TriCore

32-bit Unified Processor Core

Embedded Applications Binary Interface (EABI)

32bit

Microcontrollers

Never stop thinking

Edition 2007-02

Published by
Infineon Technologies AG
81726 München, Germany

Legal Disclaimer

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics (“Beschaffenheitsgarantie”). With respect to any examples or hints given herein, any typical values stated herein and/or any information regarding the application of the device, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

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

32-bit Unified Processor Core

Embedded Applications Binary Interface (EABI)

Microcontrollers

Never stop thinking

TriCore EABI User's Manual Revision History	2007-02	v2.3

Previous Version

Page	Subjects (major changes since last revision)
	Revised for publication.

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

1 Introduction

1.1 Scope

This manual defines the Embedded Applications Binary Interface (EABI) for Infineon Technologies TriCore™ 32-bit Unified microprocessor architecture. The EABI is a set of interface standards that writers of compilers and assemblers and linker/locators must use when creating compliant tools for the TriCore architecture. These standards cover run-time aspects as well as object formats to be used by compatible tool chains. A standard definition ensures that all TriCore tools are compatible and can interoperate at the binary object level.

This manual does not describe how to write TriCore development tools. Neither does it define the services provided by an operating system, or the library interfaces beyond libc and basic mathematical functions. Those tasks must be performed by suppliers of tools, libraries, and operating systems.

Source code compatibility and assembly language compatibility are separate issues which are not covered by this document.

1.2 Purpose

The standards defined in this manual ensure that all conforming chains of development tools for TriCore will be compatible at the binary object level. Compatible tools can interoperate and therefore make it possible to select an optimum tool for each link in the chain, rather than selecting an entire chain on the basis of overall performance.

The standards in this manual also ensure that compatible libraries of binary components can be created and maintained. Such libraries make it possible for developers to synthesize applications from binary components, and can make libraries of common services stored in on-chip ROM available to applications executing from off-chip memory. With established standards, developers can build up libraries over time with the assurance of continued compatibility.

1.3 Overview

Standards in this manual are intended to enable creation of compatible development tools for the TriCore, by defining minimum standards for compatibility between:

- Object modules generated by different tool chains.
- Object modules and the TriCore processor.
- Object modules and source level debugging tools.

These standards do not entirely disallow vendor-specific extensions and feature enhancements aimed at differentiating a vendor’s product. However, to claim compliance with this manual, a tool set must support an “EABI compliance mode”, in which any incompatible extensions are suppressed. “Incompatible extensions” are

defined as those that, if used, would prevent the object module output of the tool chain from being linked in and used by applications compiled with tool chains supporting only the standards defined herein.

Current definitions include the following types of standards:

1.3.1 Low-Level Run-Time Binary Interface Standards

- Processor specific binary interface.
  - The instruction set, representation of fundamental data types, and exception handling.
- Function calling conventions.
  - How arguments are passed and results are returned, how registers are assigned, and how the calling stack is organized.
- Memory models.
  - How code and data are located in memory.

1.3.2 Object File Binary Interface Standards

- Header convention.
- Section layout.
- Symbol table format.
- Relocation information format.
- Debugging information format.

1.3.3 Naming Conventions

- 'C' name mapping.

1.4 Associated Documentation

Please refer to the TriCore Architecture Manual for a detailed discussion of instruction set encoding and semantics.

2 Low Level Binary Interfaces

2.1 Underlying Processor Primitives

The complete TriCore™ architecture is described in the TriCore Architecture Manual.

2.1.1 Registers

The TriCore EABI (Embedded Application Binary Interface) defines how to use the 32 general-purpose 32-bit registers of the TriCore processor. These registers are named A[0] through to A[15] (Address registers), and D[0] through to D[15] (Data registers).

TriCore also has a number of control registers. Those referenced in this document that are relevant to the programming model are shown in the following table. Please refer to the TriCore Core Architecture manual for further details of the control registers.

Table 1 TriCore Control Registers

Register Name	Description	Type
PSW	Program Status Word	CSFR
PCXI	Previous Context Information	CSFR
PC	Program Counter (Read Only)	CSFR
FCX	Free Context List Head Pointer	Context Pointer
LCX	Free Context List Limit Pointer	Context Pointer
ISP	Interrupt Stack Pointer	Interrupt/trap
ICR	Interrupt Control Register	Interrupt/trap
PIPN	Pending Interrupt Priority Number	Interrupt/trap
BIV	Base Address Of Interrupt Vector Table	Interrupt/trap
BTV	Base Address Of Trap Vector Table	Interrupt/trap

2.1.2 Fundamental Data Types

The TriCore processor works with the following fundamental data types:

- Boolean; values FALSE (0) and TRUE (1 when set, $\neq 0$ when compared).
- Bit string; a packed field of 32-bits.
- Unsigned/signed character of 8-bits.
- Unsigned/signed short integer of 16-bits.
- Unsigned/signed integer of 32-bits.
- Unsigned/signed long long integer of 64-bits.
- Address of 32-bits.
- Signed fraction of 16 bits (DSP support, 1Q.15 format).
- Signed fraction of 32-bits (DSP support, 1Q.31 format).

Hardware support for the following data types is not included in the basic TriCore Instruction Set Architecture (ISA). They may be supported in particular implementations through coprocessor extensions to the ISA, otherwise they must be supported through software emulation:

- IEEE-754 floating point numbers.
  - Depending on the particular implementation of the core architecture, IEEE-754 floating point numbers are supported by direct hardware instructions or by software emulation.
- Long long integers.
  - Long long integers have only limited support in the hardware and some operations can be emulated in software.

The mapping between fundamental data types and the C language data types is shown in the following table.

Table 2 Mapping of C Data Types to the TriCore

ANSI C	Size	Align	TriCore
char	1	1	Signed character
unsigned char	1	1	Unsigned character
short	2	2	Signed short integer
unsigned short	2	2	Unsigned short integer
long	4	4	Signed integer
unsigned long	4	4	Unsigned integer
int	4	4	Signed integer
unsigned int	4	4	Unsigned integer
enum	1-4	1-4	Signed integer
pointer	4	4	Address

Table 2 Mapping of C Data Types to the TriCore (Continued)

ANSI C	Size	Align	TriCore
long long	8	4	Unsigned integer[2]
unsigned long long	8	4	Unsigned integer[2]
float	4	4	Unsigned integer
double	8	4	Unsigned integer[2]
long double	8	4	Unsigned integer[2]

Characters must be stored on byte boundaries. Short integers must be two byte aligned. In EABI compliance mode, integers and long long integers must be four byte aligned.

Enumeration types are signed integers of size 1, 2, or 4 Bytes. The size must be chosen to accommodate the largest value in the enumeration. Enumerations are aligned according to their size.

The TriCore uses little-endian byte ordering consistently for both data and instructions. The highest addressable byte of a memory location always contains the most significant byte of the value.

The TriCore architecture does not support the long long int data type with 64-bit arithmentic operations. However compliant compilers must emulate the data type.

2.1.3 Special Data Types

2.1.3.1 Circular Buffer Pointers

A circular buffer pointer is a two-word structure containing a generic pointer and two half-word integers. The pointer points to the base of an 8 Byte aligned circular buffer; the integers describe the size of the buffer and an index into it. Circular buffer pointers are 4 Byte aligned.

2.1.4 Compound Data Types

Arrays, structures, unions, and bit fields have special alignment characteristics, as described in the following sub-sections.

2.1.4.1 Arrays

Arrays of byte elements may be aligned on any byte boundary. Arrays of 16-bit elements must be aligned on 2 Byte boundaries. Arrays of 32-bit (word) or 64-bit (double-word) elements must be aligned on 4 Byte boundaries. Arrays of composite elements (unions, structures, or subarrays) must be aligned according to the alignment requirement of the composite elements comprising the array.

Note: Any array used as a circular buffer (i.e. whose elements are accessed using the circular addressing mode) must be aligned on a double-word boundary.

The requirement for word alignment on arrays of word elements is not architecturally imposed, provided that the word elements are not pointers. It is imposed for the benefit of software that may use an array of integers as anonymous storage, sometimes holding pointer elements. Such software is not strictly portable, under rules of ANSI C, but it's sufficiently common that it can not be dismissed. Since the memory cost of an occasional half-word of padding needed to align the array is small, and since word alignment does slightly improve load/store performance, the constraint is justified.

2.1.4.2 Unions and Structures

Unions and structures have the most restrictive alignment of their members. For example, a structure containing a char and an int must have 4 Byte alignment to match the alignment of the int field. In addition, the size of a union or structure must be an integral multiple of its alignment. Padding must be applied to the end of a union or structure to make its size a multiple of the alignment.

Members must be aligned within a union or structure according to their type; padding must be introduced between members as necessary to meet this alignment requirement.

To facilitate copy operations, any structure larger than 1 Byte must have a minimum 2 Byte alignment, even if its only members are byte elements.

Examples:

```
struct one
{
  char c1;
  int  i1;
  char c2;
}
```

will have the following layout:

4n	c1			
4(n+1)	i1 (LSB)	i1	i1	i1 (MSB)
4(n+2)	c2			

```
struct two
{
  char c1;
  short s1;
  char *a1;
}
```

will have the following layout:

4n	c1		s1 (LSB)	s1 (MSB)
4(n+1)	a1 (LSB)	a1	a1	a1 (MSB)

```
struct three
{
  char c1;
  double d1;
}
```

will be layed out as follows:

4n	c1			
4(n+1)	d1 (LSB)	d1	d1	d1
4(n+2)	d1	d1	d1	d1 (MSB)

2.1.4.3 Bit Fields

Individual bit fields cannot exceed 32-bits in width, neither can they cross more than one half-word (16-bit) boundary. Outside of these restrictions, adjacent bit fields are packed together with no padding in between, except as required for the special case of a zero-width bit field (A zero-width bit field, as specified by ANSI C, forces alignment to a storage unit boundary, which for TriCore is one byte).

Bit fields are assigned in little-endian order; i.e. the first bit field occupies the least significant bits while subsequent fields occupy higher bits.

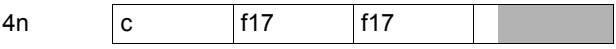
- Unsigned bit fields range from 0 to 2^W - 1 (where W is the size in bits).
- Signed bit fields range from -2^{W-1} to 2^{W-1} - 1.
- Plain int bit fields are unsigned.

In ANSI C, bit fields must be declared as integer types. However, in implementations compliant with this manual, the alignment requirements they impose as members of unions or structures are the same as those that would be imposed by the smallest integer-based data types wide enough to hold the fields. Therefore:

- Fields whose width is 8-bits or less impose only byte alignment.
- Fields whose width is from 9 to 16-bits impose half-word alignment.
- Fields whose width is from 17 to 32-bits impose word alignment.

Except for the special case of zero-width bit fields, or to comply with the prohibition against crossing more than one half-word boundary, bit fields are always allocated beginning with the next available bit in the union or structure. No alignment padding is inserted, except for the two cases noted. In the following example therefore, the character c will occupy bit positions [7:0], while the 17-bit field f17 will occupy positions [24:8]. Padding will be inserted in bits [31:25] to complete a full word, and the structure will be word alignmened.

```
struct bits_1 {
  char c;
  int f17 : 17;
};
```

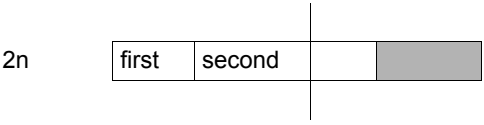


If the size of field is increased to 25-bits however, the half-word boundary crossing rule applies, and 8-bits of padding are inserted in bits [15:8].

```
struct bits_2 {
  char c;
  int f25 : 25;
};
```

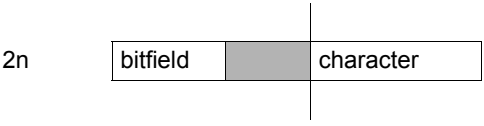
In the following example, the structure bits_3 has 2 Byte alignment and will have size of 2 Bytes. The 2 Byte alignment follows from the rule that any structure larger than 1 Byte must have at least 2 Byte alignment, not from the presence of the bit fields. The field 'second' occupies bits [10:3] of the structure.

```
struct bits_3 {
  int first : 3;
  unsigned int second : 8;
};
```



In this final example, the offset of the field 'character' is one byte. The structure itself has 2 Byte alignment and is 2 Bytes in size.

```
struct bits_4 {
  int bitfield : 5;
  char character;
};
```



2.1.5 Non-standard Alignment Options

2.1.5.1 Discrete Word and Double Word Variables

In general the TriCore architecture supports access to 32-bit integer variables on half-word boundaries. Depending on memory configuration, double-word variables may also be accessed on half-word boundaries. However there are exceptions and limitations on such accesses that preclude use of half-word alignment as the EABI standard for word and double-word data objects. A tool vendor may nonetheless elect to support half-word alignment of these objects, provided that the support is conditional on the compilation mode or specific user directives. In EABI compliance mode, word alignment must be followed.

The following are the particular restrictions of which tool vendors should be aware, when deciding whether to support an option for half-word alignment of word and double-word data objects:

- An extra cycle is taken, when the word or double-word accessed crosses a cache line boundary.
- In off-chip memory accessed via TriCore's FPI bus, double-word accesses on a half-word boundary are illegal and will generate a bus error.
- Pointers must always be word aligned, if they are to be loaded into or stored from the address register file. Legacy code that assumes the interchangeability of integers and pointers may fail, if the two are handled differently. For example, if the address of an "anonymous" word variable, with a dummy declaration as "int", is passed to a callback function that interprets its argument as the address of a pointer, the callback function may fail when it attempts to load the referenced argument into an address register.
- Integer variables that are used as semaphores must be word aligned. The SWAP instruction, used to implement semaphore operations on TriCore, requires its operand to be word aligned.
- The LDMST instruction, used to atomically store a bit field into memory, also requires its operand to be word aligned.

2.1.5.2 Packed Unions and Structures

In order to minimize space consumed by alignment padding within unions and structures, an implementation may elect to support a compilation mode in which integer and double-word member elements follow the minimum architecturally imposed alignment requirements (half-word boundaries), rather than the more conservative word alignments specified above. However, support for this mode is not an EABI requirement, and users must be warned that its use will result in creation of binary objects that are not EABI compliant.

2.1.5.3 Size-based Alignment of Data Types

In **Table 2 page 2-2**, the alignment requirements for data types are the same as their sizes, except for double-word types which are word aligned. Traditional RISC machines however, often require alignments to match sizes for all data types, including double-words.

To facilitate binary data exchange with such machines, an implementation may elect to support a compilation mode or user directive that forces double word member elements of unions and structures to be double-word aligned.

As with packed unions and structures, support for this mode is an option, and is not a requirement of the EABI. Use of this option must be suppressed when the user has requested EABI compliance mode.

2.2 Standard Function Calling Conventions

2.2.1 Register Assignments

The context of a task is divided into the upper context and the lower context.

2.2.1.1 Upper Context

The upper context consists of the upper address registers A[10] through A[15] and the upper data registers D[8] through D[15], as well as the processor status word (PSW) and the previous context information (PCXI). The upper context is saved automatically as the result of an external interrupt, exception, system call, or function call. It is restored automatically as a result of executing a RET or RFE instruction.

2.2.1.2 Lower Context

The lower context consists of the lower address registers A[2] through A[7], the lower data registers D[0] through D[7], and the saved program counter (A11). The lower context is not preserved automatically across interrupts, exceptions, or calls. There are special instructions, SVLCX and RSLCX, to save and restore the lower context.

2.2.1.3 Implicit Operands

Some of the general purpose registers serve as an implicit source or destination for certain instructions. These are listed in the following table:

Table 3 Register Assignments

Register	Use
A[0], A[1], A[8], A[9]	System Global Address Registers
D[15]	Implicit data register for many 16-bit instructions.
A[10]	Stack Pointer (SP).
A[11]	Return address register (RA) for CALL, JL, JLA, and JLI Return PC value on interrupts .
A[15]	Implicit base address register for many 16-bit load/store instructions.

2.2.1.4 System Global Registers

Address registers A[0], A[1], A[8], and A[9] are designated as system global registers. They are not part of either context partition and are not saved/restored across calls. They can be protected against write access by user applications.

By convention, A[0] and A[1] are reserved for compiler use, while A[8] and A[9] are reserved for OS or application use. A[0] is intended as a base pointer to the “small” data section, where global data elements can be accessed using base + offset addressing. It is part of the execution environment and will always be initialized in the startup code for any EABI-compliant RTOS.

A[1] is intended as a base pointer to the “literal data section”. The literal data section is a read-only data section intended for holding address constants and program literal values. Like A[0], it is initialized in the startup code for any EABI-compliant RTOS. (See **Special Sections, page 4-4**, for further information.)

As noted, A[8] and A[9] are reserved for OS use, or for application use in cases where the application and OS are tightly coupled. The compiler may support a directive that allows a global pointer variable to be bound to one of these registers. In the absence of such a directive the registers can only be used from assembly coded functions. A typical OS use would be as a pointer to the current task control block, or to the task ready queue, for reducing executive overhead in task management functions.

2.2.1.5 Cross-Call Lifetimes

Upper context registers are preserved automatically across function calls implemented with the TriCore CALL and RET instructions. All upper context registers other than A[10](SP) and A[11](RA)), are architecturally undefined after a CALL, and the registers can not be used to pass arguments to a called function. Lower context registers are not preserved; the calling function is responsible for preserving any values residing in lower context registers that are live across a call. The callee may use both upper and lower context registers without concern for saving and restoring their contents.

2.2.2 Stack Frame Management

2.2.2.1 Frame Layout

The stack pointer (SP) points to the bottom (low address) of the stack frame. The stack pointer alignment is 8 Bytes (i.e. the OS must initialize the stack pointer to a double-word boundary, and the compiler must size all stack frames to an integral number of double-words.)

Stack Frame Layouts for Three Calls, page 2-12 shows typical stack frames for three functions. The argument overflow area (see **Overflow Arguments on the Stack, page 2-15**) for outgoing arguments must be located at the bottom (low address end) of the frame, with the first overflow argument at zero offset from the stack pointer.

The caller is responsible for allocating the overflow area on the stack. The caller must also reserve stack space for return variables that do not fit in the first two argument registers (e.g. structure returns). This return buffer area is typically located with the local

variables. This space is typically allocated only in functions that make calls returning structures, and is not required otherwise.

Local variables that do not fit into the local registers are allocated space in the Local Variable area of the stack. If there are no such variables, this area is not required.

Beyond these basic requirements the conventions used for stack frame layout are at the discretion of the compiler implementation. For example, an implementation may choose to sort local data items by size, placing byte variables together at the bottom of the frame, followed by all half-word variables, etc. This enables all alignment constraints to be met with a total of at most seven bytes of padding in the frame. It also optimizes the frame layout for TriCore's addressing model, which allows only 10-bit offsets for addressing of byte and half-word variables, but supports 16-bit offsets for word variables. However, it is not required that all compiler implementations adopt this model. Frame layout is not a binary compatibility issue, because C does not present any reason for a called function to be aware of its caller's frame layout.

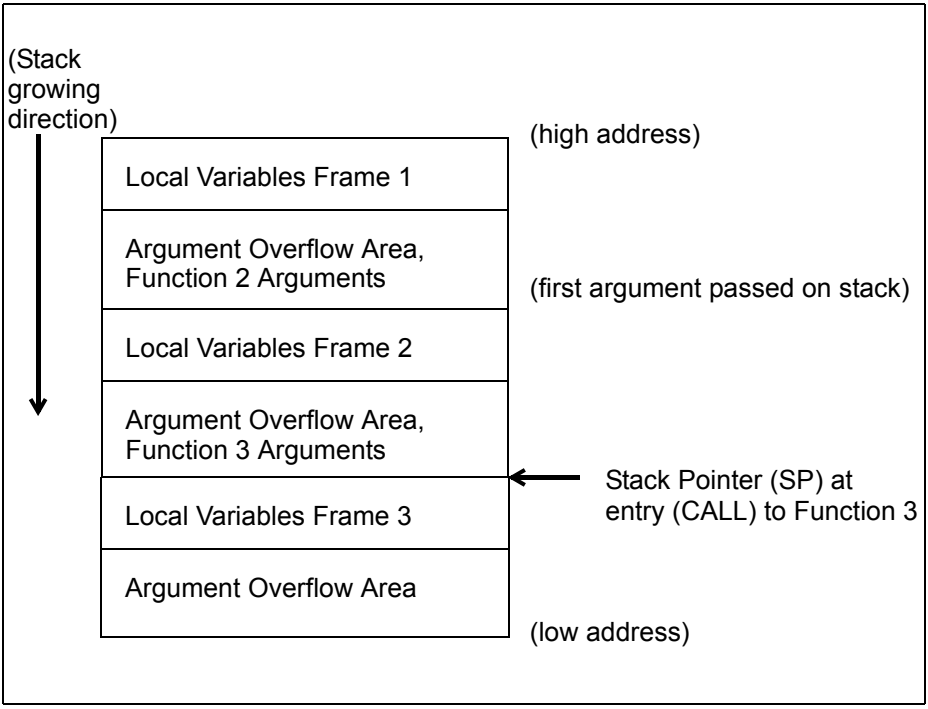


Figure 1 Stack Frame Layouts for Three Calls

2.2.2.2 Frame Addressing

As with the frame layout conventions, the conventions by which applications code addresses local data items within frames are the responsibility of the compiler, and are not dictated by this document.

If an implementation does not support dynamic allocation of variably-sized local data objects on the stack, then the offset to any local data object from the stack pointer will be static. In that case the most natural and efficient convention is to address locals through base + offset addressing, with the stack pointer as base. However, this is a recommendation, not a requirement.

For implementations that support the Gnu C _alloca() intrinsic, or other extensions that allow local allocation of objects whose size is determined at run time, it is not always feasible to use the stack pointer as a base for addressing local data items. A separate frame pointer will sometimes be needed.

Note: It is the responsibility of the calling function to manage its own frame pointer.

The called function is not required to have any knowledge of the caller's frame pointer conventions, or to co-operate in management of the frame pointer. If the caller uses an upper context address register as its frame pointer, then no action is needed to insure that the frame pointer is preserved across the call. If a lower context register is used (For example A[3], which is otherwise used only for the high-order part of a 64-bit circular buffer pointer return value), then the caller must save and restore the frame pointer value from a location that is at a static offset from the stack pointer; For example, just above the argument overflow area and below any dynamically allocated structures.

2.2.3 Argument Passing

This section describes the default argument passing conventions. It is mandatory for conforming tools.

The following conventions require correct use of function prototypes throughout the source code. Where a prototype is not provided the compiler must infer it from the actual parameters.

The conventions described here assume the use of the TriCore call / return mechanism, which automatically saves registers D[8] through D[15] and A[10] through A[15] as a side effect of the CALL instruction, and restores them as a side effect of the RET instruction. The registers saved automatically include the stack pointer A[10], so a called function requires no epilog to restore the caller's stack pointer value prior to returning.

The TriCore uses eight registers (D[4] through D[7], and A[4] through A[7]) to pass up to four non-pointer arguments and up to four pointer arguments from the caller to the called routine.

Arguments beyond those that can be passed in registers are passed on the stack, at the bottom (low address end) of the caller's stack frame. On entry to a called routine, the first

of such arguments is at offset zero from the stack pointer; subsequent arguments are at higher offsets.

2.2.3.1 Non-Pointer Arguments

Up to four 32-bit non-pointer arguments are passed in registers D[4] through D[7], with no more than one argument assigned per register. Integral arguments whose natural size in memory is less than 32-bits (characters and short integers) are expanded to 32-bits for purposes of argument passing.

Argument registers are generally allocated to arguments in numeric order, left to right, according to argument type. D[4] would be allocated to the first non-pointer argument.

Small arguments that are passed on the stack are placed in the overflow area with the same orientation they would have if passed in a register; a char is passed in the low-order byte of its overflow word. Such small overflow arguments need not be sign-extended within the argument word as they would be if passed in a register.

2.2.3.2 64-bit Arguments

Up to two 64-bit arguments are passed in the register pairs D[4]/D[5] (extended register E[4]) and D[6]/D[7] (extended register E[6]). 64-bit arguments must not be split when there are too few argument registers to hold the entire argument. Arguments larger than 64-bits must always be passed on the stack.

Because they occupy even-odd register pairs, the presence of 64-bit arguments can alter the strict left-to-right allocation order of arguments to D-registers. If the order of non-pointer arguments is, for example, WORD1, DOUBLE1, WORD2, then the allocations will be D[4],D[6]/D[7] (E[6]), and D[5].

2.2.3.3 Pointer Arguments

Up to four pointer arguments are passed in registers A[4] through A[7]. Pointer arguments include the base address for arrays passed as arguments and for structure arguments whose size is greater than 64-bits.

Argument registers are generally allocated to arguments in numeric order, left to right, according to argument type. A[4] would be allocated to the first pointer argument.

2.2.3.4 64-bit Pointer Arguments (Circular Buffer Pointers)

Up to two 64-bit pointer arguments are passed in the register pairs A[4]/A[5] and A[6]/A[7]. 64-bit arguments must not be split when there are too few argument registers to hold the entire argument.

Because they occupy even-odd register pairs, the presence of 64-bit pointer arguments can alter the strict left-to-right allocation order of arguments to address registers, in the same way described for non-pointer arguments.

2.2.3.5 Overflow Arguments on the Stack

There is a region of stack space, at the bottom of the caller's stack frame and the top of the callee's stack frame, that serves as an overflow area for those arguments of the call that do not fit into registers (see **Stack Frame Layouts for Three Calls, page 2-12**).

The first overflow argument is mapped to the lowest word (or double-word), and subsequent arguments are mapped to successive words or double-words. No gaps are needed for alignment of double-word arguments, as the architecture allows double-word access on word boundaries with no penalty.

If no arguments are passed on the stack, the overflow area is empty (size of zero).

2.2.3.6 Structure Arguments

Structures and unions whose size is less than or equal to 64-bits are passed in a Data register, or Data register pair, regardless of the types of their individual fields. This holds even if the structure or union contains only addresses. Structures and unions are never split between one or more registers and the stack. Structures and unions whose size is greater than 64-bits are passed via pointer.

The pointer will point directly to the structure or union being passed. The caller does not need to make a copy of the structure. The callee is entirely responsible for copying the structure if necessary.

2.2.4 Variable Arguments

ANSI C requires that when calling a varargs function, a prototype must be provided. By convention, non-ANSI (legacy) C code compliant with the TriCore EABI must also provide prototypes for all varargs functions called. The caller therefore knows the number and types of the fixed arguments, and which arguments are variable.

The fixed arguments are passed using the same register conventions as a regular call. All variable arguments are passed on the stack. Variable arguments on the stack begin immediately after the location of the last fixed argument on the stack, or at SP offset zero if no fixed arguments are passed on the stack.

2.2.5 Return Values

2.2.5.1 Scalar Return Values

32-bit return values, other than pointers, are returned in D[2]. This includes all types whose size in memory is less than 32-bits; they are expanded to 32-bits through zero-extension or sign-extension, according to the type.

64-bit scalar return values are returned in E[2] (register pair D[2]/D[3]).

2.2.5.2 Pointer Return Values

32-bit pointer return values are returned in A[2].

64-bit pointer return values such as circular buffer pointers are returned in the register pair A[2]/A[3].

2.2.5.3 Structure Return Values

Structure return values smaller than 32-bits are returned in D[2] regardless of their field types. Return values up to 64-bits are returned in the register pair D[2]/D[3] (E[2]) regardless of their field types. This holds true even if all fields are addresses.

Functions returning structures or unions larger than 64-bits have an implicit first parameter, which is the address of the caller-allocated storage area for the return value. This first parameter is passed in A[4]. The normal pointer arguments then start in register A[5] instead of in A[4].

The caller must provide for a buffer of sufficient size. The buffer is typically allocated on the stack to provide re-entrancy and to avoid any race conditions where a static buffer may be overwritten.

If a function result is the right-hand side of a structure assignment, the address passed may be that of the left-hand side, provided that it is not a global object that the called function might access through an alias. The called function does not buffer its writes to the return structure. (i.e. it does not write to a local temporary and perform a copy to the return structure just prior to returning).

The caller must provide this buffer for large structures even when the caller does not use the return value (for example, the function was called to achieve a side-effect). The called routine can therefore assume that the buffer pointer is valid and need not check the pointer value passed in A[4].

2.3 Alternative Function Calling Conventions (Stack-Model)

In order to support legacy code lacking some function prototypes, complying compilers must support an alternative calling model passing all parameters on the stack. The D/A-model described above shall be the default calling model.

2.3.1 Stack Model Directive

Compilers must provide a source code directive which triggers the use of the stack model. When the directive is associated with a function definition, the function must be compiled in the stack model. When the directive is associated with a function declaration all calls to that function must be translated to the stack model (See also **Runtime Model Checking, page 2-20** for error detection).

The stack model directive must be applicable to function pointers and function pointer typedefs.

2.3.2 Register Assignments

The register assignments are as described in **Register Assignments, page 2-10**.

2.3.3 Stack Frame Layout

The stack frame layout is the same as described in **Stack Frame Management, page 2-11**. However, the “argument overflow” area at the top of the frame will generally be larger, to accommodate the larger number of arguments passed on the stack (See below).

2.3.4 Argument Passing

All arguments occupying up to 64-bits are passed on the stack at the bottom (low address end) of the caller's stack frame. On entry to a called routine, the first of such arguments is at the lowest word or double-word (offset zero from the stack pointer). Subsequent arguments occupy successive higher words or double-words. No gaps are needed for alignment of double-word arguments, as the architecture allows double-word access on word boundaries with no penalty.

2.3.4.1 Structure Arguments

Structures and unions larger than 64-bits are always passed via pointer. The copying rules as described in **Structure Arguments, page 2-15**, hold true.

2.3.4.2 Return Values

Return values of up to 32-bits are returned in register D[2]. Return values of up to 64-bits are returned in the register pair D[2]/D[3] (E[2]).

Functions returning structures or unions larger than 64-bits have an implicit parameter, which is the address of the caller-allocated storage area for the return value. This address is passed to the callee in register A[4].

The caller must provide for a buffer of sufficient size. The buffer is typically allocated on the stack to provide re-entrancy and to avoid any race conditions where a static buffer may be overwritten.

If a function result is the right-hand side of a structure assignment, the address passed may be that of the left-hand side, provided that it is not a global object that the called function might access through an alias. The called function does not buffer its writes to the return structure (i.e. it does not write to a local temporary, and perform a copy to the return structure just prior to returning).

The caller must provide this buffer for large structures even when the caller does not use the return value (For example, the function was called to achieve a side-effect). The called routine can therefore assume that the buffer pointer is valid and need not check the pointer value passed in A[4].

2.4 Support for Mixed Models

Lack of prototypes and the use of different models on the calling and called side may result in some hard-to-find bugs. In order to give the best support possible, complying compilers and linkers must implement the following two mechanisms:

- **Link Time Argument and Return Type Checking, page 2-18**.
- **Runtime Model Checking, page 2-20**.

2.4.1 Link Time Argument and Return Type Checking

For direct function calls, and for assignment of function addresses to prototyped function pointers, the linker can do the type checking. Compilers must support type checking by emitting extra symbol information as described below.

2.4.1.1 Link Time Type Information

For every direct function call in a source file, and for every assignment of a function address to a function pointer having a specific prototype, the compiler produces an ELF symbol of the following convention, in the resulting ELF relocatable file:

```
__caller.<name>.<model>.<return_type>.<parameter_types>
```

For every function definition, a symbol of the following convention is required:

```
__callee.<name>.<model>.<return_type>.<parameter_types>
```

The symbol value shall be the same as the value of the original symbol for the function being described. In order to avoid type conflicts, compilers must not create a caller symbol for the main function.

The definitions for the above symbol name fields are described in **Table 4**:

Table 4 Symbol Name Definitions for Link Time Type Checking

Segment	Value	Description
<name>		Denotes the name of the function.
<model>	DA | S	DA = D/A-model S = stack model
<return_type>	<basetype>	Denotes the return type of the function.
<parameter_types>	<basetype> [<basetype>,...]	Denotes the parameter types of the function.

The possible values for <basetype> are listed in **Table 5**.

Table 5 Basic Type Definitions

Value	Description
i	Scalar type of size <= 32-bits (for example; char, short, int).
l	Scalar type of size 64-bits (for example; long long).
p	Pointer type of size 32-bits (for example; void *, int *, struct foo *, struct foo **).
p2	Pointer type of size 64-bits (circular buffer pointer).
f	Single precision float type.
d	Double precision float type.
s<num>	Struct type passed in <num> = 0, 1, 2 registers or stack words (num = 0 <==> struct is passed by reference).
v	Void type.
e	Start of an ellipsis.

2.4.1.2 Type Checking in the Linker

Linkers are required to perform type checking on the basis of the special symbols emitted by compilers.

It is an error if:

- Both the caller and the callee symbols exist but their model descriptions do not match.
- Both the caller and the callee symbols exist and specify the D/A-model, but the type descriptions do not match exactly. There is one exception to this rule: the caller return type ‘v’ matches any callee return type.

The linker may produce a warning if:

- Both the caller and the callee symbols exist and specify the stack model, but the type descriptions do not match exactly.
- A caller symbol exists without a corresponding callee symbol.

Linkers should never eliminate the special symbols in an incremental linkage step. If a linker/locator eliminates the special symbols during final linkage, it must provide a switch to turn this behaviour off. This enables other tools to exploit the information, such as incremental loaders.

2.4.2 Runtime Model Checking

As noted in **Stack Model Directive, page 2-16**, the directive that specifies use of the stack model for argument passing must be applicable to function pointers and function pointer typedefs, as well as named functions.

When a function is called indirectly through a prototyped function pointer whose value is not the result of a cast operation, a match of the argument model and argument types is guaranteed through the same link time checking mechanism that guarantees matches for direct function calls. However, if the pointer has no prototype, or if its value resulted directly or indirectly from a pointer cast operation, a match cannot be guaranteed.

In order to enable software to determine at runtime, the argument model associated with a function pointer, the following technique is mandated:

- when the address of a function is taken, and the function is one that employs the stack model for argument passing, then the address value returned shall be offset by one byte from the value of the actual symbol for the function.

Since all functions are aligned on at least a half-word boundary, offsetting the address by one byte will result in an odd address value (bit 0 = ‘1’). The TriCore architecture ignores bit 0 for indirect calls and jumps, but the bit can be tested by software. If, at a given point in a program where an indirect call is made, the user expects (but can not guarantee) that a particular argument passing model will always be called for, an ASSERT macro testing the low order bit of the pointer value can be inserted ahead of the call, to detect errors at runtime. Alternatively, if it is legitimate for functions of either model to be called at that point, an ’if…else’ test on the bit can be used to select the appropriate calling method. The call would be duplicated in the ’if’ and ’else’ parts, with the function pointer cast, to a stack model pointer in one instance and and to a D/A model pointer in the other instance. An implementation might choose to define a standard macro, named along the lines of “CHECKED_INDIRECT_CALL”, to make this type of calling more convenient for users. However that is not an EABI issue.

2.5 Memory Models

2.5.1 Data Memory Model

Small Data

Compilers and linkers/locators must support a small data section. The small data section occupies a memory segment of up to 64 KBytes. Compilers must ensure that the startup code loads the A[0] register with a pointer into the small data section. The entire small data section must be addressable with a 16-bit offset relative to A[0]. A[0] must never be reloaded after program startup. The small data section is supported with appropriate relocation information (see **Relocation Values, page 4-8**).

Absolute Data

Compilers may create, and linkers/locators must support, absolute data sections. An absolute data section may contain up to 16 KBytes of data which can be addressed with 18-bit absolute addressing and must therefore be located within the first 16 KBytes of a 256 MBytes memory segment. Compilers that create absolute data sections must provide a means to suppress the generation of such data. Libraries may not contain absolute data.

Literal Data

Compilers may employ, and linkers/locators must support, a literal data section. As with the small data section, the literal data section occupies a memory region of up to 64 KBytes. It is a read-only data section containing address literals and other program constants. The C runtime library must include a startup code module that can be linked with an application, and that will initialize register A[1] with a pointer into the literal data section. After startup, the value in A[1] must remain constant over the life of the linked application. The entire literal data section must be addressable with a 16-bit offset relative to A[1].

Use of the literal data section is a recommendation, not a requirement. Compilers may place literals in other read-only data sections, or even the text section, as long as they generate correct code to access the literals. However, linkers and locators for conforming implementations are required to support linking of library modules compiled for use of the literal data section, whether or not their associated compilers make use of the section. The specific support requirements are detailed in **Relocation Values, page 4-8**.

2.5.2 Code Memory Model

The TriCore architecture provides three alternative addressing modes for calls and unconditional jumps: PC relative, absolute, and register indirect. PC relative addressing provides a 24-bit, half-word, scaled relative offset, supporting a target address span of +/-16 MBytes around the calling point. This is almost always sufficient for calls to functions located within the same memory segment, but is not sufficient for calls across physical memory segments. Absolute addressing provides a two-component absolute address, with four bits of target segment ID and twenty bits of half-word scaled offset within that segment. It can therefore reach targets within any memory segment, but only within the first two MBytes of each segment.

Register indirect addressing provides a full 32-bit target address, held in a register specified in the instruction. Any target address is reachable with this mode, and its use presents no special complications for binary compatibility. However the instructions required to load the register with the target address represent added overhead that is seldom necessary, in practice.

In order to allow compilers to generate efficient calls to external functions whose final segment and offset are unknown at compile time, a special relocation mode is defined. The model for its operation is as follows:

1. The compiler issues a CALL, J, or JL, to the external symbol. These instructions imply PC relative addressing.
2. If the branch address resolves to a location within the +/-16 MBytes span of the CALL, J, or JL instruction, the locator resolves it normally.
3. Otherwise, if the branch address lies within the first 2 MBytes of the segment to which it is mapped, the locator changes the opcode bits that specify the addressing mode of the instruction, changing the mode to absolute, and resolves the branch address using that mode.
4. If neither of the above two conditions holds, the action taken by the locator is implementation dependent.

The recommended action in the final case, is for the locator to redirect the CALL, J, or JL instruction to a labelled location within the first 2 MBytes of the segment containing the actual target address. At the location within the first 2 MBytes of the segment, the locator will place either a 32-bit PC-relative jump to the actual target address, or a 2.5-word sequence to load register A[12] with the actual target address, and jump to it with register indirect addressing. The choice of jumptable entry formats will depend on the offset from the jumptable entry to the jump target address. If it is less than 16 MBytes, a 32-bit PC-relative jump can be used. Otherwise the 80-bit register-indirect sequence must be used.

The following procedure can be used to build the jump table:

- Build a dictionary of function entry symbols located in the segment, with values defined as offsets from the end of the jump table that will be located at the start of the segment. The dictionary should be ordered by offset value.
- Build the actual jump table, working from last entry to first. The offset to a symbol is its dictionary offset, plus the running size of jump table entries build up to the point of the current entry referencing that symbol. If that offset is greater than 16 MBytes, use an indirect jump sequence. Otherwise, use a PC-relative jump.
- Halt when the offset becomes less than 2 MBytes. Any entry symbol whose offset is less than 2 MBytes is directly reachable via absolute addressing, and does not require a jump table entry.

The designated use of A[12] for indirect jump sequences is transparent to compiled code using the standard calling mechanism. The sequence follows a CALL that has already saved the caller's A[12] value before the sequence is executed; the contents of A[12] (and all other upper context registers other than A[10](SP) and A[11](RA)), are architecturally undefined after a CALL, and the registers can not be used to pass arguments to a called function. However if assembly language programmers call external functions using the JL mechanism, they need to be aware of the possibility that A[12] will be modified.

3 High-level Language Issues

3.1 C Name Mapping

Externally visible names in the C language must be mapped through to assembly without any change.

For example:

```
void testfunc() { return;}
```

This generates assembly code similar to the following fragment:

```
testfunc:
    RET
```

Additionally, the symbols described in **Link Time Argument and Return Type Checking, page 2-18**, are created.

4 Object File Formats

TriCore tools use ELF 2.0 (or higher) object file formats. ELF provides a suitable basis for representing the information needed for embedded applications.

This section describes particular fields in the ELF format that differ from the base standards for those formats.

4.1 Header Conventions

4.1.1 E_MACHINE

The e_machine member of the ELF header contains the decimal value 44 (hexadecimal $2C_H$) which is defined as the name EM_TRICORE.

4.1.2 E_IDENT

The TriCore-specific contents of the e_ident fields are specified in the following table:

Table 6 e_ident Field Values

Field	Value	Description
e_ident[EI_CLASS]	ELFCLASS32	Identifies 32 bit architecture. Mandatory for all 32-bit implementations.
e_ident[EI_DATA]	ELFDATA2LSB	Identifies 2's complement, little-endian data encoding. Mandatory for all implementations.

4.1.3 E_FLAGS

The e_flags field of the ELF header record is used to identify one or more specific core derivatives of the TriCore architecture for which generated code is present in the object file. It also identifies the Peripheral Control Processor (PCP), when the object file includes output from the PCP assembler.

The following e_flags values are defined for identifying the presence of code for the different architecture derivatives of the core and the PCP:

Table 7 e_flags Identifying TriCore/PCP Derivatives

Name	Value
EF_TRICORE_V1_1	$8000\ 0000_H$
EF_TRICORE_V1_2	$4000\ 0000_H$
EF_TRICORE_V1_3	$2000\ 0000_H$

Table 7 e_flags Identifying TriCore/PCP Derivatives (Continued)

Name	Value
EF_TRICORE_PCP	$0100\ 0000_H$
EF_TRICORE_PCP2	$0200\ 0000_H$

As **Table 7** shows, there are currently three different core derivatives of the TriCore architecture that tools may support (TriCore 1 V1.1, TriCore 1 V1.2, TriCore 1 V1.3).

TriCore 1 V1.1 was a preliminary version, implmented on the “Rider A” test chip. The instruction opcodes were subsequently remapped, making object code for the V1.1 core incompatible with that of the V1.2 and later cores. The linker must prevent object modules compiled for the V1.1 core from being linked with those of later core derivatives.

TriCore 1 V1.3 adds an MMU and related instructions to the TriCore 1 V1.2 architecture. It is upward compatible with TriCore 1 V1.2, and it is anticipated that any future core derivatives will also be upward compatible. New e_flags values will be assigned to any such cores as they may be developed.

There are also currently two versions of the PCP; the original PCP, and a new revision, designated PCP2. It is a superset of the original PCP.

If object files with different e_flags fields are combined together, the e_flags field in the resulting file must be set to the value corresponding to the highest version present in the input files. For example, if object files for TriCore 1 V1.2 and for TriCore 1 V1.3 are linked together into an executable file then the version number for the output file must be set to TriCore 1 V1.3

Because these flag values were not defined in early versions of this document, use of the specific values described above is not a mandatory EABI conformance requirement. However these values should be used in any future development. It is recommended that vendors currently using other values supply a small utility that customers can use to convert older e_flags values to the standards defined here.

4.2 Section Layout

4.2.1 Section Alignment

The object generator (compiler or assembler) provides alignment information for the linker. The default alignment is 2 Bytes. Object producers must ensure that generated objects specify required alignment.

When a tool merges sections, it must apply the strictest alignment attribute of any of its components to the resulting section.

4.2.2 Section Attributes

Table 8 defines section attributes that are available for TriCore tools. These attributes are additions to the ELF standard flags shown in Table 9.

Table 8 TriCore Section Attribute Flags

Name	Value
SHF_TRICORE_ABS	400_H
SHF_TRICORE_NOREAD	800_H

The SHF_TRICORE_NOREAD attribute allows the specification of code that is executable but not readable. Plain ELF assumes that all segments have read attributes, which is why there is no read permission attribute in the ELF attribute list. In embedded applications, “execute-only” sections that allow hiding the implementation are often desirable.

The SHF_TRICORE_ABS attribute allows the specification of absolute sections. Absolute sections cannot themselves be relocated, although they can contain references to external symbols that may be relocatable.

Table 9 ELF Section Attributes

Name	Value
SHF_WRITE	$0000\ 0001_H$
SHF_ALLOC	$0000\ 0002_H$
SHF_EXECINSTR	$0000\ 0004_H$

Please refer to the ELF specification for a description of the standard section attributes.

4.2.3 Special Sections

Various sections hold program and control information. The sections listed in **Table 10** have special meaning which must be supported by TriCore linker and locator.

Table 10 TriCore Special Section Names

Name	Type	Attributes	Description
.ldata	SHT_PROGBITS	SHF_ALLOC	Read-only address constants and other literal data.
.sbss	SHT_NOBITS	SHF_ALLOC+SHF_WRITE	Uninitialized data which goes into the small data section. The section must be initialized with zeros at startup.
.sdata	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE	Initialized data which goes into the small data section.
.zbss	SHT_NOBITS	SHF_ALLOC+SHF_WRITE	Uninitialized data which goes into a memory region reachable with absolute data addressing. The section must be initialized with zeros at program startup.
.zdata	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE	Initialized data which goes into a memory region reachable with absolute data addressing. The size of each absolute data section is limited to 16 KBytes.

Note: The combined size of all .sbss and .sdata sections of an application cannot exceed 64 KBytes.

The section names pre-defined by ELF are shown in **Table 11**. Please refer to the ELF specification for a description of each section.

Table 11 ELF Reserved Section Names

Name	Type	Attributes
.bss	SHT_NOBITS	SHF_ALLOC+SHF_WRITE
.bss_a8	SHT_NOBITS	SHF_ALLOC+SHF_WRITE
.bss_a9	SHT_NOBITS	SHF_ALLOC+SHF_WRITE
.comment	SHT_PROGBITS	none

Table 11 ELF Reserved Section Names (Continued)

Name	Type	Attributes
.data	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.data_a8	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.data_a9	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.debug	SHT_PROGBITS	none
.dynamic	SHT_DYNAMIC	--
.dynstr	SHT_STRTAB	SHF_ALLOC
.dynsym	SHT_DYNSYM	SHF_ALLOC
.fini	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR
.hash	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR
.interp	SHT_PROGBITS	--
.line	SHT_PROGBITS	none
.note	SHT_NOTE	none
.pcpdata	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.pcptext	SHT_ PROGBITS	SHF_ALLOC+SHF_EXECINSTR
.rel*	SHT_REL	--
.rela*	SHT_RELA	--
.rodata	SHT_PROGBITS	SHF_ALLOC
.rodata_a8	SHT_PROGBITS	SHF_ALLOC
.rodata_a9	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	none
.srodata	SHT_PROGBITS	SHF_ALLOC
.strtab	SHT_STRTAB	--
.symtab	SHT_SYMTAB	--
.text	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR
.zrodata	SHT_PROGBITS	SHF_ALLOC

Note: It is recommended that read-only constants, such as string literals, be placed into a read-only data section instead of the .text section. For processor configurations with separate on-chip code and data memories, there are several cycles of added overhead to access a data item from the .text section. For configurations with unified code and data memories, there is no penalty for access to data items in the .text section, but neither is there any advantage.

4.3 Symbol Table Format

The small data section and the literal data section are supported by two special symbols:

```
extern Elf32_Addr    _SMALL_DATA_[];
extern Elf32_Addr    _LITERAL_DATA_[];
```

These symbols are used to load the registers A[0] and A[1] respectively, in the startup code. Data items in the small data section are relocated relative to the contents of A[0], while those in the literal data section are relocated relative to A[1]. Locators must resolve these symbols to locations that lie within no more than 32 KBytes above the start of their respective sections, and no more than 32 KBytes below the ends. A value equal to the start of the section plus 8000_H is guaranteed to meet these constraints, so long as the sections themselves are within their maximum allowed sizes of 64 KBytes each.

4.4 Relocation Information Format

4.4.1 Relocatable Fields

Table 12 describes the TriCore relocatable field types. The names of the field types in capital letters are taken from the corresponding instruction formats. However, more than one instruction format may correspond to a field type.

Note: In the following table; RV = Relocation Value. IW = Instruction Word.

Table 12 Relocation Types

Type	Description
word32	A 32-bit field occupying four bytes. This address is NOT required to be 4-byte aligned.
word16	A 16-bit field occupying two bytes.
relB	A 32-bit instruction word, where: • bits 1-16 of the RV go into bits 16-31 of the IW. • bits 17-24 of the RV go into bits 8-15 of the IW. • the RV must be in the range [-16777216,16777214]. bit 0 of the RV must be zero.

Table 12 Relocation Types (Continued)

Type	Description
absB	A 32-bit instruction word, where: • bits 1-16 of the RV go into bits 16-31 of the IW. • bits 17-20 of the RV go into bits 8-11 of the IW. • bits 0 and 21 to 27 of the RV must be zero. • bits 28-31 of the RV go into bits 12-15 of the IW.
BO	A 32-bit instruction word where: • bits 0-5 of the RV go into bits 16-21 of the IW. • bits 6-9 of the RV go into bits 28-31 of the IW. • bits 10-31 of the RV must be zero.
BOL	A 32-bit instruction word, where: • bits 0-5 of the RV go into bits 16-21 of the IW. • bits 6-9 of the RV go into bits 28-31 of the IW. • bits 10-15 of the RV go into bits 22-27 of the IW. • bits 16-31 of the RV must be zero.
BR	A 32-bit instruction word, where: • bits 1-15 of the RV go into bits 16-30 of the IW. • bits 16-31 of the RV must be zero.
RLC	A 32-bit instruction word, where: • bits 0-15 of the RV go into bits 12-27 of the IW. • bits 16-31 of the RV must be zero.
ABS	A 32-bit instruction word, where: • bits 0-5 of the RV go into bits 16-21 of the IW. • bits 6-9 of the RV go into bits 28-31 of the IW. • bits 10-13 of the RV go into bits 22-25 of the IW. • bits 14-27 of the RV must be zero. • bits 28-31 of the RV go into bits 12-15 of the IW.
SBR	A 32-bit instruction word, where: • bits 0-3 of the RV go into bits 8-11 of the IW. • bits 4-32 of the RV must be zero.
pcpPage	A 16-bit instruction word, where: • bits 8-15 of the RV go into bits 8-15 of the IW. • bits 0-7 and 16-31 of the RV must be zero.
PI	A 16-bit instruction word, where: • bits 0-5 of the RV go into bits 0-5 of the IW. • bits 6-15 of the RV must be zero.

4.4.2 Relocation Values

This section describes values and algorithms used for relocations. In particular it describes values the compiler/assembler must leave in place and how the linker modifies those values.

All TriCore relocations are .rela relocations. The target bits of a relocation operation should be zero. They will be overwritten regardless of their contents.

Table 13 shows the semantics of relocation operations. In the following table:

- Key S indicates the final value assigned to the symbol referenced in the relocation record.
- Key A is the addend value specified in the relocation record.
- Key P indicates the address of the relocation (for example, the address being modified).
- Key A[0] is the content of the small data base register A[0].

Table 13 Relocation Type Encodings

Name	Value	Field	Calculation
R_TRICORE_NONE	0	none	none
R_TRICORE_32REL	1	word32	S + A - P
R_TRICORE_32ABS	2	word32	S + A
R_TRICORE_24REL	3	relB	S + A - P
R_TRICORE_24ABS	4	absB	S + A
R_TRICORE_16SM	5	BOL	S + A - A[0]
R_TRICORE_HI	6	RLC	S + A + 8000_H >> 16
R_TRICORE_LO	7	RLC	S + A & $FFFF_H$
R_TRICORE_LO2	8	BOL	S + A & $FFFF_H$
R_TRICORE_18ABS	9	ABS	S + A
R_TRICORE_10SM	10	BO	S + A - A[0]
R_TRICORE_15REL	11	BR	S + A - P
R_TRICORE_10LI	12	BO	S + A - A[1]
R_TRICORE_16LI	13	BOL	S + A - A[1]
R_TRICORE_10A8	14	BO	S + A - A[8]
R_TRICORE_16A8	15	BOL	S + A - A[8]
R_TRICORE_10A9	16	BO	S + A - A[9]
R_TRICORE_16A9	17	BOL	S + A - A[9]
R_TRICORE_PCPHI	25	word16	S + A >> 16

Table 13 Relocation Type Encodings (Continued)

Name	Value	Field	Calculation
R_TRICORE_PCPLO	26	word16	S + A & $FFFF_H$
R_TRICORE_PCPPAGE	27	pcpPage	S + A & $FF00_H$
R_TRICORE_PCPOFF	28	PI	(S + A >> 2) & $3F_H$
R_TRICORE_PCPTEXT	29	word16	(S + A >> 1) & $FFFF_H$

4.5 Debugging Information Format

TriCore tools must support DWARF 2.0 debugging information formats.

4.5.1 DWARF Register Numbers

DWARF represents register names efficiently as small integers. These numbers are used in the OP_REG and OP_BASEREG atoms to locate values. The mapping of DWARF register numbers to the Tricore register set is shown in **Table 14**.

Table 14 DWARF Register Mapping for TriCore

Atom	Register	Atom	Register	Atom	Register	Atom	Register
0	D[0]	16	A[0]	32	E[0]	48	BIV
1	D[1]	17	A[1]	33	E[2]	49	BTV
2	D[2]	18	A[2]	34	E[4]		
3	D[3]	19	A[3]	35	E[6]		
4	D[4]	20	A[4]	36	E[8]		
5	D[5]	21	A[5]	37	E[10]		
6	D[6]	22	A[6]	38	E[12]		
7	D[7]	23	A[7]	39	E[14]		
8	D[8]	24	A[8]	40	PSW		
9	D[9]	25	A[9]	41	PCXI		
10	D[10]	26	A[10]	42	PC		
11	D[11]	27	A[11]	43	FCX		
12	D[12]	28	A[12]	44	LCX		
13	D[13]	29	A[13]	45	ISP		
14	D[14]	30	A[14]	46	ICR		
15	D[15]	31	A[15]	47	PIPN		

5 Extensions for Shared Object Support

Please contact your Infineon Sales Office to request a description of the optional extensions to the EABI for shared object support.

Index

http://www.infineon.com

Published by Infineon Technologies AG