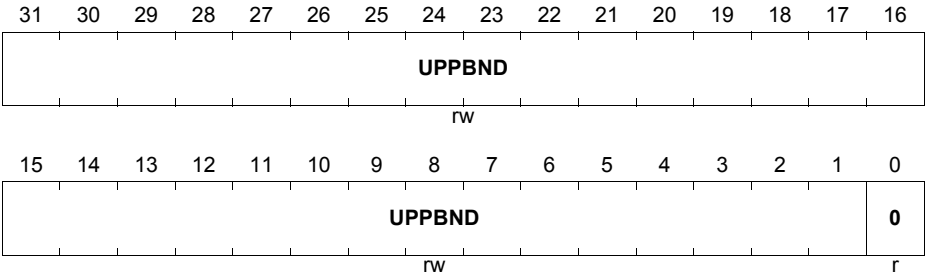
Field	Bits	Type	Description
UPPBND	[31:0]	rw	**DPR*x*_*n* Upper Boundary Address**

DPR*x*_*n*L
Data Segment Protection Register *x*_*n* Lower Bound
(x = set number 0 to 3; n = range number) Reset Value: Implementation Specific



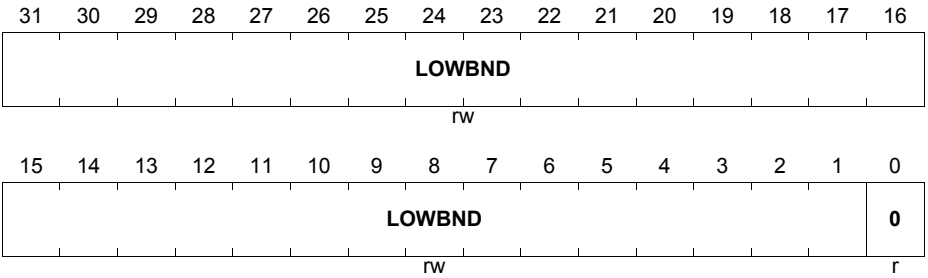
Field	Bits	Type	Description
LOWBND	[31:0]	rw	**DPR*x*_*n* Lower Boundary Address**

CPR*x*_*n*U
Code Segment Protection Register *x*_*n* Upper Bound
(x = set number 0 to 3; n = range number) Reset Value: Implementation Specific



Field	Bits	Type	Description
UPPBND	[31:0]	rw	**CPR*x*_*n* Upper Boundary Address** The least significant bit is not writeable and always returns zero.

CPR*x*_*n*L
Code Segment Protection Register *x*_*n* Lower Bound
(x = set number 0 to 3; n = range number) Reset Value: Implementation Specific



Field	Bits	Type	Description
LOWBND	[31:0]	rw	**CPR*x*_*n* Lower Boundary Address** The least significant bit is not writeable and always returns zero.

DPM*x*
Data Protection Mode Register *x*
(*x* = set number 0 to 3) **Reset Value: Implementation Specific**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
WE3	**RE3**	**WS3**	**RS3**	**WB L3**	**RB L3**	**WB U3**	**RB U3**	**WE2**	**RE2**	**WS2**	**RS2**	**WB L2**	**RB L2**	**WB U2**	**RB U2**
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
WE1	**RE1**	**WS1**	**RS1**	**WB L1**	**RB L1**	**WB U1**	**RB U1**	**WE0**	**RE0**	**WS0**	**RS0**	**WB L0**	**RB L0**	**WB U0**	**RB U0**
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Field	Bits	Type	Description
WE(3-0)	31, 23, 15, 7	rw	**Address Field Write Enable** 0 : Data write accesses to associated address range not permitted. 1 : Data write accesses to associated address range permitted.
RE(3-0)	30, 22, 14, 6	rw	**Address Field Read Enable** 0 : Data read accesses to associated address range not permitted. 1 : Data read accesses to associated address range permitted.
WS(3-0)	29, 21, 13, 5	rw	**Address Range Data Write Signal** 0 : Data write signal disabled. 1 : Signal asserted to Core Debug Controller (CDC) on data write accesses to associated address range.
RS(3-0)	28, 20, 12, 4	rw	**Address Range Data Read Signal** 0 : Data read signal disabled. 1 : Signal asserted to Core Debug Controller (CDC) on data read accesses to associated address range.
WBL(3-0)	27, 19, 11, 3	rw	**Data Write Signal on Lower Bound Access** 0 : Data write signal disabled. 1 : Signal asserted to Core Debug Controller (CDC) on data write access to an address that matches lower bound address of associated address range.

Field	Bits	Type	Description
RBL(3-0)	26, 18, 10, 2	rw	**Data Read Signal on Lower Bound Access** 0 : Data read signal disabled. 1 : Signal asserted to Core Debug Controller (CDC) on data read access to an address that matches lower bound address of associated address range.
WBU(3-0)	25, 17, 9, 1	rw	**Data Write Signal on Upper Bound Access** 0 : Write signal disabled. 1 : Signal asserted to Core Debug Controller (CDC) on data write access to an address that matches upper bound address of associated address range.
RBU(3-0)	24, 16, 8, 0	rw	**Data Read Signal on Upper Bound Access** 0 : Data read signal disabled. 1 : Signal asserted to Core Debug Controller (CDC) on data read access to an address that matches upper bound address of associated address range.

CPM*x*
Code Protection Mode Register *x*
(*x* = set number 0 to 3) **Reset Value: Implementation Specific**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
XE3	**-**	**XS3**	**-**	**BL3**	**-**		**BU3**	**XE2**	**-**	**XS2**	**-**	**BL2**	**-**		**BU2**
rw		rw		rw			rw	rw		rw		rw			rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
XE1	**-**	**XS1**	**-**	**BL1**	**-**		**BU1**	**XE0**	**-**	**XS0**	**-**	**BL0**	**-**		**BU0**
rw		rw		rw			rw	rw		rw		rw			rw

Field	Bits	Type	Description
XE(3-0)	31, 23, 15, 7	rw	**Address Range Execute Enable** 0 : Instruction fetch accesses to associated address range not permitted. 1 : Instruction fetch accesses to associated address range permitted.
-	30, 28, [26:25], 22, 20, [18:17], 14, 12, [10:9], 6, 4, [2:1]	-	**Reserved Field**
XS(3-0)	29, 21, 13, 5	rw	**Address Range Execute Signal** 0 : Execute signal disabled. 1 : Signal asserted to Core Debug Controller (CDC) on instruction fetch accesses to associated address range.
BL(3-0)	27, 19, 11, 3	rw	**Execute Signal on Lower Bound Access** 0 : Lower bound execute signal disabled. 1 : Signal asserted to Core Debug Controller (CDC) on instruction fetch access to an address that matches lower bound address of associated address range.

Field	Bits	Type	Description
BU(3-0)	24, 16, 8, 0	rw	**Execute Signal on Upper Bound Access** 0 : Upper bound execute signal disabled. 1 : Signal asserted to Core Debug Controller (CDC) on instruction fetch access to an address that matches upper bound address of associated address range.

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 ISR. The PRS field in the PSW determines the current protection register set number.

Modes of Use for Range Table Entries

Individual range table entries can be used just for memory protection or for debugging. One entry is rarely used for both purposes. If the upper and lower bound values have been set for debug breakpoints they are probably not meaningful for defining protection ranges, and vice-versa. However, it is both possible and reasonable to have some entries used for memory protection and others used for debugging.

To disable an entry for use in memory protection, clear both the RE and WE bits in a data range table entry or clear the XE bit in a code range table entry. The entry can be disabled for use in debugging by clearing any debug signal bits.

When a range entry is being used for debugging, the debug signal bits that are set determine whether it is used as a single range comparator (giving an in-range/not in-range signal) or as a pair of equal comparators. The two uses are not mutually exclusive.

Using Protection Register Sets

Supervisor mode does not automatically disable memory protection. The protection register set that is selected for Supervisor mode tasks 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. But the Supervisor mode does not implicitly override memory protection, and it is possible for a Supervisor mode task to take a memory protection trap.

9.2 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, if range A is set for read/write permission and range B read-only, the intersecting region of A and B will be read/write. Nesting of ranges can be used for example to allow read/write access to a subrange of a larger range in which the current task is allowed read access.

9.2.1 Example

Figure 31 illustrates the Data Protection Register Set *x*, where *x* is one of the four sets as selected by the PSW.PRS field. The register set in this example consists of four range table entries.

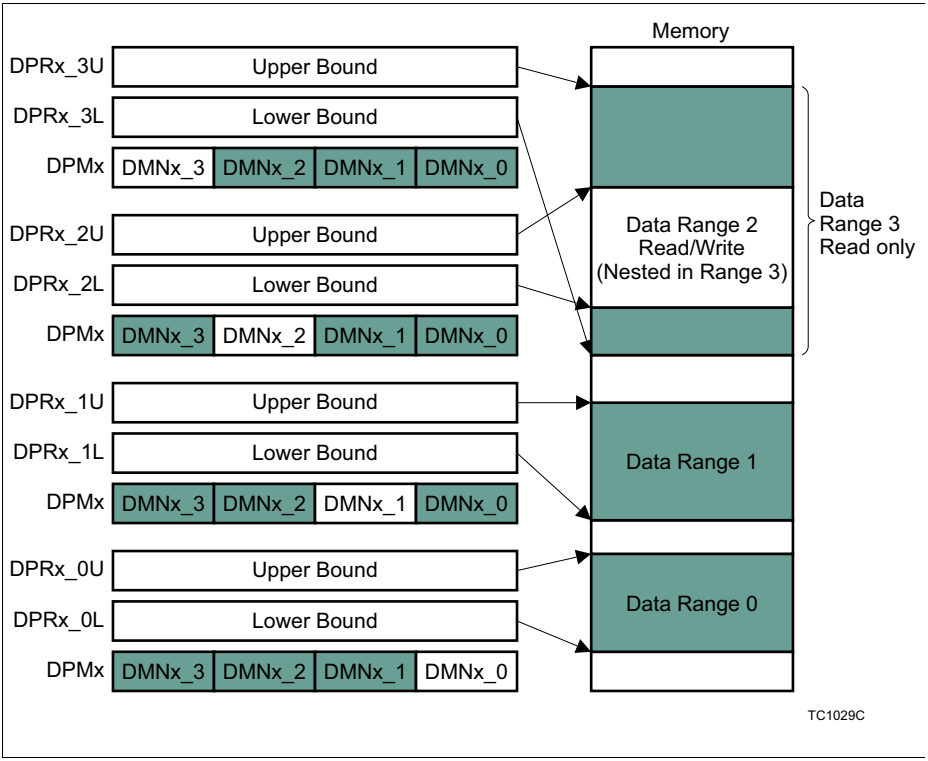


Figure 31 Example Configuration of a Data Protection Register Set

In **Figure 31** the four Data Segment and four Data Protection Mode Registers are set up as follows:

- Data Segment Protection Registers DPR*x*_3U and DPR*x*_3L, define the upper (U) and lower (L) bound for Data Range 3. Data Protection Mode Register 3 (DPM*x*_3) defines the read-only permissions for Data Range 3.
- DPR*x*_2U and DPR*x*_2L define the upper (U) and lower (L) bound for Data Range 2. DPM*x*_2 defines the read-write permissions for Data Range 2.

Note: Data Range 2, which has read/write permission, is nested within Data Range 3, which has read-only permission. Because the intersecting rules state that permission to access a memory location is the OR of the regions permissions, this region therefore has read/write permission.

- DPR*x*_1U and DPR*x*_1L define the upper (U) and lower (L) bound for Data Range 1. DPM*x*_1 defines the permissions for Data Range 1.
- DPR*x*_0U and DPR*x*_0L define the upper (U) and lower (L) bound for Data Range 0. DPM*x*_0 defines the permissions for Data Range 0.

This same configuration can be used to illustrate Code Protection Register Set *x*.

9.3 Using the 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 defined in the protection register set.

9.3.1 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.

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

9.3.3 Address Range

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

The protection system does not apply to accesses in memory regions with the peripheral space or emulator space attribute. If a memory access is attempted to either of these segments, the access is permitted by the protection system (but not necessarily the Physical Memory Attributes) regardless of the protection system settings. For more on PMA, see **Physical Memory Attributes (PMA), page 8-1**.

Saves or restores of contexts to the context save area (see **Save and Restore Context Operations, page 5-5**) do not require the permission of the protection system to proceed.

9.3.4 Traps

There are three traps generated by memory protection, 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 **Chapter 7 Trap System, page 7-1** for a complete description of Traps.

9.4 Crossing Protection Boundaries

A memory access can straddle two regions defined by the protection system. **Figure 32** 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 as to whether or not a memory protection trap is taken.

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.

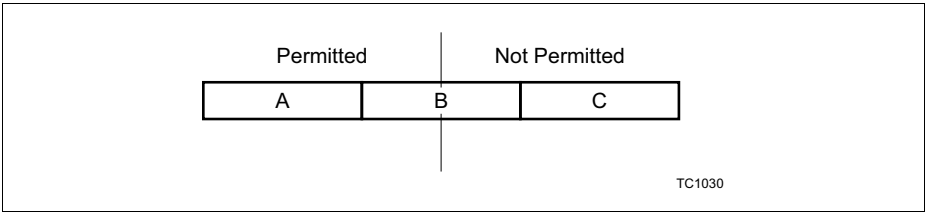


Figure 32 Protection Boundaries

10 Memory Management Unit (MMU)

This chapter describes the TriCore™ Memory Management Unit (MMU) architecture. The MMU is an optional component in TriCore configurations. It need not be present in every system that uses the core, and even when present it can be disabled.

If the MMU is not present and enabled in a system, then virtual memory and page-based memory access protection are not supported for that system. The range-based protection system (a non-optional core component) still provides basic memory protection services. For more details see **Memory Protection System, page 9-1**.

The memory management features include:

- 4 GBytes of virtual address space divided into 16 segments of 256 MBytes each. The upper half of the virtual address space (segments [8_H - F_H]) is global, and mapped directly onto the physical address space. The lower segment (segments [0_H - 7_H]) is implicitly qualified by an Address Space Identifier (ASI). The operating system can allocate distinct address spaces to each unique ASI. Two or more processes can also share an address space ID, either serially or in parallel.
- 4 GBytes of physical address space divided into 16, 256 MByte segments.
- Virtual to physical address translation by direct translation or via Page Table Entries (PTE), depending on the segment number of the virtual address and the status of the MMU.
- Cacheability and access permissions based on physical memory attributes for directly translated addresses, or by a combination of physical memory attributes and virtual page attributes for addresses translated via Page Table Entries. Attributes for segments [8_H - F_H] are pre-defined in a system memory map.

Virtual addresses are always translated into physical addresses before accessing memory. The virtual address is translated into a physical address using either direct translation or Page Table Entry (PTE) translation.

- Direct translation: If the virtual address belongs to the upper segment of the virtual address space then the virtual address is directly used as the physical address. If the virtual address belongs to the lower segment of the address space, then the virtual address is used directly as the physical address if the processor is operating in Physical mode (i.e. the MMU is disabled or not present).
- PTE translation: If the virtual address belongs to the lower segment of the address space and the processor is operating in Virtual mode (i.e. the MMU is present and enabled), then the virtual address is translated using a Page Table Entry.

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.

Six memory-mapped Memory Management Unit (MMU) Core Special Function Registers (CSFRs) control the memory management system.

10.1 Address Spaces

The TriCore virtual address space is 4 GBytes in size and divided into 16 segments, with each segment consisting of 256 MBytes. The upper 4 bits of the 32-bit virtual address are used to identify the segment. Segments are numbered [0_H - F_H].

Note: A virtual address is always translated into a physical address before accessing memory.

The physical address space is 4 GBytes in size and is divided into 16 segments of 256 MBytes each.

The physical and virtual address space maps are shown in the following figure:

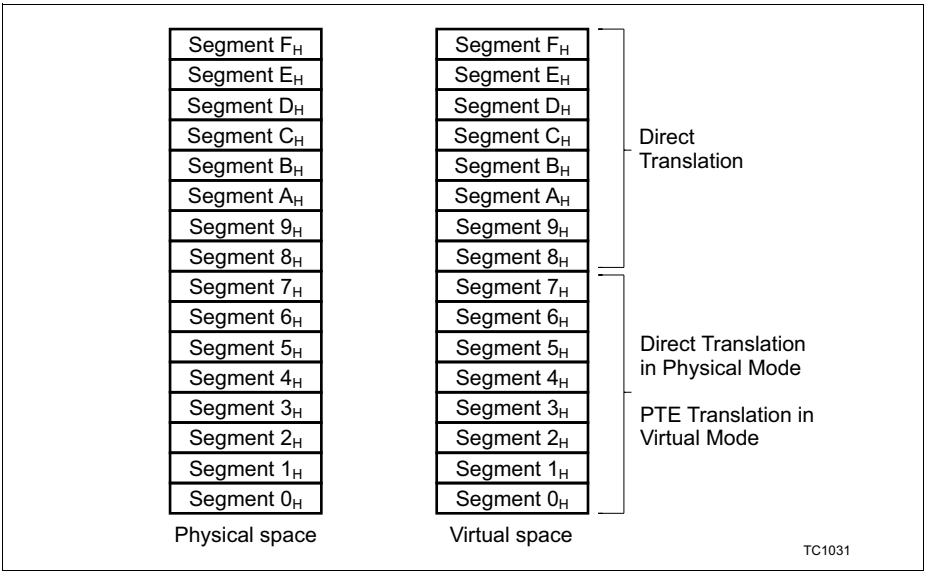


Figure 33 Physical and Virtual Address Spaces

A 32-bit virtual address is comprised of a Virtual Page Number (VPN) concatenated with a Page Offset.

A 32-bit physical address is comprised of a Physical Page Number (PPN) concatenated with a Page Offset.

10.2 Address Translation

The MMU_CON.V bit controls the operating mode, Physical or Virtual, of the processor; Physical mode if MMU_CON.V == 0, or Virtual mode if MMU_CON.V == 1 (See **MMU Configuration Register (MMU_CON), page 10-13**).

The virtual address is translated into a physical address using either direct translation or Page Table Entry translation, as shown in **Figure 34**.

Translation using the Page Table Entry (PTE) is performed by replacing the Virtual Page Number (VPN) of the virtual address by a Physical Page Number (PPN), to obtain a physical address.

If the virtual address is from the upper segments of the virtual address space then the virtual address is used directly as the physical address (direct translation).

If the virtual address is from the lower segments of the address space, then the virtual address is used directly as the physical address if the processor is operating in Physical mode, or translated using a Page Table Entry if the processor is operating in Virtual mode.

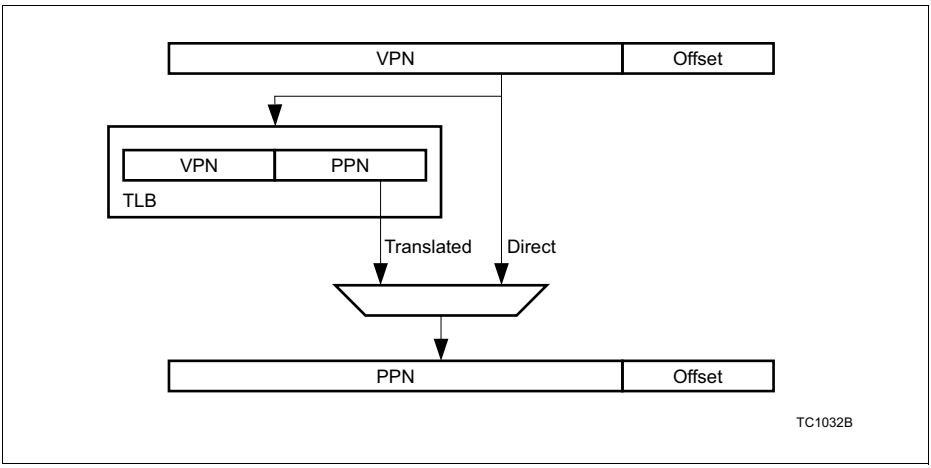


Figure 34 Virtual Address Translation

10.2.1 Address Translation for CSFR Pointers

The context pointers (PCX, FCX, and LCX), the Base Trap Vector (BTV) and the Base Interrupt Vector (BIV) are constrained to use segments that undergo direct translation. See **Context Management Registers, page 4-17** and **Interrupt and Trap Control, page 4-23**.

10.3 Translation Lookaside Buffers (TLBs)

The MMU provides PTE-based virtual address translation through two Translation Lookaside Buffers (TLBs); TLB-A and TLB-B.

The MMU supports four page sizes; 1 KByte, 4 KBytes, 16 KBytes, and 64 KBytes, although not all of these sizes can be used at once. At any given time each TLB provides translations for only one particular page size. The page size setting of each TLB is determined through the MMU_CON.SZA and MMU_CON.SZB fields, as described in **MMU Configuration Register (MMU_CON), page 10-13**.

Each TLB contains a number (*N)* of TLB Table Entries (TTEs), where *N* is a minimum of 4 and a maximum of 128. The MMU_CON.TSZ field determines the size of each TLB.

Each TTE has an 8-bit index associated with it:

- Index numbers 0, …, MMU_CON.TSZ, are used for the entries in TLB-A.
- Index numbers 128, …, 128+MMU_CON.TSZ, are used for the entries in TLB-B.

Each TTE contains a Page Table Entry (PTE).

The organization (associativity) of each TLB is implementation-dependent.

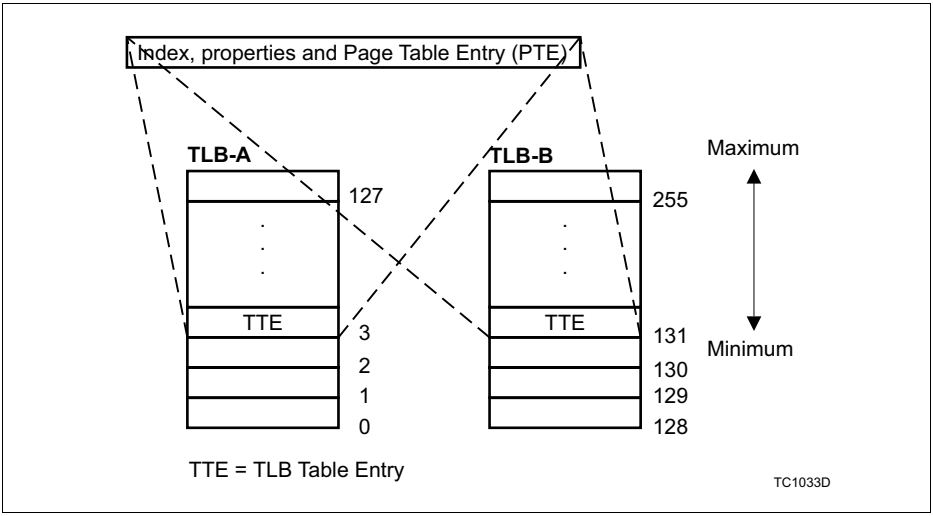


Figure 35 Translation Look-Aside Buffers

10.3.1 TLB Table Entry (TTE) Contents

TLB Table Entries (TTE) contain the following fields:

- **Address Space Identifier (ASI)**: Specifies the address space corresponding to the virtual address. ASIs allow mappings of up to 32 virtual address spaces to coexist in a TLB. An ASI is similar to a Process ID.
- **Virtual Page Number (VPN)**: Stores 32 – $\log_2$ *Pagesize* bits where *Pagesize* is the size of the page in bytes.
- **Physical Page Number (PPN)**: Stores 32 – $\log_2$ *Pagesize* bits where *Pagesize* is the size of the page in bytes.
- **Execute Enable (XE)**: Allows instruction fetches from the virtual page.
- **Write Enable (WE)**: Allows data writes to the virtual page.
- **Read Enable (RE)**: Allows data reads from the virtual page.
- **Cacheability bit (C)**: Allows the virtual page to be cached.
- **Global bit (G)**: Indicates that the page is globally mapped therefore making it visible in all address spaces.
- **Valid bit (V)**: Indicates that the TTE contains a valid mapping.

10.4 Multiple Address Spaces

The MMU provides efficient support for multiple and shared virtual address spaces. Each TTE (TLB Table Entry) contains an Address Space Identifier (ASI) which can identify the distinct address space corresponding to the particular mapping. The ASI Register (MMU_ASI) provides the current address space identifier for all memory accesses.

Virtual address translation is performed by a valid TTE if:

- It is a non-global TTE that matches the incoming VPN of the virtual address and the Address Space Identifier contained in the ASI register.
- It is a global TTE that matches the incoming VPN.

Note: Global TTEs are indicated by the TTE[ ].G bit. Such mappings are visible to all virtual address spaces.

10.5 MMU Traps

MMU traps belong to Trap Class Number (TCN) 0. The MMU can generate the following traps:

- VAF (Virtual Address Fill).
- VAP (Virtual Address Protection).

The VAF trap is generated if PTE-based translation is required for a virtual address and there is no PTE translation in the MMU. The VAP trap is generated if there is a matching PTE and the access is disallowed. The VAP trap can also occur when user-0 accesses upper segments in virtual mode.

The VAF trap is assigned a TIN (Trap Identification Number) of 0 while the VAP trap is assigned a TIN of 1. Both the VAF and VAP traps are synchronous traps.

The events that happen on an MMU trap are the same as those that happen on any other trap. In addition to context saving and control transfer, the virtual address is right shifted by 10 + 2 * min(MMU_CON.SZA, MMU_CON.SZB), and loaded into the Translation Fault Page Address register (MMU_TFA). **Figure 36** shows Virtual mode traps for read, write and fetch accesses.

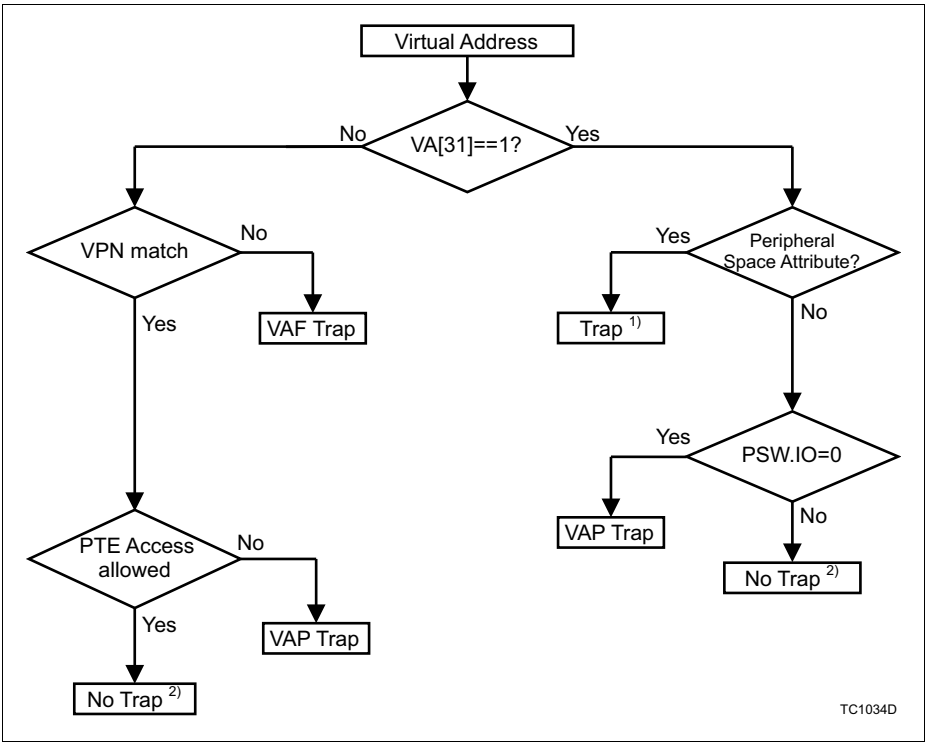


Figure 36 Virtual Mode Traps for Read, Write and Fetch Accesses[1) 2)]

Any User-0 mode access to a virtual address in the upper segments that does not have the peripheral space attribute results in a VAP trap. See **Trap Types, page 7-1** for more on traps.

1) In User-0 mode the MPP trap is for read/write accesses, and the PSE trap for fetch accesses. In User-1 and Supervisor modes, the PSE trap is for fetch accesses. There is no trap for read/write accesses in these modes.

2) Subject to constraints imposed by the Physical Memory Attributes. See **Physical Memory Attributes (PMA), page 8-1**.

10.6 Virtual Mode Protection

Memory protection is enforced using separate mechanisms for the two translation paths. These are described in this section.

10.6.1 Direct Translation

Virtual memory protection for addresses that undergo direct translation is provided using standard TriCore range-based protection. The range-based protection mechanism provides support for protecting memory ranges from unauthorized read, write or instruction fetch accesses. Refer to the chapter **Memory Protection System, page 9-1**.

10.6.2 Page Table Entry (PTE) Based Translation

Virtual memory protection for addresses that undergo PTE-based translation is provided by properties of the PTE used for the address translation. The PTE provides support for protecting a virtual address from unauthorized read, write or instruction fetches by other processes. The following PTE bits are provided for this purpose:

- Execute Enable (XE) - allows instruction fetch from the virtual page.
- Write Enable (WE) - allows data writes to the virtual page.
- Read Enable (RE) - allows data reads from the virtual page.

For each of these bits; 1 = allows and 0 = disallows.

See **Translation Physical Address Register (MMU_TPA), page 10-17**.

10.7 Cacheability

A memory access is cacheable if both the virtual address is allowed to be cached and the physical memory attributes allow the access to be cached. The physical memory attributes (PMA) are described in **Physical Memory Attributes (PMA), page 8-1.**

10.7.1 Direct Translation Virtual Address Cacheability

For all virtual addresses undergoing Direct Translation the virtual address is cacheable.

10.7.2 PTE Translation Cacheability

For all virtual addresses undergoing PTE (Page Table Entry) Translation, the virtual address is cacheable if the PTE entry C bit is set and not cacheable if the PTE entry C bit is reset.

10.7.3 Cacheability of a Virtual Address Flow

The following figure describes the determination for cacheability of a memory access.

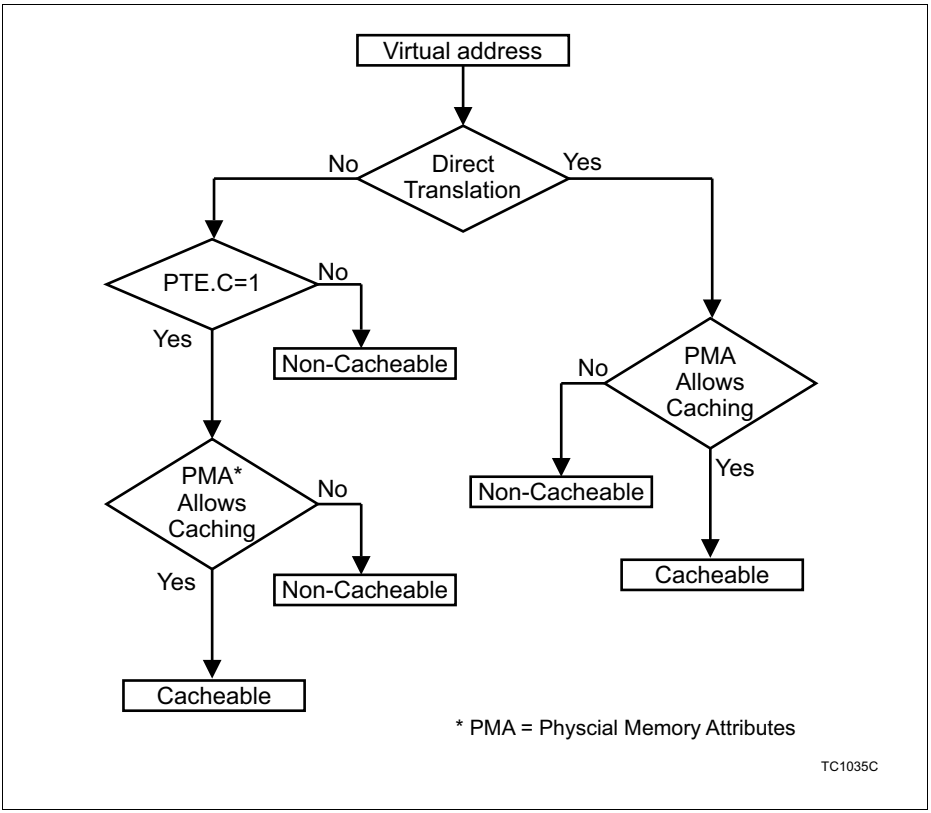


Figure 37 Cacheability of a Virtual Address

10.8 MMU Instructions

All MMU instructions are privileged instructions that require Supervisor mode for execution. If the MMU is physically present (MMU_CON.NOMMU == 0) the instructions execute normally, whether or not the MMU is enabled (MMU_CON.V == 0 or 1). If the MMU is not present (MMU_CON.NOMMU == 1), then all MMU instructions cause an un-implemented opcode (UOPC) trap.

10.8.1 TLBMAP (TLB Map)

The TLBMAP instruction is used to install a mapping in the MMU. The TLBMAP instruction takes an extended data register E[a] as a parameter. The data register D[a] contains the virtual address for the translation and the data register D[a+1] contains the page attributes and PPN (Physical Page Number). The ASI (Address Space Identifier) for the translation is obtained from the MMU_ASI register. The page attributes are contained in the most significant byte of the odd register with the format as shown:

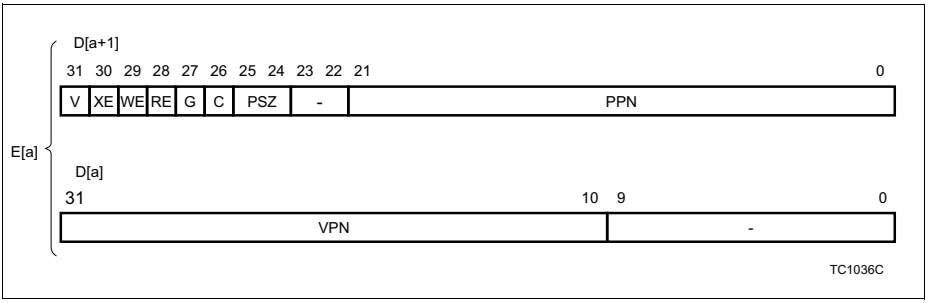


Figure 38 TLBMAP E[a] Format

Entering a mapping with any of the following properties results in undefined behaviour:

- A mapping with a virtual page that wholly or partly overlaps an existing virtual page mapping. A virtual page mapping will overlap with another virtual page mapping if the mappings have an equal or enclosing VPN and the same ASI or at least one of the mappings has its global bit (G) set.
- A mapping for a page size which is not one of the two valid page sizes.
- A mapping using a physical address in a memory region that has the peripheral space attribute.
- A mapping where the VPN is in the upper half of the address space.
- A mapping where the V bit is set to 0.

Undefined behaviour in the context of a TLBMAP instruction means that the MMU TLBs contain undefined PTEs (Page Table Entries). To restore defined behaviour both TLBs have to be flushed (see **TLBFLUSH (TLB Flush), page 10-10**), and the valid PTEs re-entered.

Entering a mapping when the two TLBs have identical page size settings results in the mapping being entered in one of the two TLBs, with the choice being implementation dependent.

Bits D[a][9:0] and D[a+1][23:22] are reserved and are 0. Bits D[a][15:10] and D[a+1][5:0] are reserved when unused, and therefore are 0 when unused by the page size. For example, if the page size (PSZ) is set to 4 KBytes (01_B) then bits D[a][11:10] of the VPN are unused and must be set to 0. Similarly, bits D[a+1][1:0] of the PPN are also unused and must be set to 0.

Note that a TLBMAP instruction must be followed by an ISYNC instruction before attempting to use a new installed mapping.

10.8.2 TLBDEMAP (TLB Demap)

The TLBDEMAP instruction is used to uninstall a single mapping in the MMU. TLBDEMAP takes as a parameter a data register that contains the virtual address whose mapping is to be removed. The address space identifier for the demap operation is obtained from the Address Space Identifier (ASI) register (MMU_ASI). Demapping a translation that is not present in the MMU is legal and results in a NOP.

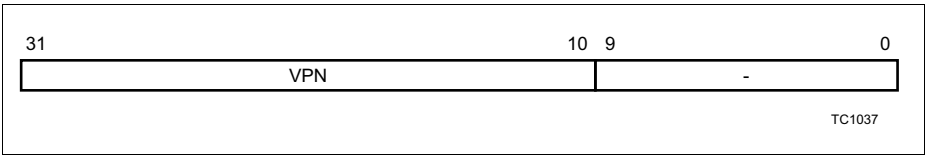


Figure 39 TLBDEMAP D[a] Format

Note: A TLBDEMAP instruction should be followed by an ISYNC before any access to an address in the demapped page is made.

10.8.3 TLBFLUSH (TLB Flush)

The TLBFLUSH instructions are used to flush mappings from the MMU. There are two variants of the TLBFLUSH instruction:

- TLBFLUSH.A flushes all the mappings from TLB-A.
- TLBFLUSH.B flushes all the mappings from TLB-B.

Note: A TLBFLUSH instruction should by followed by an ISYNC before any access to an address requiring PTE translation is made.

10.8.4 TLBPROBE (TLB Probe)

The TLBPROBE instructions are TLBPROBE.A and TLBPROBE.I.

TLBPROBE.A (TLB Probe Address)

This instruction takes a data register D[a] as a parameter and is used to probe the MMU for a virtual address. The D[a] register contains the virtual address for the probe. The address space identifier for the probe is obtained from the ASI (Address Space Identifier) register.

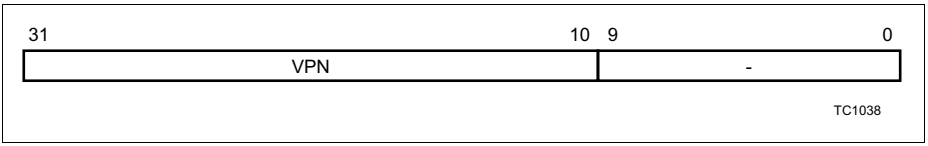


Figure 40 TLBPROBE.A

TLBPROBE.I (TLB Probe Index)

This instruction takes a data register D[a] as a parameter and is used to probe the TLB at a given TLB index. The D[a] register contains the index for the probe. The index for the TLBs is implementation specific and there is no architecturally defined way to predict what TLB index value will be associated with a given address mapping. Bits D[a][31:8] are reserved and must be set to 0's.

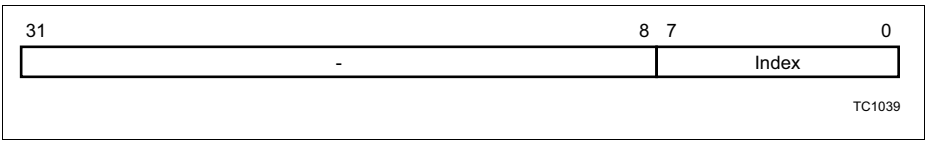


Figure 41 TLBPROBE.I

The TLBPROBE instructions return the following:

- The ASI and VPN (Virtual Page Number) of the translation in the Translation Virtual Address register (TVA).
- The PPN (Physical Page Number) and attributes in the Translation Physical Address register (TPA).
- The TLB index of the translation in the Translation Page Index register (TPX).

The MMU_TPA.V bit is set to zero if the TTE contained an invalid translation or an invalid index was used for the probe. All of the other bitfields are undefined.

10.9 TLB Usage

The TriCore architecture does not specify any PTE replacement algorithm. Entry of a valid new mapping using the TLBMAP instruction does not guarantee the continued existence of a previously entered mapping even if the TLB has not been filled; i.e. the replacement algorithm may over-write a previous mapping. An implementation will always provide a means to ensure forward progress of instructions requiring multiple mappings to execute. Use of the MMU will therefore involve a software architecture with the same fundamental mechanisms as shown in **Figure 42**.

When executing TLBDEMAP, TLBFLUSH.A or the TLBFLUSH.B instruction, an implementation may require additional operations to be performed in order to maintain coherency of the processors view of memory. For example, removing a mapping that has the cacheability bit set using the TLBDEMAP instruction may require the data and program caches to be flushed before a new mapping can be entered that uses an overlapping physical page with the cacheability bit clear.

An implementation may also impose additional restrictions on the PTEs that can be mapped in order to maintain coherency of the processors view of memory. For example, an implementation may require that the least significant *n* bits of cacheable PTEs, VPN and PPN, be identical to avoid aliasing in a virtually indexed cache.

Context saves and restores are always directly translated irrespective of the segment the context save area resides in.

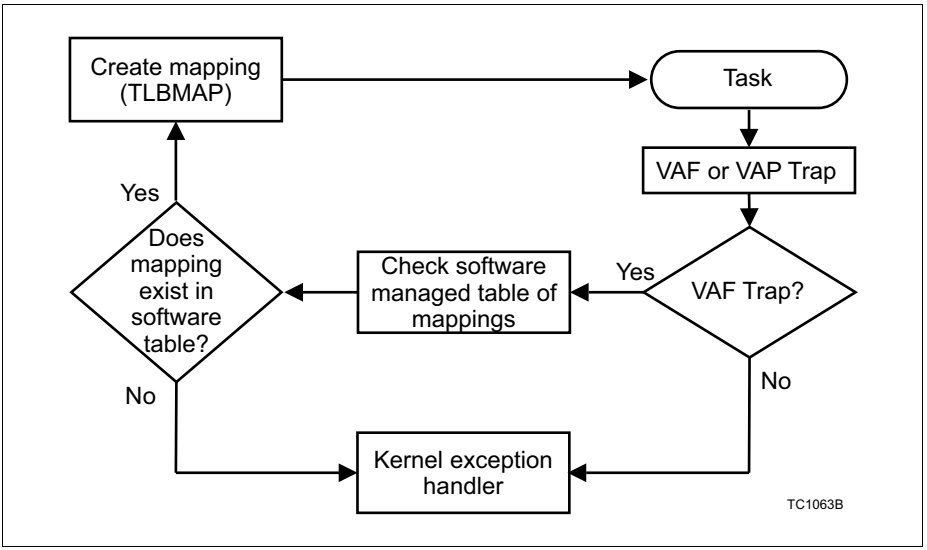


Figure 42 Using TLB (Translation Lookaside Buffer)

10.10 MMU Core Special Function Registers

All MMU Core Special Function Registers can be read using the MFCR instruction. The MMU_CON and MMU_ASI registers are the only software-writeable registers. They are both written using the MTCR instruction. The MTCR instruction must be followed by an ISYNC instruction before any PTEs are used.

Note: If MMU_CON.NOMMU == 1 (MMU not present) then all other registers in the section do not exist and are undefined. If they are accessed no error occurs, but the read and write results are undefined.

Note: If no MMU is present (MMU_CON.NOMMU == 1), then MMU instructions will cause a UOPC (Unimplemented Opcode) trap.

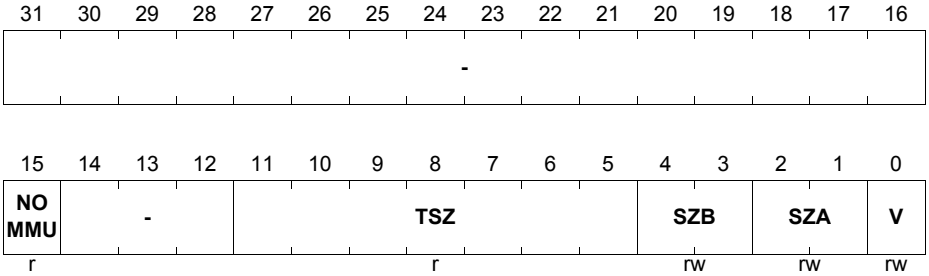
All MMU registers other than the MMU_CON register have undefined values at reset.

The MMU_CON register is set to $0yyyyyyy00000_B$ at reset where yyyyyyy is implementation dependent.

10.10.1 MMU Configuration Register (MMU_CON)

A MTCR instruction that changes the SZA (Page Size A) bit field must be followed by a TLBFLUSH.A instruction to ensure that TLB A remains coherent with the programmers view. Similarly a MTCR instruction that changes the SZB (Page Size B) bit field must be followed by a TLBFLUSH.B instruction. MTCR instructions that change the MMU_CON.V bit must only be executed from memory addresses undergoing direct translation or, in the case of disabling the MMU, be executed from a virtual address that translates to the exact same physical address, otherwise the MMU behaviour is undefined.

MMU_CON
Configuration Register **Reset Value: Implementation Specific**



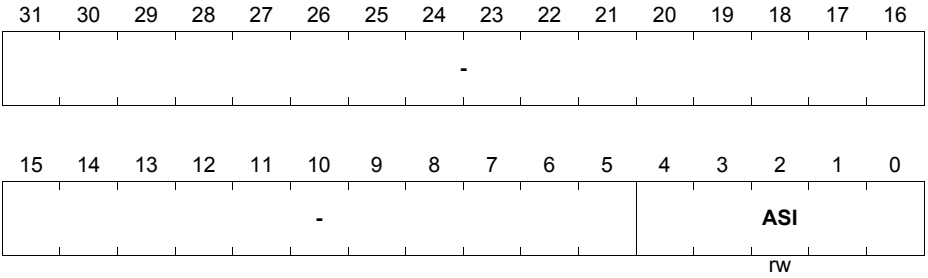
Field	Bits	Type	Description
-	[31:16]	-	**Reserved Field**

Field	Bits	Type	Description
NOMMU	15	r	**No MMU Available** 0 : MMU is present. 1 : MMU is not present (all other bits in MMU_CON undefined). Note that to enable the MMU when present (MMU_CON.NOMMU == 0), the MMU_CON.V bit must be set.
-	[14:12]	-	**Reserved Field**
TSZ	[11:5]	r	**TLB Size** Determines the size of each TLB. The entries of TLB-A are indexed 0 through TSZ while the entries of TLB-B are indexed 128 through 128+TSZ. Each TLB has a maximum of TSZ+1 entries.
SZB	[4:3]	rw	**Page Size B** Page size of the mappings in TLB-B. 00_B : 1 KByte. 01_B : 4 KByte. 10_B : 16 KByte. 11_B : 64 KByte.
SZA	[2:1]	rw	**Page Size A** Page size of the mappings in TLB-A. 00_B : 1 KByte. 01_B : 4 KByte. 10_B : 16 KByte. 11_B : 64 KByte.
V	0	rw	**Virtual mode** 0 : Physical mode. 1 : Virtual mode. This bit enables the MMU when the MMU is present (MMU_CON.NOMMU == 0).

10.10.2 Address Space Identifier Register (MMU_ASI)

The Memory Management Unit (MMU), Address Space Identifier (ASI) register description.

MMU_ASI
Address Space Identifier Register **Reset Value: Implementation Specific**

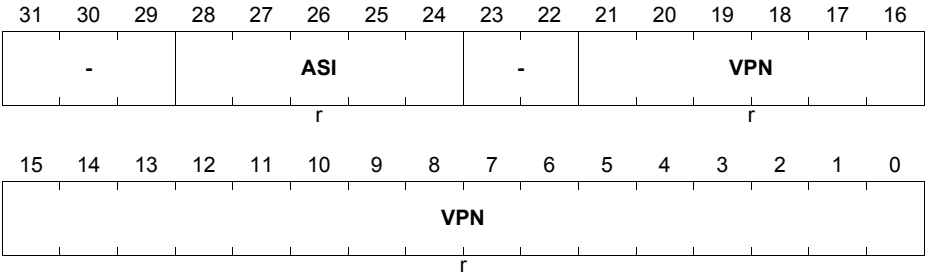


Field	Bits	Type	Description
-	[31:5]	-	**Reserved Field**
ASI	[4:0]	rw	**Address Space Identifier** The ASI register contains the Address Space Identifier of the current process.

10.10.3 Translation Virtual Address Register (MMU_TVA)

The MMU_TVA register is used to return the ASI and VPN (Virtual Page Number) result of a TLBPROBE instruction.

MMU_TVA
Translation Virtual Address Register **Reset Value: Implementation Specific**

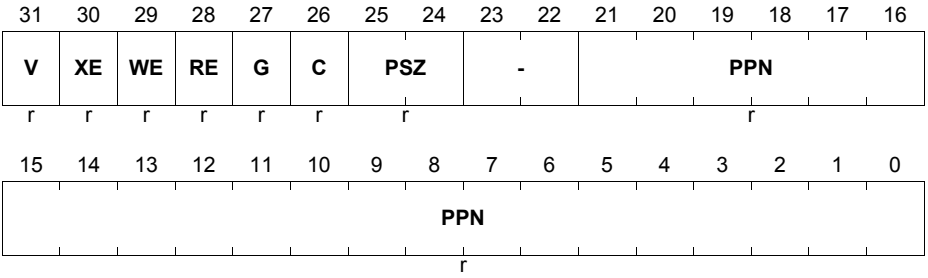


Field	Bits	Type	Description
-	[31:29]	-	**Reserved Field**
ASI	[28:24]	r	**Address Space Identifier** The ASI field contains the Address Space Identifier of the PTE.
-	[23:22]	-	**Reserved Field**
VPN	[21:0]	r	**Virtual Page Number** The VPN of the PTE accessed by the last TLBPROBE instruction.

10.10.4 Translation Physical Address Register (MMU_TPA)

The MMU_TPA register is used to return the PPN (Physical Page Number) and attributes of a translation by a TLBPROBE instruction.

MMU_TPA
Translation Physical Address Register **Reset Value: Implementation Specific**



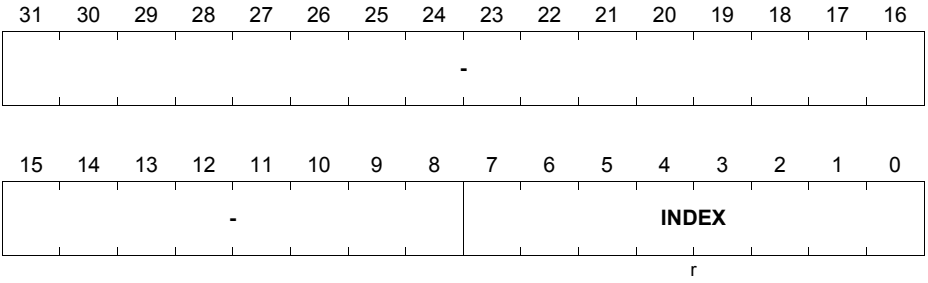
Field	Bits	Type	Description
V	31	r	**Valid bit** Indicates that the TTE contains a valid mapping. 0 : Invalid. 1 : Valid.
XE	30	r	**Execute Enable** Enables instruction fetches to the page. 0 : Disabled. 1 : Enabled.
WE	29	r	**Write Enable** Enables data writes to the page. 0 : Disabled. 1 : Enabled.
RE	28	r	**Read Enable** Enables data reads from the page. 0 : Disabled. 1 : Enabled.
G	27	r	**Global** Indicates that the page is globally mapped therefore making it visible in all address spaces. 0 : Not globally mapped. 1 : Globally mapped.

Field	Bits	Type	Description
C	26	r	**Cacheability** Indicates that the page is cacheable. 0 : Not Cacheable. 1 : Cacheable.
PSZ	[25:24]	r	**Page Size** 00_B : 1 KByte. 01_B : 4 KByte. 10_B : 16 KByte. 11_B : 64 KByte.
0	[23:22]	-	**Reserved Field**
PPN	[21:0]	r	**Physical Page Number** Holds the PPN from the PTE.

10.10.5 Translation Page Index Register (MMU_TPX)

The MMU_TPX register is used to return the TLB (Translation Lookaside Buffer) index result of a TLBPROBE instruction.

MMU_TPX
Translation Page Table Index Register **Reset Value: Implementation Specific**

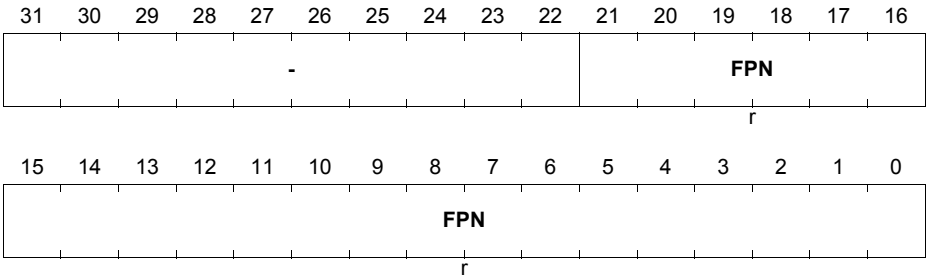


Field	Bits	Type	Description
-	[31:8]	-	**Reserved Field**
INDEX	[7:0]	r	**Translation Index**

10.10.6 Translation Fault Page Address Register (MMU_TFA)

The MMU_TFA register contains the faulting virtual page number (where faulting refers to a VAP or VAF trap). It is the faulting virtual address, right shifted by 10 + 2 * min (SZA, SZB) bits.

MMU_TFA
Translation Fault Page Address Register **Reset Value: Implementation Specific**



Field	Bits	Type	Description
-	[31:22]	-	**Reserved Field**
FPN	[21:0]	r	**Faulting Page Number** VPN from the faulting Virtual Address.

11 Core Debug Controller (CDC)

This chapter describes the debug features available in the TriCore™ CPU Core. The debug functionality is an interface of architecture, implementation and software tools, so users are advised that mechanisms may differ in subsequent architecture generations.

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.

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

CDC Features

CDC 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 CDC

To enable the CDC, the system containing the core must set the Debug Enable bit (DE) in the Debug Status Register (DBGSR). CDC is disabled when DBGSR.DE == 0, and enabled when DBGSR.DE == 1. How the DBGSR.DE bit is controlled and how CDC is enabled or disabled, is system dependent.

11.1 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 CDC provides hardware hooks into the core allowing the detection of Debug Events which result in Debug Actions.

Three signals are provided by the CDC 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-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 Control Register) or the MFCR (Move From Control Register) instruction.
  - Events raised by the Trigger Event Unit (see **Trigger Event Unit, page 11-4**).
- Debug Actions can be one or more of the following:
  - Update Debug Status register.
  - Indicate event on Core Break-Out signal or Core Suspend-Out signal.
  - Halt CPU execution.
  - Take Breakpoint Trap.
  - Raise Breakpoint Interrupt.
- 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.

11.2 Debug Events

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

- Core Break-In signal.
  – See **External Debug Event, page 11-3**.
- Execution of a DEBUG instruction.
  – See **Debug Instruction, page 11-3**.
- Execution of the MTCR or MFCR instruction.
  – See **MTCR and MFCR Instructions, page 11-3**.
- A hardware Event generation unit.
  – See **Trigger Event Unit, page 11-4**.

11.2.1 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, page 11-15**).

11.2.2 Debug Instruction

TriCore supports a User mode DEBUG instruction which can generate a Debug Event when CDC is enabled. When CDC 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, page 11-17**).

11.2.3 MTCR and MFCR Instructions

A Debug Event is raised when a MTCR (Move To Control Register) or MFCR (Move From Control 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 code reads or modifies one of the dedicated Debug SFRs (Special Function Registers):

- Debug Status Register (**DBGSR, page 11-13**).
- Core Register Access Event Register (**CREVT, page 11-16**).
- Software Debug Event Register (**SWEVT, page 11-17**).
- External Event Register (**EXEVT, page 11-15**).

- Trigger Event Register (TR*n*EVT) (**TR0EVT, page 11-18** and **TR1EVT, page 11-18**).
- Debug Monitor Start Register (**DMS, page 11-21**).
- Debug Context Pointer Register (**DCX, page 11-21**).

The Debug Action taken when the Debug Event is raised is specified in the CREVT register (See **CREVT, page 11-16**).

11.2.4 Trigger Event Unit

The Trigger Event Unit is responsible for generating Debug Events when a programmable set of Debug Triggers are active. Debug Triggers come from the protection system and 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 **Section 11.3 Debug Triggers, page 11-6** for more details on programmable combinations).

The Debug Action taken when the Debug Event is raised, is specified in the Trigger Event register (TR*n*EVT). See **page 11-18** for the register definition.

11.2.5 Priority of Debug Events

When two or more Debug Events occur on the same instruction, priorities are used to determine which Debug Event occurs. This section describes how those priorities are determined.

All Debug Events can be linked to specific instruction except for the external condition (EXEVT) Debug Event which is not linked to any instruction being executed. The latency of the EXEVT Debug Event is not defined.

When linking Debug Events to a specific instruction, they can be generated either logically before or logically after the execution of the instruction that they are linked with. This is known as Break Before Make (BBM) and Break After Make (BAM) respectively, and is controlled by the BAM[1)] bit in the various Debug Event registers.

Note: Data access and data/code combination access triggers can only create BAM Debug Events. When triggers occur, TRnEVT.BBM is ignored.

When the Core Debug Controller (CDC) is setup to create a Debug Event on two adjacent instructions, the Debug Event for the first instruction is always generated. The second instruction may not generate a Debug Event even if the Debug Action of the first instruction allows the program flow to continue to the second instruction.

1) EXEVT.BAM has no effect on the latency of external condition Debug Events.

When the CDC is setup to create both a BBM and a BAM Debug Event on the same instruction, the BBM Debug Event is created before the BAM Debug Event. If the BBM Debug Action does not prevent the instruction from executing, the BAM Debug Event will also be created when the instruction has executed.

When the CDC is setup to create more than one BBM Debug Event on the same instruction, only one Debug Event is generated. Similarly, when the CDC is setup to create more than one BAM Debug Event on the same instruction, only one Debug Event is generated. In both cases the Debug Event priorities are used to determine which Debug Event is generated.

Table 15 Debug Event Priorities

Priority (high to low)	Type of Debug Event
1	Debug Instruction (SWEVT) / Core Register Access (CREVT).
2	Trigger 0 (TR0EVT).
3	Trigger 1 (TR1EVT).
4	Trigger *n* (TR*n*EVT). Note that the number of TR*n*EVT registers is implementation dependent.

Note: The external condition may not generate a Debug Event even when programmed to do so, if a BAM Debug Event is generated in the same cycle. To avoid potential loss of EXEVT Debug Events, BBM Debug Events should be used.

11.3 Debug Triggers

The CDC 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 Debug Trigger Event Unit takes inputs from the Protection mechanism Range Table Entries (RTEs) and combines them as specified by the TR*n*EVT registers to produce a Debug Event. Range Table Entries that do not have a corresponding TR*n*EVT register can not be used to generate a Debug Event.

The RTEs which provide Debug Trigger information are those selected by the PRS (Protection Register Set) field in the PSW (Program Status Word) register (PSW.PRS). If it is not known which PRS is active, the RTEs in all potentially used PRSs should be programmed in the same way.

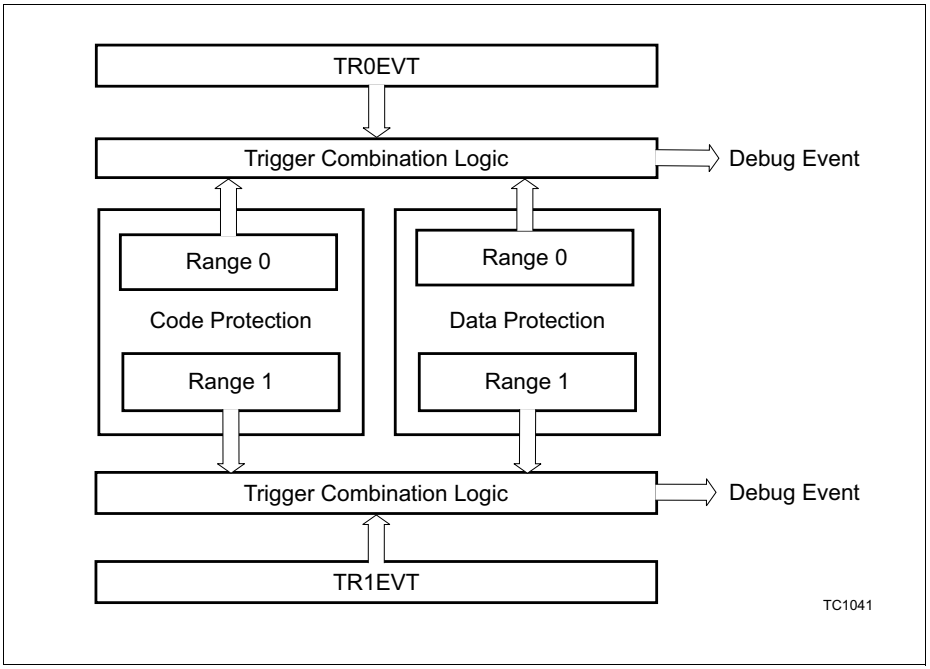


Figure 43 An Implementation Combination Example, Using two TR*n*EVT Registers

11.3.1 Combining Debug Triggers

The CDC allows Code and Data triggers to be combined to create a Debug Event. The combination is specified by the Trigger Event Register (TR*n*EVT).

The Trigger Event Unit can generate a number of Trigger Debug Events by combining four Debug Triggers for each Trigger Debug Event. The Debug Triggers are generated by the memory protection system. The four Triggers used to generate the Trigger*n* Debug Event come from the Code*n* and Data*n* Range Table Entries (RTE).

Table 16 Debug Triggers Generated by the Memory Protection System

Trigger	Description
D_U	Data read or write access to the upper bound of the Data RTE*n*, as enabled in the Data Protection Mode (DPM) register.
D_{LR}	Data read or write access to the lower bound or range of the Data RTE*n,* as enabled in the Data Protection Mode (DPM) register.
C_U	Code execution from the upper bound address of the Code RTE*n,* as enabled in the Code Protection Mode (CPM) register.
C_{LR}	Code execution from the lower bound address or address range of the Code RTE*n*, as enabled in the Code Protection Mode (CPM) register.

The combinations of Debug Triggers that generate a Debug Event are controlled by bits in the TR*n*EVT register. The possible combinations are given in **Table 17**.

Table 17 Debug Trigger Combinations that Generate a Debug Event

TR*n*EVT [11:8]	Data (D) and Code (C), Upper(U) and Lower(L) Bound, Combinations
0000	D_U or D_{LR} or C_U or C_{LR}
0001	(D_{LR} and C_{LR}) or D_U or C_U
0010	(D_{LR} and C_U) or D_U or C_{LR}
0011	(D_{LR} and C_{LR}) or (D_{LR} and C_U) or D_U
0100	(D_U and C_{LR}) or D_{LR} or C_U
0101	(D_{LR} and C_{LR}) or (D_U and C_{LR}) or C_U
0110	(D_{LR} and C_U) or (D_U and C_{LR})
0111	(D_{LR} and C_{LR}) or (D_{LR} and C_U) or (D_U and C_{LR})
1000	(D_U and C_U) or D_{LR} or C_{LR}
1001	(D_{LR} and C_{LR}) or (D_U and C_U)
1010	(D_{LR} and C_U) or (D_U and C_U) or C_{LR}

Table 17 Debug Trigger Combinations that Generate a Debug Event

TR*n*EVT [11:8]	Data (D) and Code (C), Upper(U) and Lower(L) Bound, Combinations
1011	(D_{LR} and C_{LR}) or (D_{LR} and C_U) or (D_U and C_U)
1100	(D_U and C_{LR}) or (D_U and C_U) or D_{LR}
1101	(D_{LR} and C_{LR}) or (D_U and C_{LR}) or (D_U and C_U)
1110	(D_{LR} and C_U) or (D_U and C_{LR}) or (D_U and C_U)
1111	(D_{LR} and C_{LR}) or (D_{LR} and C_U) or (D_U and C_{LR}) or (D_U and C_U)

Note: DBGSR.EVTSRC, DBGSR.PREVSUSP and DBGSR.SUSP are updated for all Debug Actions except 000 (None; disabled) and 101/110/111 reserved (same behaviour as 000).

11.4 Debug Actions

When a Debug Event occurs, one or more of the following Debug Actions are taken depending upon the programming of the External Event Register (EXEVT):

- **Update Debug Status Register (DBGSR), page 11-8**.
- **Indicate on Core Break-Out Signal, page 11-8** or **Indicate on Core Suspend-Out Signal, page 11-9**.
- **Halt, page 11-9**.
- **Breakpoint Trap, page 11-9**.
- Continue (results in no change to program flow).

11.4.1 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 TR*n*EVT).

11.4.2 Indicate on Core Break-Out Signal

A Debug Event can indicate to the OCDS that the Event has occurred. Note that it is system (i.e. implementation) dependent whether or not this signal is connected to an external pin.

11.4.3 Indicate on Core Suspend-Out Signal

On a Core Suspend-Out signal, 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 TR*n*EVT).

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.

11.4.4 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.

11.4.5 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.
- PC = DMS.
- PSW.PRS = 0_H.

- PSW.IO = $2_{H.}$
- PSW.GW = $0_{H.}$
- PSW.IS = $1_{H.}$
- ICR.IE = $0_{H.}$

The corresponding return sequence is provided through the privileged instruction RFM (Return From Monitor). This is effectively the reverse of the Breakpoint Trap entry sequence given above:

- PC = A[11].
- Restore PSW from DCX + $4_{H.}$
- Restore PCXI from DCX + $0_{H.}$
- Restore A[10] from DCX + $8_{H.}$
- Restore A[11] from DCX + $C_{H.}$

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 the CDC is disabled (i.e. DBGSR.DE == 0).

11.4.6 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 software breakpoint.

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.

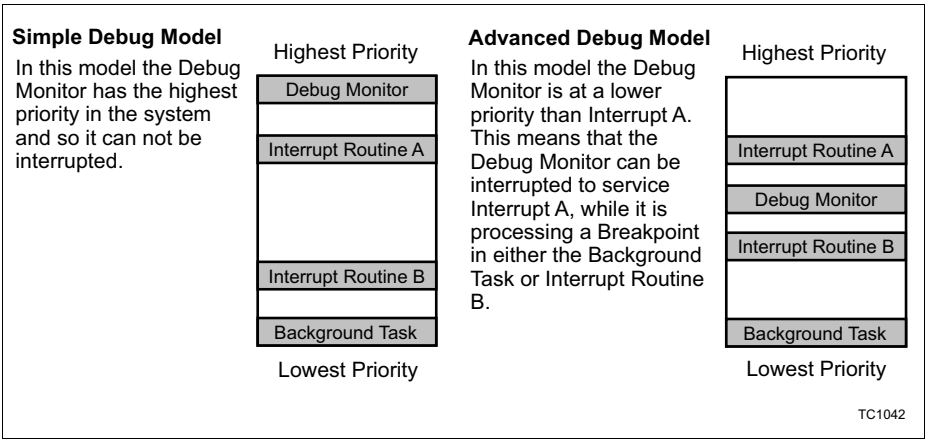


Figure 44 Debug Monitor - Simple and Advanced Models

11.4.7 Posted Software Events

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 44** 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.

11.4.8 Interrupts to Other Targets

As well as being targeted at the CPU, a software breakpoint can be targeted at other cores in the system.

11.5 CDC Control Registers

The Debug Status Register (DBGSR) contains information about the current status of the Core Debug Controller (CDC) hardware in the CPU core:

- A bit to indicate whether CDC is enabled.
- The source of the last Debug Event.
- The system debug level prior to 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.

Table 18 CDC Control Registers

Register	Description	Offset Address	Reference
DBGSR	Debug Status Register.	0000_H	page 11-13
EXEVT	External Event Register.	0008_H	page 11-15
CREVT	Core Register Access Event Register.	$000C_H$	page 11-16
SWEVT	Software Debug Event Register.	0010_H	page 11-17
TR0EVT	Trigger Event 0 Register.	0020_H	page 11-18
TR1EVT	Trigger Event 1 Register.	0024_H	page 11-18
DMS	Debug Monitor Start Address Register.	0040_H	page 11-21
DCX	Debug Context Save Area Pointer Register.	0044_H	page 11-21
SBSRC0	Software Breakpoint Service Request Control 0 Register.	$00BC_H$ [1]	page 11-22
SBSRC1	Software Breakpoint Service Request Control 1 Register. [2]	$00B8_H$ [1]	page 11-22
SBSRC2	Software Breakpoint Service Request Control 2 Register. [2]	$00B4_H$ [1]	page 11-22
SBSRC3	Software Breakpoint Service Request Control 3 Register. [2]	$00B0_H$ [1]	page 11-22

1) Software Breakpoint Service Request Control Registers are located in the address range of the CPU slave interface (CPS).

2) If implemented.

DBGSR
Debug Status Register

Reset Value: 0000 0000$_H$ (Boot Execute)
0000 0002$_H$ (Boot Halt)

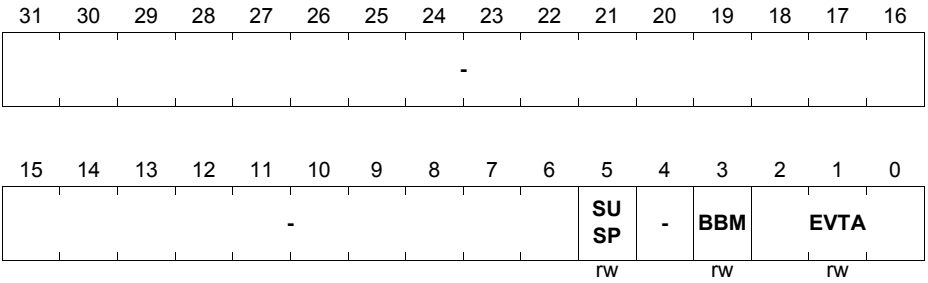
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
-															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-			EVTSRC					P EVT	PRE VSU SP	-	SU SP	-	HALT		DE
			rh					rwh	rh		rwh		rwh		rh

Field	Bits	Type	Description
-	[31:13]	-	**Reserved Field**
EVTSRC	[12:8]	rh	**Event Source** 0 : EXEVT. 1 : CREVT. 2 : SWEVT. 16 + n TR*n*EVT (n = 0, 1). 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.
-	5	-	**Reserved Field**
SUSP	4	rwh	**Current State of the Core Suspend-Out Signal** 0 : Core Suspend-Out inactive. 1 : Core Suspend-Out active.
-	3	-	**Reserved Field**

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 (CDC is enabled), set HALT[0]. If DBGSR.DE == 0 (CDC is not enabled), HALT[0] is left unchanged.
DE	0	rh	**Debug Enable** Indicates whether CDC is enabled. 0 : CDC disabled. 1 : CDC enabled.

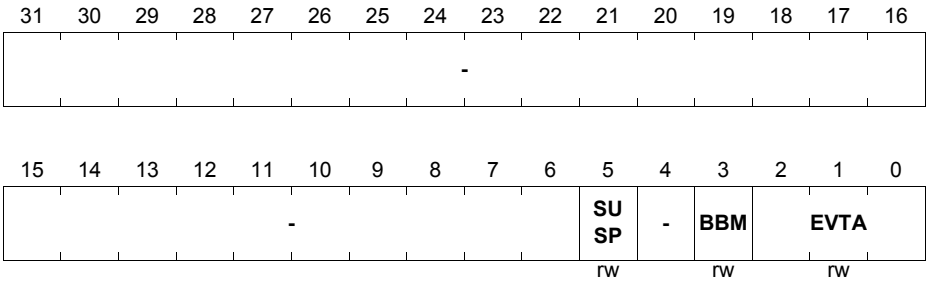
EXEVT
External Event Register **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
-	[31:6]	-	**Reserved Field**
SUSP	5	rw	**CDC Suspend-out Signal State** Value to be assigned to the CDC Suspend-Out signal when the Debug Event is raised.
-	4	-	**Reserved Field**
BBM	3	rw	**Break Before Make (BBM) or Break After Make (BAM) Selection** 0 : Break after make (BAM). 1 : Break before make (BBM).
EVTA	[2:0]	rw	**Event Associated** Specifies the Debug Action associated with the Debug Event: 000_B : None; disabled. 001_B : Pulse BRKOUT Signal. 010_B : Halt and pulse BRKOUT Signal. 011_B : Breakpoint trap and pulse BRKOUT Signal. 100_B : Software breakpoint 0 and pulse BRKOUT Signal. 101_B : If implemented, software breakpoint 1 and pulse BRKOUT Signal[1]. 110_B : If implemented, software breakpoint 2 and pulse BRKOUT Signal[1]. 111_B : If implemented, software breakpoint 3 and pulse BRKOUT Signal[1].

1) If not implemented, None.

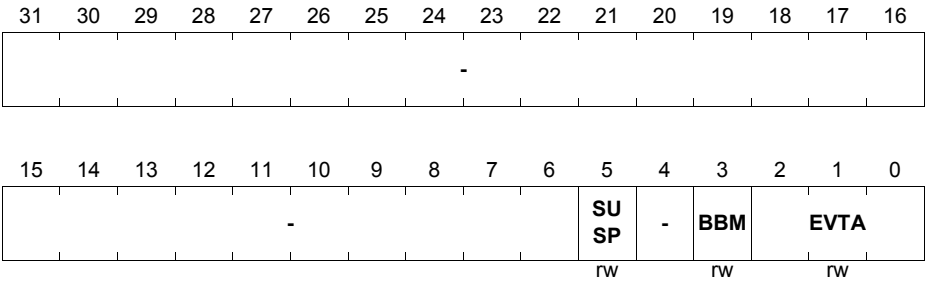
CREVT
Core Register Access Event **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
-	[31:6]	-	**Reserved Field**
SUSP	5	rw	**CDC Suspend-out Signal State** Value to be assigned to the CDC Suspend-out signal when the Debug Event is raised.
-	4	-	**Reserved Field**
BBM	3	rw	**Break Before Make (BBM) or Break After Make (BAM) Selection** 0 : Break after make (BAM). 1 : Break before make (BBM).
EVTA	[2:0]	rw	**Event Associated** Specifies the Debug Action associated with the Debug Event: 000_B : None; disabled. 001_B : Pulse BRKOUT Signal. 010_B : Halt and pulse BRKOUT Signal. 011_B : Breakpoint trap and pulse BRKOUT Signal. 100_B : Software breakpoint 0 and pulse BRKOUT Signal. 101_B : If implemented, software breakpoint 1 and pulse BRKOUT Signal[1]. 110_B : If implemented, software breakpoint 2 and pulse BRKOUT Signal[1]. 111_B : If implemented, software breakpoint 3 and pulse BRKOUT Signal[1].

1) If not implemented, None.

SWEVT
Software Debug Event **Reset Value: 0000 0000$_H$**



Field	Bits	Type	Description
-	[31:6]	-	**Reserved Field**
SUSP	5	rw	**CDC Suspend-out Signal State** Value to be assigned to the CDC Suspend-Out signal when the event is raised.
-	4	-	**Reserved Field**
BBM	3	rw	**Break Before Make (BBM) or Break After Make (BAM) Selection** 0 : Break after make (BAM). 1 : Break before make (BBM).
EVTA	[2:0]	rw	**Event Associated** Specifies the Debug Action associated with the Debug Event: 000_B : None; disabled. 001_B : Pulse BRKOUT Signal. 010_B : Halt and pulse BRKOUT Signal. 011_B : Breakpoint trap and pulse BRKOUT Signal. 100_B : Software breakpoint 0 and pulse BRKOUT Signal. 101_B : If implemented, software breakpoint 1 and pulse BRKOUT Signal[1]. 110_B : If implemented, software breakpoint 2 and pulse BRKOUT Signal[1]. 111_B : If implemented, software breakpoint 3 and pulse BRKOUT Signal[1].

1) If not implemented, None.

TR0EVT
Trigger Event 0 **Reset Value: 0000 0000$_H$**
TR1EVT
Trigger Event 1 **Reset Value: 0000 0000$_H$**

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ASI_EN				DU_U	DU_LR	DLR_U	DLR_LR	-		SUSP	-	BBM	EVTA		
rw				rw	rw	rw	rw			rw		rw		rw	

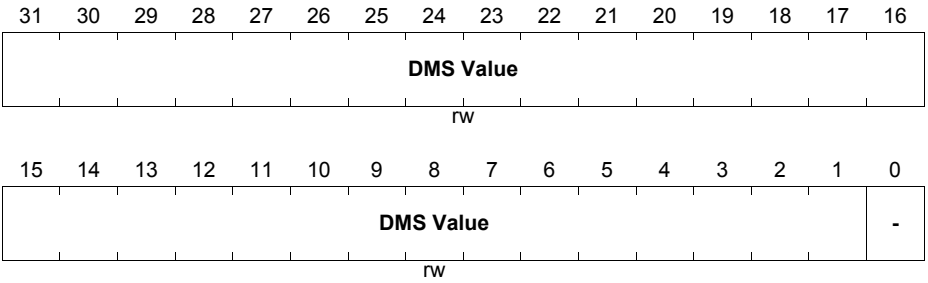
Field	Bits	Type	Description
-	[31:21]	-	**Reserved Field**
ASI	[20:16]		**Address Space Identifier** The ASI of the Debug Trigger process.
ASI_EN	15		**Enable ASI Compression** 0 : No ASI comparison performed. Debug Trigger is valid for all processes. 1 : Enable ASI comparison. Debug Events are only be triggered when the current process ASI matches TR*n*EVT.ASI. Field should be set to 0 for implementations without an MMU.
-	[14:12]	-	**Reserved Field**
DU_U	11	rw	**Controls combinations of D_U and C_U** Note: Refer to **Table 17**, **page 11-7** for clarification of trigger conditions that generate a Debug Event. 0 : D_U triggers event unless DU_LR == 1, where C_{LR} is also required. C_U triggers event unless DLR_U == 1, where D_{LR} is also required. 1 : D_U and C_U only trigger an event when they are both present.

Field	Bits	Type	Description
DU_LR	10	rw	**Controls combination of D_U and C_{LR}** Note: Refer to **Table 17**, **page 11-7** for clarification of trigger conditions that generate a Debug Event. 0 : D_U triggers event unless DU_U == 1. Where C_U is also required. C_{LR} triggers event unless DU_U == 1. Where D_U is also required. 1 : D_U and C_{LR} only trigger an event when they are both present.
DLR_U	9	rw	**Controls combination of D_{LR} and C_U** Note: Refer to **Table 17**, **page 11-7** for clarification of trigger conditions that generate a Debug Event. 0 : D_{LR} triggers event unless DLR_LU == 1. Where C_{LR} is also required. C_U triggers event unless DU_U == 1. Where D_U is also required. 1 : D_{LR} and C_U only trigger an event when they are both present.
DLR_LR	8	rw	**Controls combination of D_{LR} and C_{LR}** Note: Refer to **Table 17**, **page 11-7** for clarification of trigger conditions that generate a Debug Event. 0 : D_{LR} triggers event unless DLR_LU == 1. Where C_U is also required. C_{LR} triggers event unless DU_U == 1. Where D_U is also required. 1 : D_{LR} and C_{LR} only trigger an event when they are both present.
-	[7:6]	-	**Reserved Field**
SUSP	5	rw	**CDC Suspend-out Signal State** Value to be assigned to the CDC Suspend-Out signal when the Debug Event is raised.
-	4	-	**Reserved Field**

Field	Bits	Type	Description
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). Note that data access and data/code combination access triggers can only create BAM Debug Events. When these triggers occur, TR*n*EVT.BBM is ignored.
EVTA	[2:0]	rw	**Event Associated** Specifies the Debug Action associated with the Debug Event: 000_B : None; disabled. 001_B : Pulse BRKOUT Signal. 010_B : Halt and pulse BRKOUT Signal. 011_B : Breakpoint trap and pulse BRKOUT Signal. 100_B : Software breakpoint 0 and pulse BRKOUT Signal. 101_B : If implemented, software breakpoint 1 and pulse BRKOUT Signal[1]. 110_B : If implemented, software breakpoint 2 and pulse BRKOUT Signal[1]. 111_B : If implemented, software breakpoint 3 and pulse BRKOUT Signal[1].

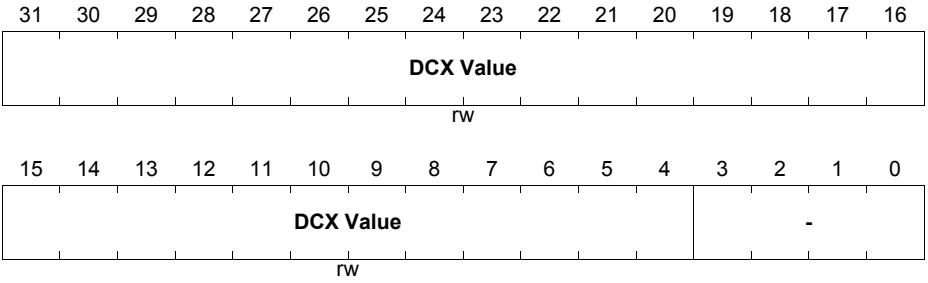
1) If not implemented, None.

DMS
Debug Monitor Start Address **Reset Value: Implementation Specific**



Field	Bits	Type	Description
DMS Value	[31:1]	rw	**Debug Monitor Start Address** DMS Value = DE00 00*n*0$_H$, where '*n*' is Core ID.
-	0	-	**Reserved**

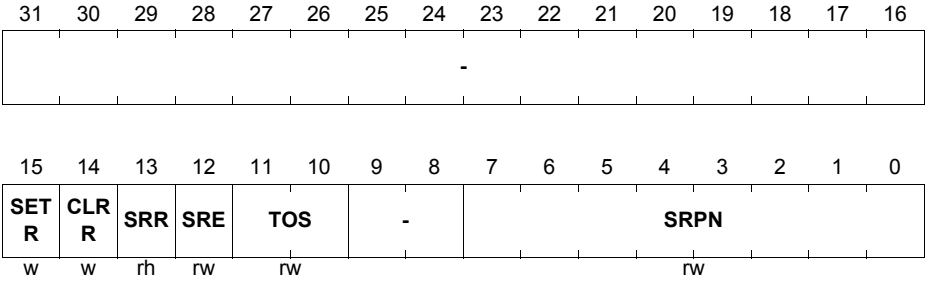
DCX
Debug Context Save Area Pointer **Reset Value: Implementation Specific**



Field	Bits	Type	Description
DCX Value	[31:4]	rw	**Debug Context Save Area Pointer** DCX Value = DE80 0000$_H$.
-	[3:0]	-	**Reserved**

11.5.1 Software Breakpoint Service Request Control Register

The Software Breakpoint Service Request Control Register (SBSRC*n*) defines the interrupt request parameters for a software breakpoint interrupt, where *n* = 0, 1, 2 or 3.



Field	Bits	Type	Description
-	[31:16]	-	**Reserved Field**
SETR	15	w	**Service Request Set** SETR is required to set SRR. 0 : No action. 1 : Set SRR. Written value is not stored. Read always returns 0. No action if CLRR is also set.
CLRR	14	w	**Service Request Clear** CLRR is required to set SRR. 0 : No action. 1 : Clear SRR. Written value is not stored. Read always returns 0. No action if SETR is also set.
SRR	13	rh	**Service Request Flag** 0 : No Software Breakpoint Service Request is pending. 1 : A Software Breakpoint Service Request is pending.
SRE	12	rw	**Service Request Enable** 0 : Software Breakpoint Service Request is disabled. 1 : Software Breakpoint Service Request is enabled.
TOS	[11:10]	rw	**Type Of Service Control** 00_B : Service Provider 0 - Typically CPU service is initiated. 01_B : Service Provider 1 - Implementation Specific. 10_B : Service Provider 2 - Implementation Specific. 11_B : Service Provider 3 - Implementation Specific.
-	[9:8]	-	**Reserved Field**

Field	Bits	Type	Description
SRPN	[7:0]	rw	**Service Request Priority Number** 00_H : Software Breakpoint Service Request is never serviced. 01_H : Software Breakpoint Service Request, lowest priority. … FF_H : Software Breakpoint Service Request, highest priority.

12 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, and even when present it can be disabled.

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

12.1 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.

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.

12.2 IEEE-754 Compliance

12.2.1 IEEE-754 Single Precision Data Format

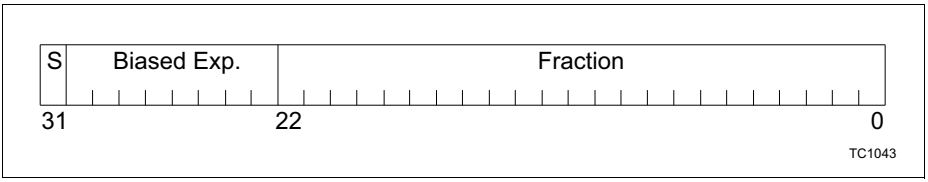


Figure 45 Single Precision IEEE-754 Floating-Point Format

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 20** 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 20 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	$+\infty$	+ infinity.
s == 1 AND e == 255 AND f == 0	$-\infty$	– 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.

12.2.2 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 12-7**.

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

12.2.3 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 12-10**.

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 $7FC00000_H$.

12.2.4 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 $< 2^{-126}$.

12.2.5 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.

12.2.6 Software Routines

Operations required for IEEE-754 compliance, but not implemented in the FPU instruction set, are detailed in **Table 21**.

Table 21 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.

12.3 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 22 Rounding Mode Definition

Rounding Mode Value	Mode
00[1)]	Round to nearest.
01	Round toward + ∞.
01	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 12-7**, 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.

12.3.1 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:

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

This is half way between:

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

and:

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

The result with the larger exponent is returned.

12.3.2 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 ($1.000 \ldots 000 * 2^{-126}$) and the largest representable positive IEEE-754 denormal number ($0.111 \ldots 111 * 2^{-126}$).

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

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

12.3.3 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 12-3**.

According to the IEEE-754 definition of the rounding modes, when rounding towards $+\infty$ ($-\infty$) 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.

12.4 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.

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 23 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 12-3**) is an operand for a FPU instruction.
- For invalid operations such as QSEED.F ($\approx 1/\sqrt{x}$) 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 24 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.[5)]
ADD.F with + ∞ and - ∞ as operands.	$7FC00001_H$
SUB.F with (+ ∞ and + ∞) or (- ∞ and - ∞) as operands.	$7FC00001_H$
MADD.F if the result of the multiplication is ± ∞ and the addend is the oppositely signed ∞.	$7FC00001_H$
MSUB.F if the result of the multiplication is ± ∞ and the minuend is the same signed ∞.	$7FC00001_H$
MUL.F with 0 and ± ∞ as multiplicands.	$7FC00002_H$
MADD.F with 0 and ± ∞ as multiplicands.	$7FC00002_H$
MSUB.F with 0 and ± ∞ as multiplicands.	$7FC00002_H$
QSEED.F with a negative operand[3)].	$7FC00004_H$
DIV.F with 0 as both operands[4)].	$7FC00008_H$
DIV.F with both operands being an ∞ of either sign.	$7FC00008_H$
FTOI, FTOU or FTOQ31 with rounded result outside the range of the target format.	n.a.[5)]
FTOI, FTOU or FTOQ31 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 12-3**.

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

FU - Underflow

As discussed in **Underflow, page 12-4**, 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 12-7**.

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.

13 Glossary

Reference	Definition
Addend	A number to be added to another number.
API	Application Program Interface - A set of routines, protocols and tools for building software applications.
ASI	Address Space Identifier - A TTE field.
ASIC	Application Specific Integrated Circuit.
ATPG	Automatic Test Pattern Generation
AV	Advance Overflow - PSW status flag.
Ax	Address Registers.
BCU	Bus Control Unit.
Biased exponent	The sum of the exponent and a constant (bias) chosen to make the biased exponent's range non-negative.
BISR	Begin Interrupt Service Routine.
BIST	Built-In Self Test.
BIV	Base Address of Interrupt Vector Table.
BPI	Bus Peripheral Interface
BTV	Base Address of Trap Vector Table.
CALL	Function call instruction.
CCPN	Current CPU Priority Number.
CDC	Call Depth Control field.
CLO	Count Leading Ones.
CLS	Count Leading Signs.
CLZ	Count Leading Zeros.
CMOV	Conditional Move instruction.
Context	Every Task has a Context. The Context is everything the processor needs in order to define the state of the associated task and enable its continued execution. Context's are subdivided into Upper & Lower Contexts.
CPM	Code Protection Mode.
CPR	Code Segment Protection Register.
CPS	CPU Slave.

Reference	Definition
CPU	Central Processing Unit.
CREVT	Core Register Access Event Register. Emulator Resource Protection Event register.
CSA	Context Save Area. Each CSA can hold 1 upper or 1 lower Context. CSAs are linked together through a Link Word.
CSFR	Core Special Function Register (SFR).
CVE	Core Verification Environment.
CVE	Co-Verification Environment.
DBGSR	Debug Status Register.
DC	Direct Current.
DCU	Data Control Unit.
DCX	Debug Context Save Area Pointer Register.
DEC	Decode Pipeline stage.
Denormalized number	A non-zero floating-point number whose exponent has the format's minimum, and whose implicit leading mantissa bit is zero.
DFF	D-type Flip Flop.
Dividend	A number to be divided by the divisor.
Divisor	A number by which a dividend is divided.
DMA	Direct Memory Access.
DMS	Debug Monitor Start Address Register.
DMU	Data Memory Unit.
DPM	Data Protection Mode.
DPR	Data Segment Protection Register.
DRAM	Direct Random Access Memory.
DSP	Digital Signal Processing.
Dx	Data Registers.
EBU	External Bus Unit.
EDA	Electronic Design Automation.
EMU	Emulation Monitoring Unit.
EQ	Equal - compare instruction.

Reference	Definition
EX1	Execute 1 pipeline stage.
EX2	Execute 2 pipeline stage.
EXEVT	External Event Register.
Exponent	The component of a binary floating-point number that normally signifies the integer power to which 2 is raised in determining the value of the represented number.
FCD	Free Context list Depletion trap.
FFT	Fast Fourier Transformations.
FI	Flag Invalid exception.
FPD	Fetch and Pre-decode.
FPI	Flexible Peripheral Interface - Used for on-chip interconnections, connecting the core with peripherals including ports and the External Bus Controller.
FPU	Floating Point Unit.
FPU	Floating-Point Unit.
FS	Flag some exception.
FU	Flag underflow exception.
FV	Flag overflow exception.
FX	Flag inexact exception.
FZ	Flag divide by zero exception.
GDI	Graphical Device Interface.
GE	Greater Than or Equal to - compare instruction.
GPR	General Purpose Register.
GPTU	General Purpose Timer Unit.
GRWP	Global Register Write Protection.
HLL	High Level Language.
I/F	Interface.
IC	Integrated Circuit.
ICR	Interrupt Unit Control Register.
ICU	Interrupt Control Unit.
ID	Identity / Identification.
IE	Interrupt Enable.

Reference	Definition
IEEE	Institute of Electrical & Electronics Engineers.
IIR	Infinite Impulse Response A digital filter with internal registers that hold past output of the filter.
Infinite precision	The result of a floating-point operation assuming unbounded exponent and mantissa bits.
IOPC	Illegal Opcode.
IS	Interrupt Stack.
ISA	Instruction Set Architecture.
ISP	Interrupt Stack Pointer.
ISR	Interrupt Service Routine.
JTAG	Joint Test Action Group.
LDCUX	Instruction to Load Upper Context from memory.
LDLCX	Instruction to Load Lower Context from memory.
LDMST	Load Modify Store instruction.
LEA	Load Effective Address.
LEADDR	LMB Error Address.
LEATT	LMB Error Attributes.
LEDAT	LMB Error Data.
LFI	LMB to FPI Interface: A bridge between the two main buses, LMB and FPI.
LMU	Local Memory Unit.
LT	Less Than - compare instruction.
MAC	Multiply and Accumulate.
MAC	Multiply and Accumulate.
Mantissa	Tthe component of a binary floating-point number that consists of the implicit leading bit to the left of the implied binary point, and the fractional field.
MCU	MicroController Unit.
Minuend	A number from which a subtrahend is to be subtracted.
MIPS	Million Instructions Per Second.

Reference	Definition
MMU	Memory Management Unit - translates virtual addresses issued by the load / store, instruction fetch unit into physical addresses to feed into the PMU and DMU respectively.
MTCR	Move To Control Register.
Multiplicand	A number to be multiplied by another number.
NaN	Not a Number.
NE	Not Equal compare instruction.
NMI	Non-Maskable Interrupt.
Normalized number	A floating-point number whose implicit leading mantissa bit is one.
OCDS	On-Chip Debug Support.
Operand	Data that is to be operated on.
OTP	One-Time Programmable.
PC	Program Counter.
PCP	Peripheral Control Processor - an I/O control processor that performs tasks typically handled by a dedicated DMA controller and CPU interrupt service routines.
PCPN	Previous CPU Priority Number.
PCXI	Previous Context Information Register.
PIPN	Pending Interrupt Priority Number.
PMU	Program Memory Unit.
PPN	Physical Page Number (see Address Spaces).
PRS	Protection Register Set (see **Memory Protection System**).
PSW	Processor Status Word.
PTE	Page Table Entry (see **Address Translation**).
Quotient	The number resulting from the division of one number by another.
RA	Return Address register.
RAM	Random Access Memory.
Remainder	The final undivided part after division that is less or of lower degree than the divisor.
RET	Return from CALL instruction.

Reference	Definition
RFE	Return From Exception - return from an interrupt or trap handler.
RISC	Reduced Instruction Set Computer.
RM	Rounding Mode.
ROM	Read Only Memory.
Rounded result	The infinitely precise result rounded to fit into IEEE-754 format.
RSLCX	Restore Lower Context instruction.
RSTV	Reset Overflow Bits.
RTL	Register Transfer Level.
RTOS	Real-Time Operating System.
SAV	Sticky Advance Overflow - PSW status flag.
SDF	Standard Delay Format.
SFR	Special Function Register.
SFR	Special Function Register.
Significand	See mantissa.
SIMD	Single Instruction Multiple Data.
SMT	Software Managed Tasks.
SOC	System-On-a-Chip.
SP	Stack Pointer.
SPR	Scratch Pad RAM.
SRAM	Static Random Access Memory.
SRE	Software Request Enable field in the SRC register.
SRN	Service Request Node.
SRPN	Service Request Priority Number.
STA	Static Timing Analysis.
STLCX	Store Lower Context instruction.
STPG	Static Test Pattern Generation.
STUCX	Store Upper Context.
Subtrahend	A number that is to be subtracted from a minuend.
SV	Sticky Overflow - PSW status flag.
SVLCX	Save Lower Context instruction.

Reference	Definition
SWEVT	Software Debug Event Register.
SYSCON	System Control Register.
Task	Refers to an independent thread of control. There are two types of tasks: Software Managed Tasks (SMTs) and Interrupt Service Routines (ISRs).
TC	Abbreviation for TriCore (TriCore 1 or TriCore 2, for example).
TFA	Translation Fault Address.
TIN	Trap Identification Number - Identifies the cause of a trap within its class (TriCore has 8 Trap classes).
TLB	Translation Lookaside Buffer.
TOS	Type Of Service field.
TPA	Translation Physical Address.
TPX	Translation Page Index.
TRAPSV	Trap on Sticky Overflow instruction.
TRAPV	Trap on Overflow instruction.
TR*n*EVT	Trigger Event *n* Specifier register.
TSIM	TriCore Instruction Set Simulator (Tricore SIMulator) - A configurable, instruction-accurate model of the TriCore core architecture, providing a simulation environment that models the core, memory configuration and interrupt mechanism. Used for debugging and developing customized designs.
TTE	TLB Table Entries (see Memory Management).
UL	Upper Lower context.
Unrounded result	The infinitely precise result.
UOPC	Unimplemented Opcode.
V	Overflow - PSW status flag.
VAP	Virtual Address Protection trap.
VPN	Virtual Page Number.
WB	Write Back pipeline stage.

14 List of Registers

15 Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

http://www.infineon.com

Published by Infineon Technologies AG

Ordering No. B158-H8581-X-X-7600
Printed in Germany
UM 0205.15