C7000 Embedded Application Binary Interface (EABI)

Reference Guide

Literature Number: SPRUIG4C
January 2018–Revised August 2019

Contents

Read This First

About This Manual

This document specifies the ELF-based Application Binary Interface (ABI) for the C7000™ family of processors from Texas Instruments. The ABI is a broad standard that specifies the low-level interface between tools, programs and program components.

This document does not cover the following features, because they have not been implemented:

- Dynamic linking
- Thread local storage
- Symbol versioning
- Position independent code
- Linux
- Global offset tables

Related TI Documentation

The following documents provide related information for the C7000:

- *C7000 C/C++ Optimizing Compiler User's Guide* (SPRUIG8)
- *C7000 Host Emulation User's Guide* (SPRUIG6)
- *C6000-to-C7000 Migration User's Guide* (SPRUIG5)
- *VCOP Kernel-C to C7000 Migration Tool User's Guide* (SPRUIG3)

Other Relevant Documentation

- ELF Specification System V Application Binary Interface
- IA-64 (Itanium) C++ ABI, v1.86
- IA-64 (Itanium) Exception Handling ABI, v1.22
- Application Binary Interface for the ARM Architecture, v2.08
- DWARF Debugging Format, v4
- C Language Standard, ISO/IEC 9899:1990
- C99 Language Standard, ISO/IEC 9899:1999
- C11 Language Standard, ISO/IEC 9899:2011
- C++14 Language Standard, ISO/IEC 14882:2014

Trademarks

C7000, C6000 are trademarks of Texas Instruments.
OpenCL is a trademark of Apple Inc. used with permission by Khronos.
Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.
Itanium is a registered trademark of Intel Corporation or its subsidiaries in the U.S. and/or other countries.
Linux is a registered trademark of Linus Torvalds in the U.S. and other countries.

Introduction

This chapter provides an introduction to the ABIs available for the C7000™, also known as C7x.

Topic **Page**

1.1 ABIs for the C7000

The C7000, which is also known as C7x, is a high-performance processor from Texas Instruments.

The C7000 inherits many attributes from the C6000™ family, whose ABI is specified in SPRAB89. However there are many important differences and the C7000 should be thought of a separate and distinct ISA. Object code from C6000 is not compatible with C7000.

The ABI for C7000 is based on the ELF object file format, and includes support for dynamic linking and position independence. It is derived from industry standard models, including the IA-64 C++ ABI and the System V ABI for ELF and Dynamic Linking.

The C7000 has a 64-bit address space for both code and data. Accordingly, object files use the ELF64 format. ELF64 differs from ELF32 primarily in that all addresses are represented as 64-bit fields.

A *platform* is the software environment upon which a program runs. The ABI has platform-specific aspects, particularly in the area of execution-time behavior such as dynamic linking and loading. Currently there are two supported platforms: *bare-metal* and *Linux*®. The term bare-metal represents the absence of any specific environment. That is not to say there cannot be an OS; it simply says that there are no OS-specific ABI specifications. In other words, how the program is loaded and run, and how it interacts with other parts of the system, is not covered by the bare-metal ABI.

- **The bare-metal ABI** allows substantial variability in many specific aspects. For example, part of the ABI specifies conventions for position independence (PIC), but if a given system does not require position independence, these conventions do not apply. Because of this variability, programs may still be ABI-conforming but incompatible; for example if one program uses PIC but the other doesn't, they cannot interoperate. Toolchains should endeavor to enforce such incompatibilities.
- **The Linux ABI** augments the bare-metal ABI by narrowing its variability and detailing additional requirements, so that a program or subprogram can run under a Linux-based OS on the C7000.

1.2 Scope

Figure 1-1 shows the components of the ABI and their relationship. We will briefly describe the components, beginning with the lower part of the diagram and moving upward, and provide references to the appropriate chapters of this ABI specification.

The components in the bottom area relate to object-level interoperability.

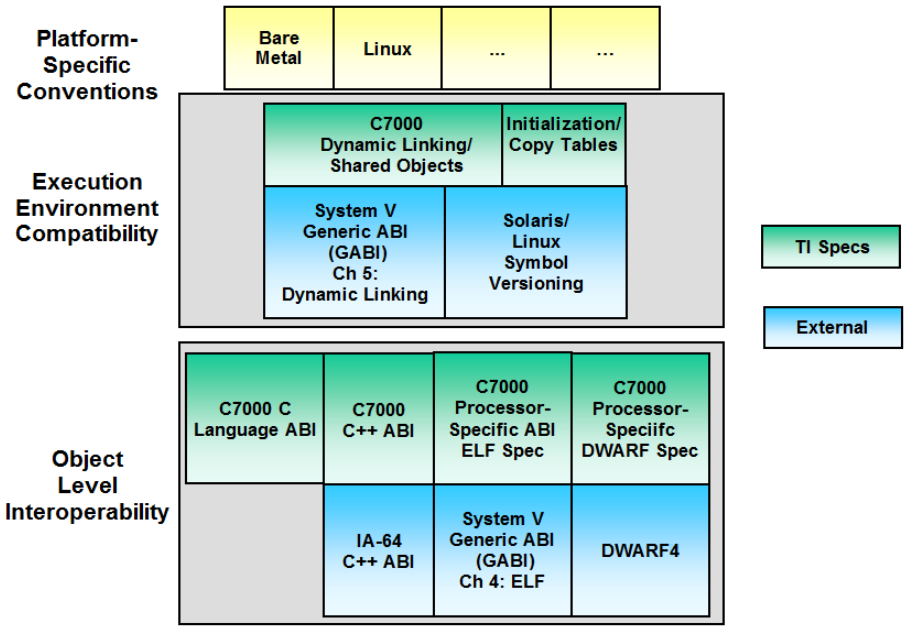


Figure 1-1. Parts of the ABI Specification

The *C Language ABI* (Chapter 2, Chapter 3, Chapter 4, Chapter 5, Chapter 6, and Chapter 7 in this document) specifies function calling conventions, data type representations, addressing conventions, and the interface to the C runtime library.

The *C++ ABI* (Chapter 8) specifies how the C++ language is implemented, such as virtual function tables, name mangling, how constructors are called, and the exception handling mechanism (Chapter 9). The C7000 C++ ABI is based on the commonly prevalent IA-64 (Itanium®) C++ ABI.

The *DWARF* component (Chapter 10) specifies the representation of object-level debug information. The base standard is the DWARF4 standard. This specification details processor-specific extensions.

The *ELF* component (Chapter 11) specifies the representation of object files. This specification extends the System V ABI specification with processor specific information.

Build Attributes (Chapter 12) refer to a means of encoding into an object file various parameters that affect inter-object compatibility, such as target device assumptions, memory models, or ABI variants. Toolchains can use build attributes to prevent incompatible object files from being combined or loaded.

Finally, there is a set of specifications that are not formally part of the ABI but are documented here both for reference and so that other toolchains can optionally implement them.

Initialization (Chapter 13) refers to the mechanism whereby initialized variables obtain their initial values. Nominally these variables reside in the .data section and they are initialized directly when the .data section is loaded, requiring no additional participation from the tools. However the TI toolchain supports a mechanism whereby the .data section is encoded into the object file in compressed form, and decompressed at startup time. This is a special use of a general mechanism that programmatically copies compressed code or data from offline storage (for example, ROM) to its execution address. We refer to this facility as copy tables. While not part of the ABI, the initialization and copy table mechanism is documented here so that other toolchains can support it if desired.

Program Header Attributes (Chapter 14) are an extension to ELF implemented by the TI toolchain in order to represent various additional properties of ELF segments beyond what is specified by the base ELF standard. The TI tools use them to encode memory connectivity/latency requirements, protection, cache behavior, and other system-specific properties. They are designed to be flexible and extensible. Again, we document them here so that other tools can interoperate with them if needed.

1.3 ABI Variants

As mentioned, the ABI does not define specific behavior in all instances but rather is a canon of principles that allow for platform or system-specific variation. For example, the ABI does not specify that PIC (position independent code) addressing will be used in all cases, but standardizes its implementation for those cases where it is used.

This section describes some common use cases and how they relate to the ABI. These cases are not mutually exclusive, nor do they completely cover all the possibilities.

- **Bare-metal - Standalone.** This model refers to a single self-contained statically-linked executable. It is the simplest form in terms of interoperability. The relevant parts of the ABI are the object-level components in the lower part of Figure 1-1. Since the executable is statically linked and bound (relocated), there is typically no need for position-independence. Since it is self-contained, it need not contain dynamic linking information, PLT stubs, or a GOT.
- **Shared Objects.** This refers to a dynamic linking model in which statically linked modules (libraries) can be shared among multiple separately-linked clients (executables or other libraries). The fundamental issue is that each client must have its own copy of the library's data. The C7000 ABI solves this through two related structures: position-independent addressing, and virtual addressing.

1.4 Toolchains and Interoperability

This ABI is not specific to any particular vendor's toolchain. In fact, its purpose is to enable alternative toolchains to exist and interoperate.

The ABI describes how mechanisms are implemented; not how toolchains support them at the user level. Occasionally references are made to the TI tools; these are for illustration only.

However, TI's C7000 Compiler Tools by nature have unique status since they originate from the silicon vendor and were co-developed with the ABI specification, and in some cases form its basis.

In cases where the behavior of the TI tools conflict with this ABI, it shall be considered a defect in the tools; if you find such a case please submit a defect report to support@tools.ti.com. However, in cases where this specification is incomplete or unclear, the behavior of the TI tools shall be considered definitive. A major goal of the ABI standard is interoperability with TI tools. Toolchain vendors should strive to meet this goal regardless of omissions or ambiguities in the standard itself. Please notify us in such cases and we will endeavor to clarify the specification.

1.5 Libraries

Generally a toolchain includes a linker as well as standard runtime libraries that implements part of the language support provided by the toolchain.

The library format used by the C7000 is the common GNU/SVR4 ar format.

Often the linker and libraries have interdependencies that are outside the realm of the ABI. For example, many linkers use special symbols to control the inclusion or exclusion of various library components; alternatively some libraries refer to special linker-defined symbols. For this reason the linker and library are expected to come from the same toolchain. Using a linker from one toolchain and a library from a different one is not supported under this ABI. This only applies to the built-in libraries that are part of the toolchain; application libraries built with a different toolchain can be linked.

1.6 Types of Object Files

ELF defines three distinct classes of object files:

- A *relocatable* file holds code and data suitable for static linking with other object files to create an executable or shared object file.
- An *executable* file holds a program suitable for execution. It may or may not have dynamic linking information.
- A *shared object* file is a constituent portion of a program that can be combined with an executable and other shared objects at load time to form a process image. Shared objects always contain dynamic linking information. To avoid confusion with relocatable object files we sometimes use the term *shared library* to refer to shared objects.

This specification uses the terms "static link unit" and "load module" interchangeably to refer to executables and shared libraries.

1.7 Segments

An ELF load module (an executable file or shared object) represents the memory image of the program in the form of segments. In this context a segment is a contiguous, indivisible range of memory with common properties. A segment becomes bound when its address is determined, which can either be statically at link time or dynamically at load time.

1.8 C7000 Architecture Overview

The C7000 (also known as C7x), is a family of VLIW Digital Signal Processors from Texas Instruments. Table 1-1 lists the members of the C7000 family covered by this ABI.

Table 1-1. C7000 ISAs

ISA	Description
C71x	Original ISA

C7000 devices are byte-addressable. Memory can be configured as big-endian or little-endian. All C7000 CPUs have an MMU that performs virtual-to-physical address translation. The addressing mechanisms described in this ABI assume the MMU is always enabled.

The C7000 has a 48-bit address space. Pointers are represented as 64-bit values.

The C7000 has two types of general purpose registers. Scalar registers are 64 bits wide. Vector registers are 512 bits wide. There are also predicate registers used exclusively for conditional execution of vector operations; these are 64 bits wide, with each bit controlling one byte (8 bits) of the associated vector operation.

1.9 Code Fragment Notation

Throughout this document we use code fragments to illustrate addressing, calling sequences, and so on. In the fragments, the following notational conventions are used:

- `sym` – a symbol being referenced
- `label` – a symbol referring to a code address
- `func` – a symbol referring to a function
- `tmp` – a temporary register (also temp, tmp1, tmp2, etc.)
- `reg` – an arbitrary register
- `dest` – the destination register for a resulting value or address

There are several assembler built-in operators introduced. These serve to generate appropriate relocations for various addressing constructs, and are generally self-evident.

For simplicity, code sequences are unscheduled. That is, each instruction is assumed to complete before commencing execution of the next instruction.

Data Representation

This chapter describes the representation in memory and registers of the standard C data types, plus other types supported by the architecture. Other languages may be supported but presumably their objects would correspond to these types.

The C7000 adopts the LP64 representational convention. That is, the type `int` is 32 bits, while long and pointer types are 64 bits.

In the descriptions and diagrams in this chapter, bit 0 always refers to the least-significant bit.

Topic **Page**

2.1 Basic Types

Integral values use twos-complement representation. Floating-point values are represented using IEEE 754.1 representation. Floating-point operations follow IEEE 754.1 to the degree supported by the hardware.

Table 2-1 gives the size and alignment of the basic C data types in bits. The generic names in the table are used in this specification to identify these types in a language-independent way.

Table 2-1. Data Sizes for Standard Types

C Type	Generic Name	Size	Alignment
char	int8_t	8	8
short	int16_t	16	16
int	int32_t	32	32
long	int64_t	64	64
long long	int64_t	64	64
bool	bool	8	8
float	float32_t	32	32
double	float64_t	64	64
long double	float64_t	64	64
pointer	void *	64	64
int40_t	int40_t	64	64

The type `char` is signed by default.

The integral types have complementary unsigned variants. The generic names are prefixed with ‘u’ (for example `uint_t`).

The type `bool` uses the value 0 to represent `false` and 1 to represent `true`. Other values are undefined.

The following additional types from C, C99, and C++ are defined as synonyms for standard types:

Table 2-2. C Language Standard Types

Type	Definition
size_t	unsigned long
ptrdiff_t	long
wchar_t	unsigned int
wint_t	int
va_list	char *

2.1.1 Basic Types in Memory

Alignment: The C7000 supports access to any memory object on any byte boundary; therefore there is no inherent alignment restriction imposed by the ISA. However, for software compatibility and to facilitate workable layout of structures in registers, the ABI specifies conventional alignment restrictions. For basic (scalar) types, the minimum alignment of an object is the size of its type. The minimum alignment for an object with array type is that specified by the type of its elements.

Alignment of structures, vectors, and complex types is covered in sections that follow.

Endianness: The C7000 can be configured to run in either *big endian* or *little endian* mode. Endianness refers to the layout of multi-byte values in memory. In big endian mode, the most significant byte of the value is stored at the smallest address. In little endian mode, the least significant byte is stored at the smallest address.

Endianness affects only objects’ memory representation; scalar values in registers always have the same representation regardless of endianness. Endianness does affect the layout of structures and bit-fields, which carries over into their register representation.

2.1.2 Basic Types in Registers

In general, implementations are free to use registers as they see fit. The ABI specifies register representations only for the purpose of standardizing how values are passed to or returned from functions.

Any objects whose size is 64 bits or less—including all scalars—are passed and returned in scalar registers. Scalar values in registers are always right justified; that is, bit 0 of the register contains the least significant bit of the value.

32-bit integral values are not sign or zero extended into bits 32-63 of the register. These bits may contain unspecified values.

Integral values smaller than 32 bits are sign or zero extended only to `int` size; that is, through bit 31.

Aggregate objects—including complex types, vectors, and structures—whose size is 64 bits or less are passed and returned in scalar registers. Aggregates whose size is 512 bits or less are passed and returned in vector registers. The register layout for such objects is specified in sections that follow.

2.2 Complex Types

The C7000 has various operations that operate on complex values, consisting of a real component and an imaginary component. The component types are either signed integer or floating point scalar types. The naming convention is the letter ‘c’ followed by the type of each component.

The ABI specifies the following complex types:

Table 2-3. Complex Types

Complex Type Name	Component Type	Native?	C99 type	Size	Alignment
cchar	char	yes		16	8
cshort	short	yes		32	16
cint	int	yes		64	32
cfloat	float	yes	float _Complex	64	32
clong	long	no	long _Complex	128	64
cdouble	double	no	double _Complex	128	64
NA	long double	no	long double _Complex	128	64

The types indicated as “native” are types for which there is direct instruction support.

2.2.1 Memory Representation of Complex Types

The C99 language includes complex types with floating-point components. Some of the C99 complex types map to the C7000 native complex types as shown in Table 2-3.

The C99 standard dictates the layout and alignment of its complex types to be equivalent to a two-element array of the corresponding floating-point type, with the real part as the first element and the imaginary part as the second. To enable a seamless mapping between the C99 types and the native types, the ABI specifies the same representation for the native complex types: they are stored in memory with the real component first, and have the same alignment as their component types.

2.2.2 Register Representation of Complex Types

The C7000 has direct ISA support for some complex types. These instructions require complex values to occupy an even-odd pair of SIMD lanes, with the imaginary part in the lower-numbered (even) lane and the real part in the next (odd) lane. [1] This is the *native complex layout*. When a complex value or vector of complex elements is passed to or returned from a function, the native layout is used.

Treating a complex value as separate elements and loading it using a normal 2-element vector load results in the incorrect ordering in the register. Therefore, to achieve the native layout, complex values are loaded and stored using custom instructions, with an element type that covers both components. For example, a cint, consisting of 32-bit real and imaginary components, is loaded with a double-word (64-bit) load. In big endian, the normal byte-swapping causes the two components to be swapped. In little-endian, a special form of the instruction is used that swaps them explicitly.

Figure 2-1 illustrates the memory and register layout of two example complex types.

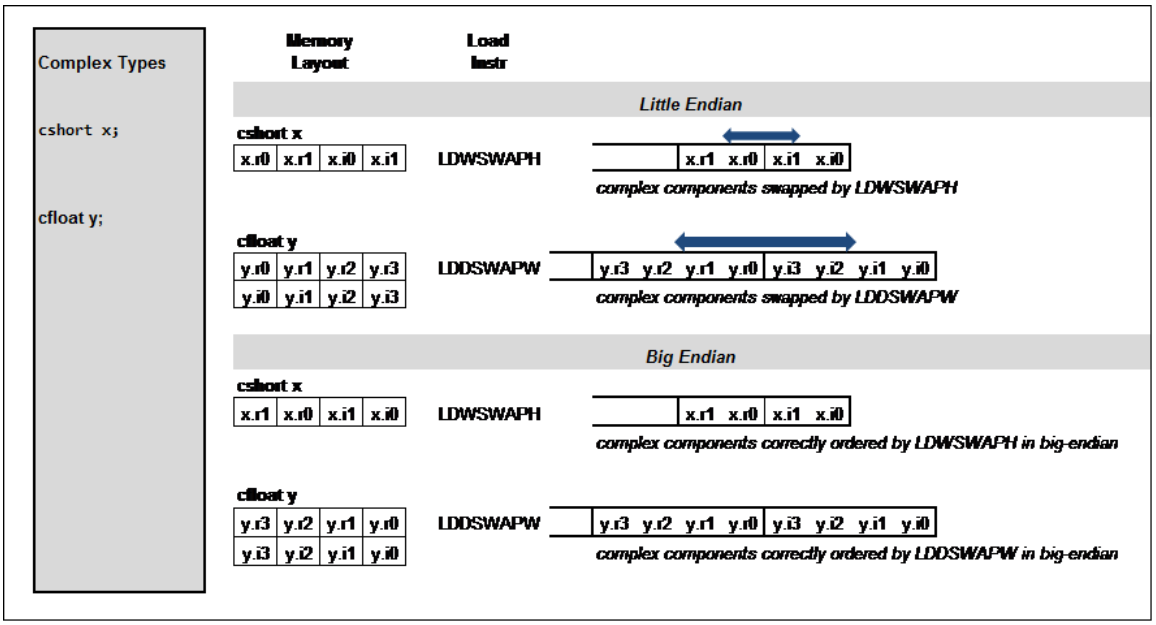


Figure 2-1. Register Layout for Complex Types

2.3 Vector Types

The C7000 has SIMD (Single-Instruction, Multiple Data) operations that operate on multiple values simultaneously. The programming model defines native vector types to express the operands of SIMD operations. Vector types are specified by the type of each element, and the number of elements. The element types are: 8, 16, 32, and 64-bit signed and unsigned integers, 32 and 64-bit floating point, and the native complex types. The number of elements is between 2 and the number of elements that can fit in a 512-bit vector register, and must be a power of two, with the exception of 3 element vectors. The naming convention for vector types derives from OpenCL™, and consists of the name of the type followed by the number of elements. (The C7000 extends the set of OpenCL vector types with complex element types and wider vectors.)

[1] The rationale for this layout is compatibility with legacy C6x instructions.

Table 2-4 lists the vector types specified by the ABI. The prefix (u) indicates unsigned variants. Not all the vector types in this table are supported directly by the C7000 ISA. However, the ABI specifies the representation to ensure consistency and compatibility among toolchains.

Table 2-4. Vector Types

Vector Type	Component Type	*N*	Element Size	Alignment
(u)char*N*	(u)char	2,3,4,8,16,32,64	8	8
(u)short*N*	(u)short	2,3,4,8,16,32	16	16
(u)int*N*	(u)int	2,3,4,8,16	32	32
(u)longlong*N*	(u)longlong	2,3,4,8	64	64
float*N*	float	2,3,4,8,16	32	32
double*N*	double	2,3,4,8	64	64
cchar*N*	cchar	2,3,4,8,16,32	16	8
cshort*N*	cshort	2,3,4,8,16	32	16
cint*N*	cint	2,3,4,8	64	32
cfloat*N*	cfloat	2,3,4,8	64	32
clong*N*	clong	2,3,4	128	64
cdouble*N*	cdouble	2,3,4	128	64

Vectors are stored in memory in with their elements in consecutive order starting with element 0. Each element is aligned according to its type. The alignment of the vector is equal to the alignment of its elements.

Vectors are stored in registers with element 0 occupying the LSBs of the register (lane 0) and successive elements occupying successive lanes. A vector with complex elements is laid out using the native complex layout as specified in Section 2.2.2, with successive elements occupying pairs of successive lanes. The vector layout described here also applies to vectors passed to and returned from functions in registers.

2.4 Structures and Unions

Structure members in memory are assigned offsets starting from 0 unless otherwise mandated by the C or C++ standards. Each member is assigned the lowest available offset that satisfies its alignment. Padding may be required between members to satisfy the alignment requirement.

Union members are all assigned offset 0.

The underlying representation of a C++ class is a structure. In this document, the term structure applies to classes and unions as well.

The alignment requirement of a structure is equal to the most strict alignment requirement among its members, including bit-field containers as described in Section 2.4.1. The size of a structure in memory is rounded up to a multiple of its alignment by inserting padding after the last member.

2.4.1 Structures in Registers

Some structures qualify to be passed to and returned from functions in registers, as described in Section 3.2. This section describes the conditions under which a structure qualifies, and how structures are represented in registers. Structures whose size is 64 bits or less are passed and returned in scalar registers.

Structures whose size is 512 bits or less are passed and returned in vector registers.

Moving structures between memory and registers necessitates the use of load and store instructions with arithmetic (not structure) type. Using an arithmetic type to access a structure in memory is a form of *type punning*, wherein an object in memory is accessed using a type different than its actual type, in effect treating the object as a blob of bits. We use the term *container type* to mean the arithmetic type used to load or store a structure. The layout of the structure in the register depends on the container type, due to endianness-dependent swapping. Therefore the ABI defines the register layout of a structure in terms of its container type.

The register layout dictated by the container type may or may not allow for efficient access to the fields in the structure. This is particularly true for structures larger than 64 bits, which must be accessed using vector instructions. Whether efficient access is possible depends on how well the vector lanes correspond to elements of the structure. The ABI dictates that structures whose size is 512 bits or less are passed to and returned from functions and what their layout is, but the compiler is free to move them to memory to access them in cases where that enables more efficient access. In practice, since the layout is defined in terms of 64-bit lanes, efficient access is possible as long as no field crosses a 64-bit boundary.

Register Layout of Structures

In little endian mode, vector (and scalar) loads preserve the memory layout. That is, the first byte occupies the LSB of the register and subsequent bytes of the structure are filled into the increasingly significant bytes of the register, regardless of the container type. So in little endian, the register layout matches the memory layout.

In big endian mode, however, bytes from memory are swapped in the register according to the container (arithmetic) type used for the load or store. If the type is a vector, this swapping occurs on an element-by-element basis.

The container type associated with a structure type is defined as follows:

1. If the structure contains a single nonstatic data member with scalar or vector type at offset 0, the container type is the type of that member.
2. If the structure contains a single nonstatic data member with structure (or class or union) type at offset 0, the container type is the container type of that member. (In other words, rule 1 applies recursively to trivially nested structures.)
3. Otherwise, the container type is the first of the following types that is at least as large as the structure:

Size of Struct Type	Container Type	Load Instruction	Store Instruction
0-8 bits	uchar	LDBU	STB
9-16 bits	ushort	LDHU	STH
17-32 bits	uint	LDWU	STW
33-64 bits	ulong	LDDU	STD
65-128 bits	ulong2	VLD2D	VST2D
129-256 bits	ulong4	VLD4D	VST4D
257-512 bits	ulong8	VLD8D	VST8D

The register layout of a structure passed to or returned from a function is defined as the layout that results from loading that structure from memory using a single (vector or scalar) load instruction with the associated container type.

Figure 2-2 illustrates the layout of two example structures in big and little endian modes. In this example, u0 refers to the least-significant byte of member u, and so on. In little-endian, the layout matches the memory layout, and is not affected by the container type. In big-endian, the container type affects the layout; vector instructions that load and store a structure to and from a vector register must have the proper type, or the layout will be incorrect.

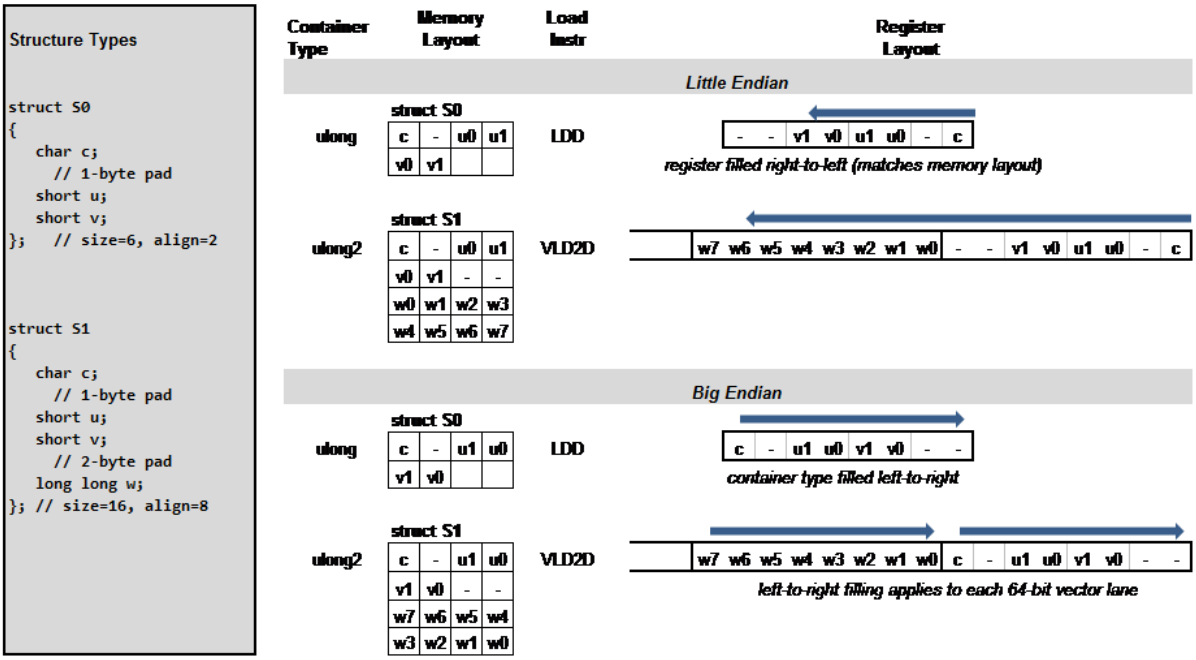


Figure 2-2. Layout for Structures in Registers

2.5 Bit-Fields

The C7000 EABI adopts its bit-field layout from the IA-64 C++ ABI. The following description is consistent with that standard unless explicitly indicated.

The *declared type* of a bit-field is the type that appears in the source code. To hold the value of a bit-field, the C and C++ standards allow an implementation to allocate any *addressable storage unit* large enough to hold it, which need not be related to the declared type. The addressable storage unit is commonly called the *container type*, and that is how we will refer to it in this document. The container type is the major determinant of how bit-fields are packed and aligned.

C89, C99, and C++ have different allowable declared types:

- **C89:** `int`, `unsigned int`, `signed int`
- **C99:** `int`, `unsigned int`, `signed int`, `_Bool`, or "some other implementation-defined type"
- **C++:** any integral or enumeration type, including `bool`

There is no `long long` type in strict C++, but because C99 has it, C++ compilers commonly support it as an extension. The C99 standard does not require an implementation to support `long` or `long long` declared types for bit-fields, but because C++ allows it, it is not uncommon for C compilers to support them as well.

A bit-field's value is fully contained within its container, exclusive of any padding bits. Containers are properly aligned for their type. The alignment of the object containing the field is affected by that of the container, in the same way as a member object of that type. This also applies to unnamed fields, which is a difference from the IA-64 C++ ABI. The container may contain other fields or objects, and may overlap with other containers, but the bits reserved for any one field, including padding for oversized fields, never overlap with those of another field.

In the C7000 EABI, the container type of a bit-field is its declared type, with one exception. C++ allows so-called oversized bit-fields, which have a declared size larger than the declared type. In this case the container is the largest integral type not larger than the declared size of the field.

The layout algorithm maintains a “next available bit” that is the starting point for allocating a bit-field. The steps in the layout algorithm are:

1. Determine the container type T, as above.
2. Let C be the properly-aligned container of type T that contains the next available bit. C may overlap previously allocated containers.
3. If the field can be allocated within C, starting at the next available bit, then do so.
4. If not, allocate a new container at the next properly aligned address and allocate the field into it.
5. Add the size of the bit-field (including padding for oversized fields) to determine the next available bit.

In little endian, containers are filled from LSB to MSB. In big endian, containers are filled from MSB to LSB.

Zero-length bit-fields force the alignment of the following member of the containing structure or union to the next alignment boundary corresponding to the declared type, and affect the alignment of the containing structure or union.

A declared type of plain `int` is treated as a `signed int` by the C7000 ABI.

2.5.1 Volatile Bit-Fields

When a volatile bit-field—that is, one declared with the C volatile keyword, is read, its container must be read exactly once using the appropriate access for the entire container. When a volatile bit-field is written, its container must be read exactly once and written exactly once using the appropriate access. The read and the write are not required to be atomic with respect to each other.

Multiple accesses to the same volatile bit-field, or to additional volatile bit-fields within the same container may not be merged. For example, an increment of a volatile bit-field must always be implemented as two reads and a write. These rules apply even when the width and alignment of the bit-field would allow more efficient access using a narrower type. For a write operation the read must occur even if the entire contents of the container will be replaced. If the containers of two volatile bit-fields overlap then access to one bit-field will cause an access to the other.

For example, given the structure

```
struct S
{
       volatile int  a:8;     // container is 32 bits at offset 0
       volatile char b:2;     // container is 8 bits at offset 8
};
```

An access to `a` will also cause an access to `b`, but not vice-versa. If the container of a non-volatile bit-field overlaps a volatile bit-field then it is undefined whether access to the non-volatile field will cause the volatile field to be accessed.

2.6 Enumeration Types

Enumeration types (ctype `enum`) are represented using an underlying integral type. Normally the underlying type is `int` or `unsigned int`, unless neither can represent all the enumerators, in which case the underlying type is `long long` or `unsigned long long`. When both the signed and unsigned versions can represent all the values, the ABI leaves the choice among the two alternatives to the implementation. (An application that requires consistency among different toolchains can ensure the choice of the signed alternative by declaring a negative enumerator.)

Calling Conventions

This chapter describes calling conventions for the C7000. This includes register conventions, return value handling, and passing by reference.

Topic **Page**

3.1 Call and Return

The C7000 has dedicated instructions and registers to manage call and return operations. The CALL instruction saves the return address in the Return Pointer (RP) register and transfers control to the called function. The RET instruction restores the PC from the RP, thereby returning control to the callee.

The control flow aspects of call and return are described in Section 5.2.

3.1.1 Pipeline Conventions

The CPU must be in the protected state when a call or return is made. The CPU must not be in unprotected mode.

3.1.2 Weak Functions

A weak function is a function whose symbol has binding of STB_WEAK. A program can successfully link without a definition for a weak function, leaving references to it unresolved. In that case, the linker replaces the call with a NOP instruction.

3.2 Register Conventions

The C7000 has two datapaths: an A-side datapath with 64-bit “scalar” registers, and a B-side datapath with 512-bit “vector” registers. The lower 64-bits of any B-side vector register can also be accessed as a scalar register by removing the ‘V’ from its name. Scalar registers are not limited to storing scalar values; a vector can be stored in a scalar register if it fits.

D15 is designated as the Stack Pointer (SP). The stack pointer must always remain aligned on a 2-word (8-byte) boundary. The SP points at the first aligned address below (less than) the currently allocated stack. Stack management and the local frame structure are presented in Section 4.4.

The ABI does not designate a dedicated Frame Pointer (FP) register since it does not affect interoperability between separately compiled functions. A compiler is free to employ any register in the role of a frame pointer as needed. It may be useful to have a FP in order to avoid using the constant extension when the local stack is large, to facilitate accessing arguments from the stack in a code size efficient manner.

The ABI designates RP, D15 (SP), A8-A15, B14/VB14, and B15/VB15 as *callee-save*. That is, a called function is required to preserve them so they have the same value on return from a function as they had at the point of the call.

All other registers are *caller-save*; that is, they are not preserved across a call, so if their value is needed following the call, the caller is responsible for saving and restoring their contents.

Table 3-1 lists the registers and their roles in the calling conventions.

Table 3-1. Register Conventions

Register	File	Preserved by Callee?	Role in Calling Convention
A0	A side scalar	no	
A1		no	Pointer to return-by-reference value
A2		no	
A3		no	
A4		no	1st scalar argument
A5		no	2nd scalar argument
A6		no	3rd scalar argument
A7		no	4th scalar argument
A8		yes	5th scalar argument
A9		yes	6th scalar argument
A10		yes	7th scalar argument
A11		yes	8th scalar argument
A12		yes	9th scalar argument
A13		yes	
A14		yes	
A15		yes	
AL0-AL7	A side local L	no	
AM0-AM7	A side local M	no	
VB0	B side vector	no	1st vector argument
VB1		no	2nd vector argument
VB2		no	3rd vector argument
VB3		no	4th vector argument
VB4		no	5th vector argument
VB5		no	6th vector argument
VB6		no	7th vector argument
VB7		no	8th vector argument
VB8		no	9th vector argument
VB9		no	10th vector argument
VB10		no	11th vector argument
VB11		no	12th vector argument
VB12		no	13th vector argument
VB13		no	14th vector argument
VB14		yes	15th vector argument
VB15		yes	16th vector argument
VBL0-VBL7	B side local L	no	
VBM-VBM7	B side local M	no	
D0-D14	D unit local	no	
D15		yes	Stack Pointer
RP	Control	yes	Return Pointer
P0	Vector predicates	no	1st vector predicate argument
P1		no	2nd vector predicate argument
P2		no	3rd vector predicate argument
P3		no	4th vector predicate argument
P4		no	5th vector predicate argument
P5		no	6th vector predicate argument
P6		no	7th vector predicate argument
P7		no	8th vector predicate argument
CUCR0-CUCR3	C-unit Control Register	no	

Nine “scalar” and sixteen “vector” registers are available for argument passing. Arguments whose declared type is 64-bits or less are assigned, in declared order, to scalar registers in the following sequence:

```
A4, A5, A6, A7, A8, A9, A10, A11, A12
```

(Note that any value, including vectors and structures, can be passed in a so-called scalar register as long as it does not exceed 64 bits.)

Arguments whose declared type is between 64 and 512 bits are passed in vector registers, in the following sequence:

```
VB0, VB1, ... VB15
```

Arguments that are declared as vector predicates are passed in vector predicate registers, in the following sequence:

```
P0, P1, ... P7
```

When arguments—including scalars, vectors, complex values, and structures—are passed or returned in registers, they are laid out as specified in Chapter 2.

In C++, the `this` pointer is passed to non-static member functions in A4 as an implicit first argument.

Arguments larger than 512 bits are passed by reference, as described in Section 3.4. Arguments whose declared size is 512 bits or less are passed by value, either in registers or on the stack as follows.

Any arguments not passed in registers are placed on the stack at increasing addresses, such that the first one will be at address SP+16 upon entry to the callee. Each argument with scalar or vector type is placed at the next available address correctly aligned for its type. Structures are aligned to the next power of two greater than or equal to their size, up to a maximum of 8 bytes. (This is so they can be accessed using their container type as defined in Section 2.4.1.) Each argument reserves an amount of stack space equal to its size rounded up to the next multiple of its alignment.

For a variadic C function (that is, a function declared with an ellipsis indicating that it is called with varying numbers of arguments), the last explicitly declared argument and all remaining arguments are passed on the stack, so that its stack address can act as a reference for accessing the undeclared arguments.

Undeclared scalar arguments to a variadic function that are smaller than `int` are promoted to and passed as `int`, in accordance with the C language.

3.3 Return Values

Values whose declared type is 64 bits or less are returned in A4. Values whose declared type is between 64 and 512 bits are returned in VB0. Values that are declared as vector predicates are returned in P0. Values larger than 512 bits are returned by reference, as described in Section 3.4.

3.4 Values Passed and Returned by Reference

Values larger than 512 bits are passed and returned by reference. To pass an argument by reference, the caller places its address in the appropriate location: either in a register or on the stack, according to its position in the argument list. To preserve pass-by-value semantics (required for C and C++), the callee may not modify the pointed-to object; it must make its own copy.

If a called function declared return type is larger than 512 bits, the caller must pass an additional argument in A1 containing a destination address for the returned value, or 0 if the returned value is not used. The callee returns the object by copying it to the address in A1, if non-zero. The caller is responsible for allocating memory if required. Typically this involves reserving space on the stack, but in some cases the address of an already-existing object can be passed and no allocation is required.

For example, if `f` returns a structure, the assignment `s = f()` can be compiled by passing `&s` in A1.

3.5 Conventions for Compiler Helper Functions

The ABI specifies *helper functions* that the compiler uses to implement language features. Generally these functions adhere to the standard calling convention. Exceptions can be made for better performance.

3.6 Scratch Registers for Trampolines and PLT Entries

When a caller-save register is live across a call, but the callee is known not to modify that register, the compiler may optimize the caller by omitting the save and restore of that register around the call. This arises when the definition has been seen, or when calling helper functions with special conventions as described in Section 6.3.

However, the registers AL0-AL2 are designated as potentially modified in any of the following cases, even if the definition has been seen or when calling helper functions:

- The call crosses a section boundary.
- The call is to an imported function.
- The call is to a function that is subject to preemption (Section 5.2).

This is so that if the call requires a far-call trampoline (Section 5.2) or PLT entry (Section 6.3), AL0-AL2 are available as scratch registers

Calls to non-imported functions within the same section never require trampolines or PLT entries; for such intra-section calls AL0-AL2 are treated no differently than other caller-save registers.

Data Allocation and Addressing

This chapter describes how variables are accessed. Section 4.2 covers statically allocated data (including globals). Section 4.3 and Section 4.4 cover local variables and the frame layout. Section 4.5 covers dynamically allocated objects.

Topic **Page**

4.1 Data Sections and Segments

In a relocatable file—that is, an object file output by the compiler or assembler—variables are allocated into sections using default rules and compiler directives. A section is an indivisible unit of allocation in a relocatable file. Sections often contain objects with similar properties. Various sections are designated for data, depending on whether the section is initialized, whether it is writable or read-only, how it will be addressed, and what kind of data it contains.

The data sections defined by the ABI are shown in Figure 4-1. Conventions for placement of static variables into sections and for how they are addressed are covered in Section 4.2.

The linker combines sections from object files to form segments in an ELF load module (executable or shared library). A segment is a continuous range of memory allocated to a load module, representing part of the execution image of the program.

A load module may contain one or more segments for data, into which the linker allocates stack, heap, and static variables. These items may be grouped into a single segment or use multiple segments, subject only to these restrictions:

- All data within a given segment are subject to the same segment attributes (see Chapter 14).
- Within a segment, initialized data must precede uninitialized data. This is a structural constraint of ELF.
- Any additional restrictions imposed by platform-specific conventions.

The runtime environment can dynamically allocate or resize uninitialized data segments, to allocate space for items such as the stack and heap.

Figure 4-1 shows the data sections defined by the ABI, and an abstract mapping of sections into segments. The mapping is only representative; the specific configuration may vary by platform or system. Initialized sections are shaded blue; uninitialized sections are shaded gray.

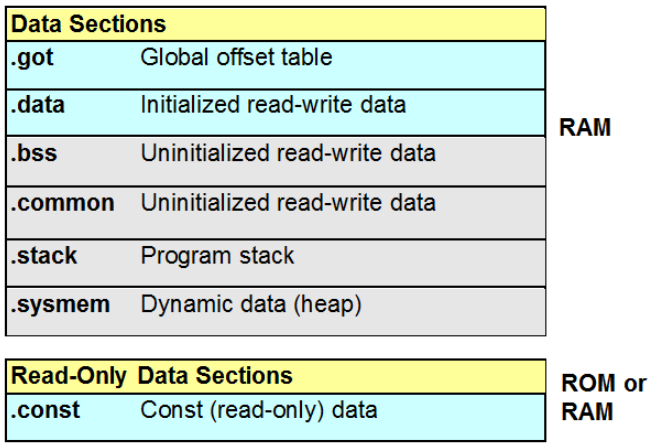


Figure 4-1. Data Sections and Segments (Typical)

The `.got` section contains data structures related to dynamic linking.

The `.data` section contains initialized read-write variables.

The `.bss` section and `.common` sections contain uninitialized variables. Variables in `.bss` are allocated by the assembler, whereas variables in `.common` are allocated by the linker.

The `.stack` section contains local variables and activation frames for functions. It is allocated by the linker.

The `.sysmem` section contains dynamically allocated objects via calls to `malloc()` (or `new()` in C++). It is allocated by the linker.

The `.const` section contains read-only constants. Depending on the platform, the `.const` section may be located in read-only memory.

4.2 Allocation and Addressing of Static Data

All variables that are not auto or dynamic are considered static data; that is, variables with C storage classes `extern` or `static` whose address is established at (static or dynamic) link time. These are allocated into various sections according to their properties and then combined into one or more static data segments.

The C7000 has an MMU that provides memory virtualization. A program executes in the context of a 64-bit virtual address space. All addresses generated and used by the program are virtual addresses, as are all addresses represented in object files and executable files. The MMU maps virtual addresses to physical addresses at runtime. The mapping is established at program load time by whatever agent is responsible for loading the program.

All static data is accessed using PC-relative addressing. That is, whenever the address of a variable is encoded into the program, it is encoded as an offset from the program itself. The use of PC-relative addressing is made practical by the availability of purpose-specific addressing modes and virtual-to-physical address translation performed by the MMU, which combine to provide several advantages:

- Since addresses are encoded as offsets rather than absolute values, the code is naturally position independent.
- There is no need for a data-page pointer (sometimes called a static base pointer) since the encoded offset is relative to the PC.
- The program can be linked, and offsets established, in virtual space without regard to the typical constraints of the physical memory system. Although code and data may need to be far apart in physical space, making it impractical to encode physical data addresses relative to physical code addresses, the virtual addresses can arbitrarily close. Furthermore, the mapping from virtual to physical space can happen after the program is linked, and no change to the encoded offsets (relocation) is required.
- Similarly, shared libraries, in which code is being shared by multiple processes that each require separate instances of the data, can be handled through address translation. The system simply maps the (virtual) data address to a different physical address in each process. • It supports the conventional segment model expected by Linux, in which all the code and static data of a static link unit is collected into one contiguous load segment.

The ISA supports this scheme with efficient PC-relative addressing for load and store instructions. These instructions encode the effective address as a 32-bit signed offset from the PC of the fetch packet containing the access. This provides an address reach of +/- 2GB for variables addressed directly (that is, by name). An implication of using PC-relative addressing for data is that the virtual space available to a given static link unit's code and statically defined variables is limited to 2GB.

This limit does not apply to the stack or other variables accessed exclusively via pointers, such as heap-allocated variables. Pointers are 64 bits wide and can span the entire address space, thereby providing access to objects of arbitrary size and address via calls to malloc() or other mechanisms.

4.2.1 Addressing Methods for Static Data

This section details instruction sequences used for data addressing, along with relevant relocation types.

4.2.1.1 Direct Addressing

A direct reference to a variable (that is, by name) uses a load or store instruction that encodes a PC-relative byte offset as a 32-bit signed value. The offset is split between two fields of the instruction packet, necessitating two relocations.

```
LDW *PC($PCR_OFFSET(sym)),dest   ;reloc R_C7X_PCR_OFFSET_LO5
                                 ;reloc R_C7X_PCR_OFFSET_HI27
```

The LO5 relocation applies to the 5-bit field in the instruction. The HI27 field applies to the constant extension. Section 11.5 describes relocation processing in detail.

The offset is encoded relative to the address of the fetch packet containing the instruction. The `$PCR_OFFSET` assembly operator evaluates to the difference in bytes between the named label (sym in this case) and the address of the fetch packet of the instruction in which it appears.

4.2.1.2 Taking the Address of a Variable

The address of a named variable can be computed by adding its PC-relative offset to the PC.

```
ADDKPC        $PCR_OFFSET(sym),dest    ;reloc R_C7X_PCR_OFFSET_ADDKPC_LO5
                                       ;reloc R_C7X_PCR_OFFSET_ADDKPC_HI27
```

This instruction is relocated in the same ways described for Direct Addressing in Section 4.2.1.1.

4.2.1.3 Absolute Addressing

The C7000 is designed to efficiently support position independent addressing via the PC-relative modes described above. There is little or no performance benefit from using absolute addressing, and its use by the compiler is discouraged. However, for completeness, we show the instruction sequence here.

This instruction formulates a 64-bit constant into a register:

```
MVK64         #sym,dest         ;reloc R_C7X_ABS_MVK_LO10
                                ;reloc R_C7X_ABS_MVK_MID27
                                ;reloc R_C7X_ABS_MVK64_HI27
```

In this instruction, the constant is split between 3 relocatable fields: one in the instruction, plus two constant extension fields.

The `MVK49` instruction is a pseudo-op for `MVK64` (that is, different mnemonic, but same opcode) with a different relocation type for the most significant bits that checks for overflow at 49 bits rather than 64, making it more suitable for use with addresses.

```
MVK49         #sym,dest         ;reloc R_C7X_ABS_MVK_LO10
                                ;reloc R_C7X_ABS_MVK_MID27
                                ;reloc R_C7X_ABS_MVK49_HI12
```

4.2.2 Initialization of Static Data

A static variable that has an initial non-zero value should be allocated into an initialized data section. The section's contents should be an image of the contents of memory corresponding to the initial values of all variables in the section. The variables thus obtain their initial values directly as the section is loaded into memory. This is the so-called *direct initialization model* used by most ELF-based toolchains.

Variables that are expected to be initialized to zero can be allocated into uninitialized sections. The loader is responsible for zeroing uninitialized space at the end of a data segment.

Although the compiler is required to encode initialized variables directly, the linker is not. The linker may translate the directly encoded initialized sections in the object files into an encoded format for the executable file, and rely on a library function to decode the information and perform the initialization at program startup. (Recall that the linker may assume that the library is from the same toolchain.) Encoding initialization data helps save space in the executable file; it also provides an initialization mechanism for self-booting ROM-based systems that do not rely on a loader. The TI toolchain implements such a mechanism, described in Chapter 13. Other toolchains may adopt a compatible mechanism, a different mechanism, or none at all.

4.3 Automatic Variables

Local variables of a procedure, that is, variables with C storage class auto, are allocated either on the stack or in registers, at the compiler's discretion. Variables on the stack are addressed via the stack pointer (D15). The offset is encoded as either an unsigned 5-bit constant, or as a signed 32-bit constant using an extension word of the instruction packet.

The stack is allocated from the `.stack` section, and is part of the data segment(s) of the program.

The stack grows from high addresses toward low addresses. The stack pointer must always remain aligned on a 64-bit (8 byte) boundary. The SP points at the first aligned address below (less than) the currently allocated stack.

Section 4.4 provides more detail on the stack conventions and local frame structure.

4.4 Frame Layout

The ABI specifies the layout of the local frame only to the extent required for procedure linkage and stack unwinding. The rest is at the compiler's discretion.

This section describes conventions for managing the stack, the general layout of the frame, and the layout of the callee-save area.

The stack grows toward zero. The SP is 8-byte aligned, and must remain 8-byte aligned at all times in case an interrupt occurs during frame allocation or deallocation. This means that every SP increment or decrement must be a multiple of 8 bytes. The rationale for 8-byte alignment is so that most 64-bit and smaller objects do not cross memory bank boundaries, which are at 8-byte intervals.

The SP points to a 16-byte *free area* above the topmost allocated location; that is, *(SP+0) through *(SP+15) are free, but *(SP+16) is allocated. This allows the compiler to save two SOE scalar registers to the stack in the first cycle of a called function while allocating the frame with one of the STD instructions (that is, decrementing the SP).

Objects in the frame are accessed using SP-relative addressing with positive offsets.

The stack frame of a function contains the following areas:

- *Incoming arguments* that are passed on the stack are part of the caller's frame.
- The *callee-save* area stores registers modified by the function that must be preserved. If exceptions are enabled, a specific layout must be adhered to. If not, a compiler is free to use alternative schemes for saving registers.
- The *locals* and *spill temps* area consists of temporary storage used by the function.
- The *outgoing arguments* section is for passing non-register arguments to called functions, as detailed in Section 3.2. The size of the section is the maximum required for any single call.
- The 16-byte *free area* facilitates stack allocation by the callee.

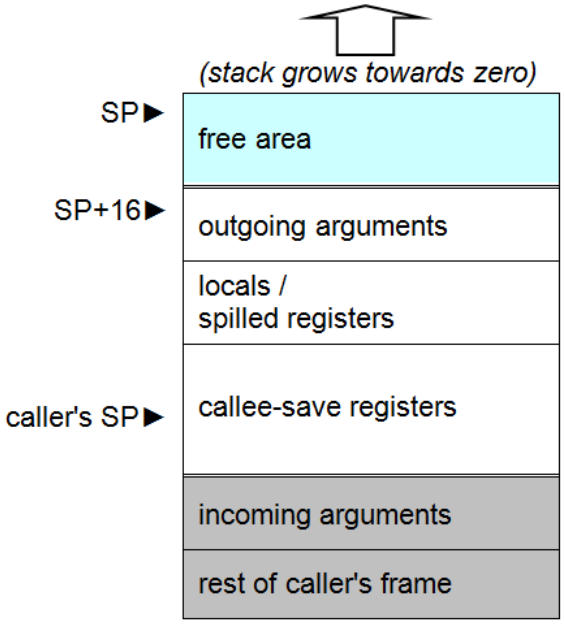
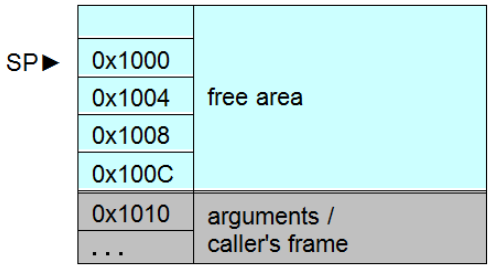


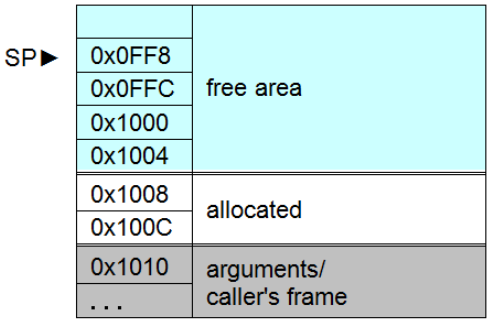
Figure 4-2. Local Frame Layout

4.4.1 Stack Allocation

Before the first instruction in a function, the stack looks like this:



If this function needs one or more bytes on the stack to store something, it will need to allocate a frame of at least 8 bytes (because SP must remain 8-byte aligned). The allocation is performed by decrementing SP by 8, either with an explicit decrement or in combination with a store (that is, a “push” instruction). With the previous example, after a minimal allocation of 8 bytes, the stack looks like this:



4.4.2 Register Save Order

As discussed in Section 3.2, functions are responsible for preserving the contents of registers designated as callee-save, normally accomplished by saving modified registers in the local frame upon entry to the function and restoring them before exit. Usually, the order, locations, and layout of the callee-save registers on the stack don’t matter, as long as they are restored from the same location and in the same way as they were saved.

By default, the compiler saves registers in an arbitrary order. However, when exception handling is enabled, the stack unwinding process that processes an exception at runtime needs to know exactly where each register is so that it can simulate the function epilog. To efficiently encode this information using a bitvector, we define a fixed order called the *TDEH Save Order* [2].

The TDEH save order is:

```
A8, A9, A10, A11, A12, A13, A14, A15, RP, VB15/B15, VB14/B14
```

When using the TDEH save order, the compiler will always save registers in that order, starting at the bottom (highest address) of the frame. Vector registers are saved as byte vectors – that is, using the VSTB instruction, regardless of their contents. If any registers are not saved, the registers will be packed so that there are no holes in the stack, but the relative order will remain the same.

4.5 Heap-Allocated Objects

Dynamically allocated objects, such as via C’s `malloc()` or C++’s `new` operations, are allocated by the runtime library. An execution environment may provide its own implementation of these functions provided they conform to the API specified by the language standard. This ABI does not specify any additional requirements on the dynamic allocation mechanism.

[2] TDEH stands for Table-Driven Exception Handling

Code Allocation and Addressing

The compiler and assembler generate code into one or more sections. The default code section is called `.text` but the programmer may direct code into additional named sections. The linker combines code sections into one or more segments. The base ABI imposes no restrictions on the number, size, or placement of code sections, although there may be platform-specific restrictions.

All instructions on the C7000 are 32-bits wide. Labels that represent function addresses, as well as most other labels, are always aligned on 32-bit boundaries. Instructions are grouped into *fetch packets* of 512 bits (64 bytes, or 16 instructions). Fetch packets are aligned on 64-byte boundaries. Instructions are also grouped into *execute packets*; instructions in the same execute packet execute in parallel. The groupings are independent, in that execute packets may span fetch packet boundaries, and vice versa. The main significance of the fetch packet boundary is that it defines the PC (referred to as PCE1) used to compute PC-relative offsets.

As with data, all code addresses are expressed using PC-relative addressing; this facilitates position independence.

There are three ways a code object may be referenced: as a branch destination, by calling it as a function, or computing its address for use in an indirect branch or call. These cases are described in the sections that follow.

Topic **Page**

5.1 Branching

Branches are always assumed to be within the same function, and therefore can always use PC-relative addressing and be resolved no later than static link time. There are two encodings. Conditional branches use a 19-bit signed offset scaled by 2 bits, yielding a range of $\pm 2^{20}$ bytes (1MB). Unconditional branches use a 24-bit signed offset scaled by 2 bits, yielding a range of $\pm 2^{25}$ (32MB).This effectively limits the size of any given function to 32MB.

Conditional Branch

```
[creg]    B label  ; R_C7X_PCR_BRANCH_LO19
```

Unconditional Branch

```
B    label         ; R_C7X_PCR_BRANCH_LO24
```

5.2 Calls

The C7000 has a specific call instruction, which integrates transfer of control with computation of the return address. Like branches, calls use PC-relative addressing.

Direct PC-relative Call

```
CALL    sym        ;reloc R_C7X_PCR_BRANCH_LO24
```

Unconditional calls are encoded similarly to unconditional branches, with a 24-bit PC-relative offset, yielding a direct call reach of 32MB. If, at link time, the call exceeds this range, the static linker may need to route the call through a small “stub” procedure called a *trampoline*.

Far Call Trampoline

If the call target is defined in the same static link unit, but is unreachable with a 24-bit offset, the static linker generates a *trampoline*, which is a stub that uses some form of extended addressing to reach the target. Assuming the target address can be bound at static link time, the trampoline [3] uses an “extended branch” that can cover the entire address space. It uses a 46-bit scaled PC-relative offset, yielding a range of $\pm 2^{47}$ bytes.

```
$Tramp$$sym:
    BE  sym               ;reloc R_C7X_PCR_EBRANCH_LO19
                          ;reloc R_C7X_PCR_EBRANCH_HI27
```

The linker is responsible for placing the trampoline within the reachable range of the call.

If the call target is not defined in the same static link unit, or cannot be statically bound due to preemption or other post-link resolution, the static linker generates a *PLT entry*, which is similar to a trampoline. This case is covered in Section 6.3.

The trampoline can use AL0, AL1, and AL2 as temporary registers. The trampoline must not modify any other registers.

Indirect Calls

An indirect call through a function pointer generates a call with a register operand.

```
; ...  load reg with dest addr
CALLA        reg
```

Return Instruction

A function return is executed by branching to the address stored in the RP (Return Pointer) register. The callee is free to move the address and store it elsewhere, typically required for nested calls. Assuming the address is in RP, a function return is:

```
RET                   ; return
```

[3] The name of the trampoline label may vary by convention.

5.3 Computing Code Addresses

A code address is needed in three circumstances:

- It appears as an expression in the program (`&func`)
- In switch tables
- In a trampoline or PLT entry generated by the linker

There are two basic ways to form the address of a code object: PC-relative and GOT-based. Both forms are position independent.

PC-Relative Addressing

To materialize a code address into a register, the address is computed as the sum of the address of the current fetch packet and a PC-relative offset.

```
ADDKPC    $PCR_OFFSET(label),dest   ;reloc R_C7X_PCR_OFFSET_ADDKPC_LO5
                                    ;reloc R_C7X_PCR_OFFSET_ADDKPC_HI27
```

The `$PCR_OFFSET` assembly operator evaluates to the difference in bytes between the fetch packet of the instruction in which it appears and the target label. The encoded offset is a signed 32-bit field, giving a reach of $\pm 2^{31}$ bytes (2GB).

Switch Tables

Switch tables are tables of code addresses that are indexed by the switch value. There are two separate issues with addressing switch tables: how to address the table itself, and how to encode the code addresses within the table.

PC-relative addressing provides a straightforward way to address the table. The table is read-only, so it naturally associates with the code and is position independent.

Assume the switch expression has been computed into `switch_index`. Using PC-relative addressing, the sequence to fetch a switch table entry is:

```
ADDKPC        $PCR_OFFSET(table),table_base        ; 2 relocs for ADDKPC
LDW           *table_base[switch_index],target_pc_offset
...
```

The ADDKPC instruction materializes the address of the switch table into a register. The LDW then indexes into the table to read the proper entry.

The value read from the table is another PC-relative offset, this time between the base address of the table and the case label. Adding these together yields the destination address:

```
ADDAB        table_base,target_pc_offset,dest
```

The final step is an indirect branch.

```
BA           dest
```

The table itself contains offsets to the case labels:

```
table:
   .word        $LABEL_DIFF(case0,table)
   .word        $LABEL_DIFF(case1,table)
   .word        $LABEL_DIFF(case2,table)
   ...
```

The `$LABEL_DIFF` operator simply evaluates to the difference, in bytes, between the two labels. Currently, these are expected to be in the same section, so no relocation type is defined.

Absolute Addressing for Code

As with data, absolute addressing for code is discouraged as it is position dependent and has little to no performance advantage.

The `MVK49` pseudo-instruction can be used to materialize absolute addresses, as shown in the section on data addressing (Section 4.2.1).

Helper Function API

To enable object files built with one toolchain to be linked with a runtime support ("RTS") library from another, the API between them must be specified. The interface has two parts. The first specifies functions on which the compiler relies to implement aspects of the language not directly supported by the instruction set. These are called helper functions, and are documented in this chapter. The second involves standardization of compile-time aspects of the source language library standard, such as the C, C99, or C++ Standard Libraries, which are covered in separate chapters.

Topic **Page**

6.1 Floating-Point Behavior

Floating-point behavior varies by device and by toolchain and is therefore difficult to standardize. The goal of the ABI is to provide a basis for conformance to the C, C99, and C++ standards. Of these C99 is the best-specified with respect to floating-point. Appendix F of the C99 standard defines floating-point behavior of the C language behavior in terms of the IEEE floating-point standard (ISO IEC 60559:1989, previously designated as ANSI/IEEE 754−1985).

The C7000 ABI specifies that the helper functions in this section that operate on floating-point values must conform to the behavior specified by Appendix F of the C99 standard.

C99 allows customization of, and access to, the floating-point behavioral environment though the `<fenv.h>` header file. For purposes of standardizing the behavior of the helper functions, they are specified to operate in accordance with a basic default environment, with the following properties:

- The rounding mode is round to nearest. Dynamic rounding precision modes are not supported.
- No floating-point exceptions are supported.
- Inputs that represent Signaling NaNs behave like Quiet NaNs.
- The helper functions support only the behavior under the `FENV_ACCESS` “off” state. That is, the program is assumed to execute in non-stop mode and assumed not to access the floating-point environment.

A toolchain is free to implement more complete floating-point support, using its own library. Users who invoke toolchain-specific floating-point support may be required to link using that toolchain’s library (in addition to an ABI-conforming helper function library).

6.2 C Helper Function API

The compiler generates calls to helper functions to perform operations that need to be supported by the compiler, but are not supported directly by the architecture, such as floating-point operations on devices that lack dedicated hardware. These helper functions must be implemented in the RTS library of any toolchain that conforms to the ABI.

Helper functions are named using the prefix `__c7xabi_`. Any identifier with this prefix is reserved for the ABI.

The helper functions adhere to the standard calling conventions, except as indicated in Section 6.3.

Table 6-1 specifies the helper functions using C notation and syntax. The types in the table correspond to the generic data types specified in Section 2.1.

Table 6-1. Helper Functions

C Function	Description
void __c7xabi_abort_msg(const char *)	Report failed assertion. (See notes below.)
double __c7xabi_divd(double, double)	Divide two double-precision floats.
float __c7xabi_divf(float, float)	Divide two single-precision floats.
int __c7xabi_divi(int, int)	32-bit signed integer division.
long long __c7xabi_divlli(long long, long long)	64-bit signed integer division.
unsigned __c7xabi_divu(unsigned, unsigned)	32-bit unsigned integer division.
unsigned long long __c7xabi_divull(unsigned long long, unsigned long long)	64-bit unsigned integer division.
long long __c7xabi_fixdlli(double)	Convert double-precision float to 64-bit integer.
unsigned __c7xabi_fixdu(double)	Convert double-precision float to 32-bit unsigned integer.
unsigned long long __c7xabi_fixdull(double)	Convert double-precision float to 64-bit unsigned integer.
long long __c7xabi_fixflli(float)	Convert single-precision float to 64-bit integer.
unsigned __c7xabi_fixfu(float)	Convert single-precision float to 32-bit unsigned integer.
unsigned long long __c7xabi_fixfull(float)	Convert single-precision float to 64-bit unsigned integer.
double __c7xabi_fltllid(long long)	Convert 64-bit integer to double-precision float.
float __c7xabi_fltllif(long long)	Convert 64-bit integer to single-precision float.
double __c7xabi_fltud(unsigned)	Convert 32-bit unsigned integer to double-precision float.

Table 6-1. Helper Functions (continued)

C Function	Description
float __c7xabi_fltuf(unsigned)	Convert 32-bit unsigned integer to single-precision float.
double __c7xabi_fltulld(unsigned long long)	Convert 64-bit unsigned integer to double-precision float.
float __c7xabi_fltullf(unsigned long long)	Convert 64-bit unsigned integer to single-precision float.
int __c7xabi_remi(int, int)	32-bit integer modulo.
long long __c7xabi_remlli(long long, long long)	64-bit integer modulo.
unsigned __c7xabi_remu(unsigned, unsigned)	32-bit unsigned integer modulo.
unsigned long long __c7xabi_remull(unsigned long long, unsigned long long)	64-bit unsigned integer modulo.
void __c7xabi_strasg(int*, const int*, unsigned)	Block copy. (See notes below.)
__c7xabi_unwind_cpp_pr0	Short frame unwinding, 16-bit scope.
__c7xabi_unwind_cpp_pr1	Long frame unwinding, 16-bit scope.
__c7xabi_unwind_cpp_pr2	Long frame unwinding, 32-bit scope.
__c7xabi_unwind_cpp_pr3	Unwinding, 24-bit encoding, 16-bit scope.

__c7xabi_abort_msg() Function:

```
void __c7xabi_abort_msg(const char *msg)
```

The function `__c7xabi_abort_msg()` is generated to print a diagnostic message when a run-time assertion (for example, the C assert macro) fails. It must not return. That is, it must call abort or terminate the program by other means.

__c7xabi_strasg() Function:

```
void __c7xabi_strasg(int* dst, const int* src, unsigned             cnt)
```

The function `__c7xabi_strasg()` is generated by the compiler for efficient out-of-line structure or array copy operations. The `cnt` argument is the size in bytes, which must be a multiple of 4 greater than or equal to 28 (7 words). It makes the following assumptions:

- The src and dst addresses are word-aligned.
- The src and dst objects do not overlap.

6.3 Special Register Conventions for Helper Functions

The helper functions adhere to the standard calling conventions, except as specifically noted above. However, typical implementations require a small subset of the available registers. If a caller is using a register that would normally have to be preserved across a call (that is, a caller-save register), but the helper function is known not to use it, then the caller can avoid having to save it. For this reason the ABI changes the designation of these registers on a function-by-function basis so that callers are not required to unnecessarily preserve unused registers.

Note that from a compiler's point of view, use of this information is optional, providing only an optimization opportunity. From a library implementer's point of view, the ABI mandates that alternate implementations of the helper functions must conform to the additional restrictions.

Currently the compiler does not specialize calls for any helper functions.

Note that AL0-AL2 are assumed to be modified by any call, even if they are not used by the callee. This is so they are available as scratch registers if the call is routed through a trampolines or PLT entry. See Section 3.6.

Standard C Library API

Toolchains typically include standard libraries for the language they support, such as C, C99, or C++. These libraries have compile-time components (header files) and runtime components (variables and functions).

Interoperability requires that code built with one toolchain can be linked with a library from another, implying that the ABI must specify the interface between any compile-time and runtime aspects of the library. During compilation, the compiler and the library header files are required to be from the same implementation.

During linking, the linker and library are required to be from the same implementation, which may be different from the implementation of the compiler.

Topic **Page**

7.1 Reserved Symbols

A number of symbols are reserved for use in the RTS library as described for the ABI. These include the following:

- _ctypes_
- _ftable

In addition, any symbols listed in Section 11.4.4 or symbols with the prefixes listed in Section 10.3 and Section 11.4.3 are reserved.

7.2 <assert.h> Implementation

The library must implement assert as a macro. If its expression argument is false, it must eventually call a helper function `__c7xabi_abort_msg` to print the failure message. Whether or not the helper function actually causes something to be printed is implementation-defined. As specified by the C standard, this helper function must terminate by calling abort.

7.3 <complex.h> Implementation

The C99 standard requires that a complex number be represented as a struct containing one array of two elements of the corresponding real type. Element 0 is the real component, and element 1 is the imaginary component. For instance, _Complex double is:

```
{ double _Val[2]; } /* where 0=real 1=imag */
```

TI's C7000 toolset supports the C99 complex numbers and provides this header file.

7.4 <ctype.h> Implementation

The ctype.h functions are locale-dependent and therefore may not be inlined. These functions include:

- isalnum
- isalpha
- isblank (a C99 function; this is not yet provided by the TI toolset)
- iscntrl
- isdigit
- isgraph
- islower
- isprint
- ispunct
- isspace
- isupper
- isxdigit
- isascii (obsolete function, not a standard C99 function)
- toupper (currently inlined by the TI compiler, but subject to change)
- tolower (currently inlined by the TI compiler, but subject to change)
- toascii (obsolete function, not a standard C99 function)

7.5 <errno.h> Implementation

The following constants are defined for use with errno:

```
#define EDOM            0x0021
#define ERANGE          0x0022
#define EILSEQ          0x0058
#define E2BIG           0x0007
#define EACCES          0x000D
#define EAGAIN          0x000B
#define EBADF           0x0009
#define EBADMSG         0x004D
#define EBUSY           0x0010
#define ECANCELED       0x002F
#define ECHILD          0x000A
#define EDEADLK         0x002D
#define EEXIST          0x0011
#define EFAULT          0x000E
#define EFBIG           0x001B
#define EINPROGRESS     0x0096
#define EINTR           0x0004
#define EINVAL          0x0016
#define EIO             0x0005
#define EISDIR          0x0015
#define EMFILE          0x0018
#define EMLINK          0x001F
#define EMSGSIZE        0x0061
#define ENAMETOOLONG    0x004E
#define ENFILE          0x0017
#define ENODEV          0x0013
#define ENOENT          0x0002
#define ENOEXEC         0x0008
#define ENOLCK          0x002E
#define ENOMEM          0x000C
#define ENOSPC          0x001C
#define ENOSYS          0x0059
#define ENOTDIR         0x0014
#define ENOTEMPTY       0x005D
#define ENOTSUP         0x0030
#define ENOTTY          0x0019
#define ENXIO           0x0006
#define EPERM           0x0001
#define EPIPE           0x0020
#define EROFS           0x001E
#define ESPIPE          0x001D
#define ESRCH           0x0003
#define ETIMEDOUT       0x0091
#define EXDEV           0x0012
/* TI specific value used in ftell() and fgetpos() */
#define EFPOS           0x0098
```

7.6 <float.h> Implementation

The macros in this file are defined in the natural way. Float is IEEE-32; double and long double are IEEE-64.

7.7 <inttypes.h> Implementation

The macros, functions and typedefs in this file are defined in the natural way according to the integer types of the architecture.

7.8 <iso646.h> Implementation

The macros in this file are fully specified by the C standard and are defined in the natural way.

7.9 <limits.h> Implementation

Aside from MB_LEN_MAX, the macros in this file are defined in the natural way according to the integer types of the architecture. MB_LEN_MAX is defined as follows:

```
#define MB_LEN_MAX 1
```

7.10 <locale.h> Implementation

TI's toolset provides only the "C" locale. The LC_* macros are defined as follows:

```
#define LC_ALL 0
#define LC_COLLATE 1
#define LC_CTYPE 2
#define LC_MONETARY 3
#define LC_NUMERIC 4
#define LC_TIME 5
#define LC_MESSAGES 6
```

The order of the fields in the lconv struct is as follows: (These are the C89 fields. Additional fields added for C99 are not included.)

```
char *decimal_point;
char *grouping;
char *thousands_sep;
char *mon_decimal_point;
char *mon_grouping;
char *mon_thousands_sep;
char *negative_sign;
char *positive_sign;
char *currency_symbol;
char frac_digits;
char n_cs_precedes;
char n_sep_by_space;
char n_sign_posn;
char p_cs_precedes;
char p_sep_by_space;
char p_sign_posn;
char *int_curr_symbol;
char int_frac_digits;
```

7.11 <math.h> Implementation

The macros defined by this library must be floating-point constants (not library variables).

- HUGE_VALF must be float infinity.
- HUGE_VAL must be double infinity.
- HUGE_VALL must be long double infinity.
- INFINITY must be float infinity.
- NAN must be quiet NaN.
- MATH_ERRNO is 1.
- MATH_ERREXCEPT is 2.

The following FP_* macros are defined:

```
#define FP_INFINITE 1
#define FP_NAN 2
#define FP_NORMAL 3
#define FP_SUBNORMAL 5
#define FP_ZERO 4
#define FP_ILOGB0 (-__INT_MAX)
#define FP_ILOGBNAN (__INT_MAX)
```

The other FP_* macros are not currently specified.

7.12 <setjmp.h> Implementation

The type and size of jmp_buf are defined in setjmp.h.

The size and alignment of jmp_buf is the same as an array of 26 "long long"s (that is, 64 bits * 26).

The setjmp and longjmp functions must be not be inlined because jmp_buf is opaque. That is, the fields of the structure are not defined by the standard, so the internals of the structure are not accessible except by setjmp() and longjmp(), which must be out-of-line calls from the same library. These functions cannot be implemented as macros.

7.13 <signal.h> Implementation

TI's toolset creates the following typedefs:

```
typedef int sig_atomic_t;
typedef void __sighandler_t(int);
```

TI's toolset defines the following constants:

```
#define SIG_DFL ((__sighandler_t *) 0)
#define SIG_ERR ((__sighandler_t *) -1)
#define SIG_IGN ((__sighandler_t *) 1)
#define SIGINT 2
#define SIGILL 4
#define SIGABRT 5
#define SIGFPE 8
#define SIGSEGV 11
#define SIGTERM 15
#define NSIG 32
```

7.14 <stdarg.h> Implementation

Upon a call to a variadic C function declared with an ellipsis (...), the last declared argument and any additional arguments are passed on the stack as described in Section 3.2 and accessed using the macros in <stdarg.h>. The macros use a persistent argument pointer that is initialized via an invocation of va_start and advanced via invocations of va_arg. The following conventions apply to the implementation of these macros.

- The type of `va_list` is `char *`.
- Invocation of the macro `va_start(ap, parm)` sets `ap` to point 1 byte past the last (greatest) address allocated to parm.
- Each successive invocation of `va_arg(ap, type)` leaves `ap` pointing 1 byte past the last address reserved for the argument object indicated by type.

7.15 <stdbool.h> Implementation

For C++, the type `bool` is a built-in type.

For C99, the type `_Bool` is a built-in type. For C99, the header file stdbool.h defines a macro "bool" which expands to _Bool.

Each of these types is represented as an 8-bit unsigned type.

7.16 <stddef.h> Implementation

stddef.h contains the following typedefs:

```
typedef long ptrdiff_t;
typedef unsigned long size_t;
typedef unsigned int wchar_t;
```

7.17 <stdint.h> Implementation

The macros and typedefs in this header file are defined in the natural way according to the integer types of the architecture.

7.18 <stdio.h> Implementation

The TI toolset defines the following constants for use with the stdio.h library:

```
#define _IOFBF       0x0001
#define _IOLBF       0x0002
#define _IONBF       0x0004
#define _BUFFALOC    0x0008
#define _MODER       0x0010
#define _MODEW       0x0020
#define _MODERW      0x0040
#define _MODEA       0x0080
#define _MODEBIN     0x0100
#define _STATEOF     0x0200
#define _STATERR     0x0400
#define _UNGETC      0x0800
#define _TMPFILE     0x1000
#define BUFSIZ       256
#define FOPEN_MAX    _NFILE
#define FILENAME_MAX 256
#define TMP_MAX      65535
#define L_tmpnam     _LTMPNAM
#define stdin        (&_ftable[0])
#define stdout       (&_ftable[1])
#define stderr       (&_ftable[2])
#define SEEK_SET     (0x0000)
#define SEEK_CUR     (0x0001)
#define SEEK_END     (0x0002)
#define EOF          (-1)
```

The FOPEN_MAX, FILENAME_MAX, TMP_MAX, and L_tmpnam values are actually minimum maxima. The library is free to provide support for more/larger values, but must at least provide the specified values.

Because the TI toolset defines stdout and stderr as `&_ftable[1]` and `&_ftable[2]`, the size of FILE must be known to the implementation.

In the TI header files, stdin, stdout, and stderr expand to references into the array `_ftable`. To successfully interlink with such files, any other implementations need to implement the FILE array with exactly that name. The C7000 EABI does not have a "compatibility mode" (like the mode in the Arm® EABI) in which stdin, stdout, and stderr are link-time symbols, not macros. The lack of a compatibility mode means that linkers that need to interlink with a module that refers to stdin directly need to support `_ftable`.

If a program does not use the stdin, stdout, or stderr macros (or a function implemented as a macro that refers to one of these macros), there are no issues with the FILE array. C I/O functions commonly implemented as macros—getc, putc, getchar, putchar—must not be inlined.

The `fpos_t` type is defined as a `long`.

7.19 <stdlib.h> Implementation

The TI toolset defines the stdlib.h structures as follows:

```
typedef struct { int quot, rem; } div_t;
typedef struct { long quot, rem; } ldiv_t;
typedef struct { long long quot, rem; } lldiv_t;
```

The TI toolset defines constants for use with the stdlib.h library as follows:

```
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
#define MB_CUR_MAX 1
```

The results of the rand function are not defined by the ABI specification. The function is required to be thread-local. This ABI specification does not require a library to implement either the getenv or system function. The TI toolset does provide a getenv function, which requires debugger support.

7.20 <string.h> Implementation

The `strtok` function must not be inlined, because it has a static state. The `strcoll` and `strxfrm` functions also must not be inlined, because they depend on the locale.

7.21 <tgmath.h> Implementation

The C99 standard completely specifies this header file.

7.22 <time.h> Implementation

The typedefs and constants defined for this library are dependent on the execution environment. In order to make code portable, the code must not make assumptions about the type and range of `time_t` or `clock_t` or the value of `CLOCKS_PER_SEC`.

7.23 <wchar.h> Implementation

The TI toolset defines the following type and constant for use with this library:

```
typedef int wint_t;
#define WEOF ((wint_t)-1)
```

The type `mbstate_t` is the size and alignment of `int`.

7.24 <wctype.h> Implementation

The TI toolset defines the following types for use with this library:

```
typedef void *wctrans_t;
typedef void *wctype_t;
```

8 C++ ABI

The C++ ABI specifies aspects of the implementation of the C++ language that must be standardized in order for code from different toolchains to interoperate. The C7000 C++ ABI is based on the Generic C++ ABI originally developed for IA-64 but now widely adopted among C++ toolchains, including GCC. The base standard, referred to in this chapter as GC++ABI, can be found at http://refspecs.linux-foundation.org/cxxabi-1.83.html.

This chapter documents additions to and deviations from that base document.

Topic **Page**

8.1 Limits (GC++ABI 1.2)

The GC++ABI constrains the offset of a non-virtual base subobject in the full object containing it to be representable by a 56-bit signed integer, due to the RTTI implementation. This implies a practical limit of 2^{55} bytes on the size of a class.

8.2 Export Template (GC++ABI 1.4.2)

Export templates are not currently specified by the ABI.

8.3 Data Layout (GC++ABI Chapter 2)

The layout of POD ("Plain Old Data"), is specified in Chapter 2 of this C7000 ABI document. The layout of non-POD data is as specified by the base document. There is a minor exception for bit-fields, which are covered in Section 2.5.

8.4 Initialization Guard Variables (GC++ABI 2.8)

The guard variable is a one-byte field stored in the first byte of a 32-bit container. A non-zero value of the guard variable indicates that initialization is complete. This follows the IA-64 scheme, except the container is 32 bits instead of 64.

This is a reference implementation of the helper function `__cxa_guard_acquire`, which reads the guard variable and returns 1 if the initialization is not yet complete, 0 otherwise:

```
int __cxa_guard_acquire(unsigned int *guard)
{
       char *first_byte = (char *)guard;
       return (*first_byte == 0) ? 1 : 0;
}
```

This is a reference implementation of the helper function `__cxa_guard_release`, which modifies the guard object to signal that initialization is complete:

```
void __cxa_guard_release(unsigned int *guard)
{
       char *first_byte = (char *)guard;
       *first_byte = 1;
}
```

8.5 Constructor Return Value (GC++ABI 3.1.5)

The C7000 follows the ARM EABI, under which the C1 and C2 constructors return the `this` pointer. Doing so allows tail-call optimization of calls to these functions.

Similarly, non-virtual calls to D1 and D2 destructors return `this`. Calls to virtual destructors use thunk functions, which do not return `this`.

Section 3.3 of the GC++ABI specification describes several library helper functions for array new and delete, which take pointers to constructors or destructors as parameters. In the GC++ABI these parameters are declared as pointers to functions returning `void`, but in the C7000 ABI they are declared as pointers to functions that return `void*`, corresponding to this.

8.6 One-time Construction API (GC++ABI 3.3.2)

The guard variable is an 8-bit field stored in the first byte of a 32-bit container. See Section 8.4.

8.7 Controlling Object Construction Order (GC++ ABI 3.3.4)

The C7000 ABI does not specify a mechanism to control object construction.

8.8 Static Data (GC++ ABI 5.2.2)

The GC++ ABI requires that a static object referenced by an inline function be defined in a COMDAT group. If such an object has an associated guard variable, then the guard variable must also be defined in a COMDAT group. The GC++ABI permits the static variable and its guard variable to be in different groups, but discourages this practice. The C7000 ABI forbids it altogether; the static variable and its guard variable must be defined in a single COMDAT group with the static variable's name as the signature.

8.9 Virtual Tables and the Key function (GC++ABI 5.2.3)

The GC++ABI defines a class's key function, whose definition triggers creation of the virtual table for that class, to be the first non-pure virtual function that is not inline *at the point of class definition*. The C7000 ABI modifies this to be the first non-pure virtual function that is not inline *at the end of the translation unit*. In other words, an inline member is not a key function if it's first declared inline after the class definition.

8.10 Unwind Table Location (GC++ABI 5.3)

Exception handling is covered in Chapter 9 of this document.

Exception Handling

The C7000 EABI employs *Table-Driven Exception Handling (TDEH)*. TDEH implements exception handling for languages that support exceptions, such as C++.

TDEH uses tables to encode information needed to handle exceptions. The tables are part of the program's read-only data. When an exception is thrown, the exception handling code in the runtime support library propagates the exception by *unwinding* the stack to the stack frame representing a function with a catch clause that will catch the exception. As the stack is unwound, locally-defined objects must be destroyed (by calling the destructor) along the way. The tables encode information about how to unwind the stack, which objects to destroy when, and where to transfer control when the exception is finally caught.

TDEH tables are generated into executable files by the linker, using information generated into relocatable files by the compiler. This chapter specifies the format and encoding of the tables, and how the information is used to propagate exceptions. An ABI-conforming toolchain must generate tables in the format specified here.

Topic **Page**

9.1 Overview

The C7000's exception handling table format and mechanism is based on that of the ARM processor family, which itself is based on the IA-64 Exception Handling ABI (http://www.codesourcery.com/public/cxx-abi/abi-eh.html). This chapter focuses on the C7000-specific portions.

TDEH data consists of three main components: the EXIDX, the EXTAB, and catch and cleanup blocks.

The Exception Index Table (*EXIDX*) maps program addresses to entries in the Exception Action Table (*EXTAB*). All addresses in the program are covered by the EXIDX.

The EXTAB encodes instructions which describe how to unwind a stack frame (by restoring registers and adjusting the stack pointer) and which catch and cleanup blocks to invoke when an exception is propagated.

Catch and cleanup blocks (collectively known as *landing pads*) are code fragments that perform exception handling tasks. Cleanup blocks contain calls to destructor functions. Catch blocks implement catch clauses in the user's code. These blocks are only executed when an exception actually gets thrown. These blocks are generated for a function when the rest of the function is generated, and execute in the same stack frame as the function, but may be placed in a different section.

9.2 PREL30 Encoding

Some fields of the EXIDX and EXTAB tables need to record program memory addresses or pointers to other locations in the tables, both of which are typically in code or read-only segments. To facilitate position independence, references in TDEH tables use a position-relative offset. Such references are always within the same static link unit, and the address range of a single static link unit is constrained to 2GB, so a 32-bit signed byte offset suffices. The offset is scaled as words and encoded using a 30-bit field and a special-purpose position-relative relocation called R_C7X_PREL30, abbreviated here as "PREL30".

A PREL30 field is encoded as a scaled, signed 30-bit offset which occupies the least significant 30 bits of a 32-bit word. The remaining (most significant) bits are used for different purposes in different contexts. The relocated address to which the field refers is found by left-shifting the encoded offset by 2 bits and adding it to the address of the field.

9.3 The Exception Index Table (EXIDX)

When a throw statement is seen in the source code, the compiler generates a call to a runtime support library function named `__cxa_throw`. When the throw is executed, the return address for the `__cxa_throw` call site is used to identify which function is throwing the exception. The library searches for the return address in the EXIDX table.

Each entry in the table represents the exception handling behavior of a range of program addresses, which may be one or several functions that share exactly the same exception handling behavior. Each entry encodes the start of a program address range, and is considered to cover all program addresses until the address encoded in the next entry. The linker may combine adjacent functions with identical behavior into one entry.

Each entry consists of two 32-bit words. The first word of each entry is a PREL30 field representing the starting program address of the function or functions. Bit 31 of the first word shall be 0. The second word has one of three formats, depending on bit 31 of the second word. If bit 31 is 0, the second word is either a PREL30 pointer to an EXTAB entry somewhere else in memory or the special value `EXIDX_CANTUNWIND`. If bit 31 is 1, the second word is an inlined EXTAB entry. These three formats are as follows:

Pointer to Out-of-Line EXTAB entry

In this format, the second word of the EXIDX table entry contains 0 in the top bit and the PREL-30-encoded address of the EXTAB entry for this address range in the other bits.

31	30	29-0
0	X	PREL30 representation of function address
0	X	PREL30 representation of EXTAB entry

EXIDX_CANTUNWIND

As a special case, if the second word of the EXIDX has the value 0x1, the EXIDX represents `EXIDX_CANTUNWIND`, indicating that the function cannot be unwound at all. If an exception tries to propagate through such a function, the unwinder will call `abort` or `std::terminate` depending on the language.

31	30	29-0
0		PREL30 representation of function address
0x00000001 (EXIDX_CANTUNWIND)		

Inlined EXTAB Entry

If the entire EXTAB entry for this function is small enough, it is placed in the second EXIDX word and bit 31 is set to one. The second word uses the same encoding as the EXTAB compact model described in Section 9.4, but with no descriptors and no terminating NULL. This saves 4 bytes that would have been a pointer to an out-of-line EXTAB entry plus 4 bytes for the terminating NULL.

31	30-28	27-24	23-0
0	X	PREL30 representation of function address	
1	**000**	PR index	data for personality routine specified by ‘index’

9.4 The Exception Handling Instruction Table (EXTAB)

Each EXTAB entry consists of one or more 32-bit words that encode frame unwinding instructions and descriptors to handle catch and cleanup. The first word describes that entry’s *personality*, which is the format and interpretation of the entry.

When an exception is thrown, EXTAB entries are decoded by *personality routines* provided in the runtime support library. Personality routines specified by the ABI are listed in Table 9-1.

EXTAB Generic Model

A generic EXTAB entry is indicated by setting bit 31 of the first word to 0. The first word has a PREL30 entry representing the address of the personality routine. The rest of the words in the EXTAB entry are data that are passed to the personality routine.

31	30	29-0
0	X	PREL30 representation of personality routine address
optional data for the personality routine		

The format of the optional data is up to the discretion of the personality routine, but the length must be an integer multiple of whole 32-bit words. The unwinder calls the personality routine, passing it a pointer to the first word of optional data.

EXTAB Compact Model

A compact EXTAB entry is indicated by a 1 in bit 31 of the first word. (When an EXTAB entry is encoded into the second word of an EXIDX entry, the compact form is always used.) In the compact form, the personality routine is encoded by a 4-bit PR index in the first byte of the entry. The remaining 3 bytes contain unwinding instructions as specified by the personality routine. In a non-inlined EXTAB entry, additional data is provided in additional successive 32-bit words: any additional unwinding instructions, followed optionally by action descriptors, terminated with a NULL word.

31	30-28	27-24	23-0
1	**000**	PR index	encoded unwinding instructions
zero or more additional 32-bit words of unwinding instructions (out-of-line EXTAB only)			
zero or more catch, cleanup, or FESPEC descriptors (out-of-line EXTAB only)			
32-bit NULL terminator (out-of-line EXTAB only)			

Personality Routines

The C7000 has four ABI-specified personality routines. The first three have the same format as the ARM EABI. Table 9-1 specifies the personality routines and their PR indexes.

Table 9-1. C7000 TDEH Personality Routines

PR Index (bits 27-24)	Personality	Routine Name	Unwind Instructions	Width of Scope Fields	Notes
0000	PR0 (Su16)	__c7xabi_unwind_cpp_pr0	up to 3 one-byte instructions	16	
0001	PR1 (Lu16)	__c7xabi_unwind_cpp_pr1	unlimited one-byte instructions	16	
0010	PR2 (Lu32)	__c7xabi_unwind_cpp_pr2	unlimited one-byte instructions	32	Must be used if 16-bit scope fields won't reach
0011	PR3	__c7xabi_unwind_cpp_pr3	24 bits	16	Optimized C7x-specific unwinding format

When using compact model EXTAB entries, a relocatable file must explicitly indicate which routines it depends on by including a reference from the EXTAB's section to the corresponding personality routine symbol, in the form of a `R_C7X_NONE` relocation.

9.5 Unwinding Instructions

Unwinding a frame is performed by simulating the function's epilog. Any operation that may be performed in a function's epilog needs to be encoded in the EXTAB entry so that the stack unwinder can simulate it.

The unwinding instructions make assumptions about the C7000 stack layout. In particular, the TDEH ordering for callee-saved registers as described in Section 4.4.2 is always assumed.

9.5.1 Common Sequence

Abstractly, all unwinding sequences take the following form:

1. Restore SP. SP := SP + constant
2. (Optional) Restore RP from a callee-save register.
3. (Optional) Restore callee-save registers. reg1 := SP[0]; reg2 := SP[-1]; and so on
4. Return through RP.

Step 1: Restore SP

An actual epilog does not restore SP until after the callee-save registers are restored, but because stack unwinding is a virtual operation, the simulated unwinding of TDEH may perform the SP restore first. This simplifies the restoration of the other callee-save registers.

SP is restored by incrementing by a constant. The encoded constant does not include the size of the callee-save area.

Step 2: Restore RP

The return address must be in RP before the return occurs. If it is stored in a callee-save register (say R*n*), then RP needs to be restored from R*n* before step 3 restores R*n* itself.

Step 3: Restore Registers

Abstractly, the callee-save registers are restored in TDEH order (Section 4.4.2) starting with the location pointed to by (the old) SP and moving to lower addresses.

Vector registers are saved, and therefore restored, as vectors of bytes, regardless of their actual contents.

Step 4: Return

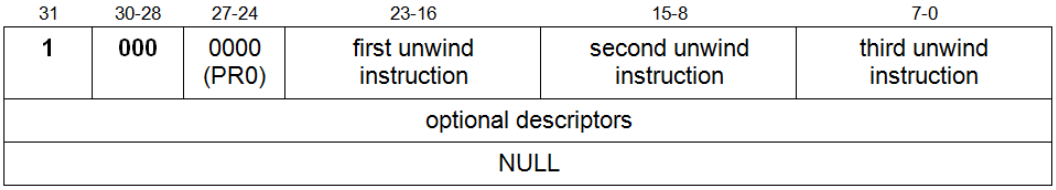
Every unwinding sequence ends with an implicit or explicit RET, which indicates that unwinding is complete for the current frame.

9.5.2 Byte-Encoded Unwinding Instructions

Personality routines PR0, PR1, and PR2 use a byte-encoded sequence of instructions to describe how to unwind the frame. The first few instructions are packed into the three remaining bytes of the first word of the EXTAB; additional instructions are packed into subsequent words. Unused bytes in the last word are filled with “RET” instructions.

Although the instructions are byte-encoded, they are always packed into 32-bit words starting at the MSB. As a consequence, the first unwinding instruction will not be at the lowest-addressed byte in little-endian mode.

Personality routine PR0 allows at most three unwinding instructions, all of which are stored in the first EXTAB word. If there are more than three unwinding instructions, one of the other personality routines must be used.



For PR1 and PR2, bits 23-16 encode the number of extra 32-bit words of unwinding instructions, which can be 0.

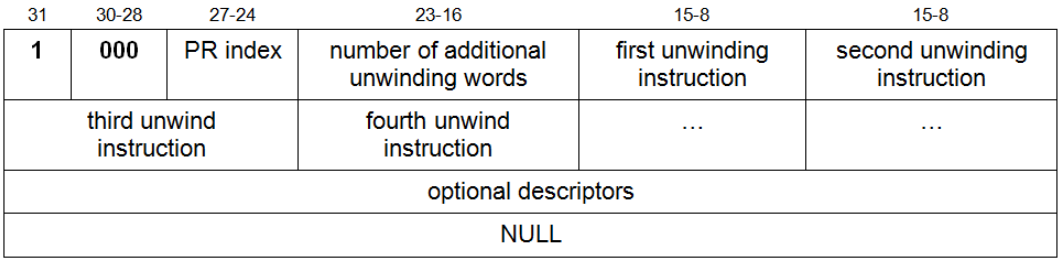


Table 9-2 summarizes the unwinding instruction set. Each instruction is described in more detail after the table.

Table 9-2. Stack Unwinding Instructions

Encoding	Instruction	Description
0kkk kkkk	SP += (k << 3) + 8	Increment SP by a small constant
1110 0000 kkkk kkkk …	SP += (ULEB128 << 3) + 0x408	Increment SP by a ULEB128-encoded constant

Table 9-2. Stack Unwinding Instructions (continued)

Encoding	Instruction	Description
1110 0001	CANTUNWIND	Function cannot be unwound, but might catch exceptions
10xx xxxx xxxx xxxx	POP bitmask	POP one or more registers (x != 0)
1101 rrrr	restore RP	RP := register r (R != RP)
1101 0000	RET	Unwinding is complete for this frame

All other bit patterns are reserved.

The rest of this section details the interpretation of the unwinding instructions.

Small Increment

7	6	5	4	3	2	1	0
0	**k**	k	k	k	k	k	k

SP += (k << 3) + 8

The value of 'k' is extracted from the lower 7 bits of the encoding, then scaled by 3 bits since the stack is 8-byte aligned, and biased by 8 since there is no need to encode a 0 increment. This instruction can increment the SP by a value in the range 0x8 to 0x400, inclusive. Increments in the range 0x408 to 0x800 should be done with two of these instructions.

Large Increment

7	6	5	4	3	2	1	0
1	**1**	**1**	**0**	**0**	**0**	**0**	**0**
k	k	k	k	k	k	k	k
…							

SP += (ULEB128 << 3) + 0x808

The value 'ULEB128' is ULEB128-encoded in the bytes following the 8-bit opcode. This instruction can increment the SP by a value of 0x808 or greater. Increments less than 0x808 should be done with one or two "Small Increment" instructions.

CANTUNWIND

7	6	5	4	3	2	1	0
1	**1**	**1**	**1**	**0**	**0**	**0**	**1**

This instruction indicates that the function cannot be unwound, usually because it is an interrupt function. However, an interrupt function can still have try/catch code, so `EXIDX_CANTUNWIND` is not appropriate.

POP Bitmask

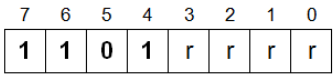
7	6	5	4	3	2	1	0
1	**0**	**X**	B14	VB14	B15	VB15	RP
A15	A14	A13	A12	A11	A10	A9	A8

This two-byte instruction indicates that up to thirteen callee-save registers should be popped from the virtual stack, as specified by the bitmask. Registers must be restored in the same order they appear in the TDEH ordering.

For callee-save vector registers VB14 and VB15, a function may save either the entire register or only the (64-bit) scalar part, but not both. Therefore at most one of the bits corresponding to VB14 and B14 are set; similarly for V15 and B15.

When any registers are popped using the "POP Bitmask" instruction, the SP is first implicitly incremented by the size of the saved registers represented by the bitmask. This is in addition to any explicit SP increment instruction(s).

Restore RP



RP := r

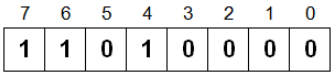
This instruction restores RP from the contents of ‘r’. This must be performed before any “POP” instruction that overwrites the register. The 4-bit nibble ‘r’ represents the encoding of a callee-save register. Since the value of RP can never be restored from VB14/15 or B14/15, a 4-bit encoding is not defined for VB14, VB15, B14 or B15.

The 4-bit register encoding is as follows:

Table 9-3. Register Encoding in Unwinding Instructions

Encoding	Register
0000	RP
0001	A15
0010	A14
0011	A13
0100	A12
0101	A11
0110	A10
0111	A9
1000	A8
1001	Reserved
1010	Reserved
1011	Reserved
1100	Reserved
1101	Reserved
1110	Reserved
1111	Reserved

RET

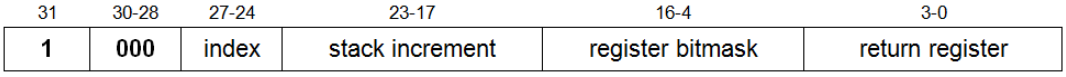


This instruction encodes a simulated return, indicating that unwinding is complete for this frame. Note that the encoding is the same as “Restore RP” but with the source register indicated as RP itself.

Every sequence of unwinding instructions ends with an explicit or an implicit "RET." This instruction may be omitted from the explicit unwinding instructions, and the unwinder will implicitly add it.

9.5.3 24-bit Unwinding Encoding

PR3 uses an optimized inline encoding intended to cover the majority of functions.



The stack increment is similar to the byte-encoded small constant increment, but it is not biased by 8. SP is incremented by (value << 3).

The return register field encodes the register in which the return address is stored, using the encoding from Table 9-3. If this register is any register other than RP itself, RP should be restored from this register before executing the “POP” operation described in the next paragraph.

The bitmask is interpreted as in the byte-encoded “POP Bitmask” instruction, including the implicit increment of SP.

9.6 Descriptors

If any local objects need to be destroyed, or if the exception is caught by this function, the EXTAB will contain descriptors describing what to do and for which exception types.

If present, the descriptors follow the unwinding instructions. The format of the descriptors is a sequence of descriptor entries followed by a 32-bit zero (NULL) word. Each descriptor starts with a scope, which identifies what kind of descriptor it is and specifies a program address range within which the descriptor applies. Additional descriptor-specific words follow the scope.

Descriptors shall be listed in depth-first order so that all of the applicable descriptors can be handled in one pass.

The general form for an EXTAB entry with descriptors is:

31	30-28	27-24	23-0
1	**000**	PR index	unwinding instructions
zero or more additional 32-bit words of unwinding instructions			
zero or more catch, cleanup, or FESPEC descriptors			
32-bit NULL terminator			

9.6.1 Encoding of Type Identifiers

Catch descriptors (Section 9.6.4) and FESPEC descriptors (Section 9.6.5) encode type identifiers to be used in matching the type of thrown objects against catch clauses and exception specifications. These fields are encoded to reference the type_info object corresponding to the specified type, using a PREL30 relocation.

9.6.2 Scope

The scope identifies the descriptor type and specifies a program address range in which an action should take place. The range corresponds to a potentially-throwing call site. The unwinder looks through the descriptor list for descriptors containing a scope containing the call site; once a match is found, the descriptor is activated.

The scope encodes a program address range by specifying an offset from the starting address of the function and a length, both in bytes. If the length and offset each fit in a 15-bit unsigned field, the scope uses the short form encoding and the rest of the EXTAB entry can be encoded for PR0, PR1, or PR3. If either the length or offset exceed 15-bits, the scope uses the long form encoding and PR2 must be used.

Short Form Scope

31-17	16	15-1	0
length	**X**	offset	**Y**
data for descriptor			

The short form scope may not be used with PR2 (Lu32).

Long Form Scope

31-1	0
length	**X**
offset	**Y**
data for descriptor	

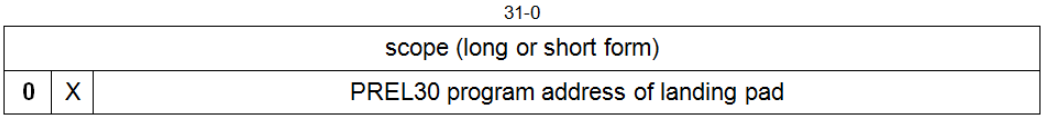
If the length or offset require the long form scope, personality routine PR2 (Lu32) must be used.

Bits X and Y in the scope encodings indicate the kind of descriptor that follows the scope:

X	Y	Descriptor
0	0	cleanup descriptor
1	0	catch descriptor
0	1	function exception specification (FESPEC) descriptor

9.6.3 Cleanup Descriptor

Cleanup descriptors control destruction of local objects which are fully constructed and are about to go out of scope, and thus must be destructed.

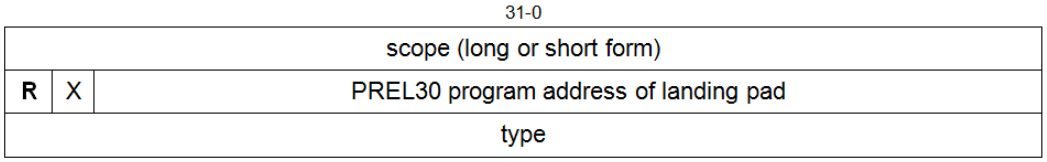


The cleanup descriptor simply contains a single pointer to a cleanup code block containing one or more calls to destructor functions.

9.6.4 Catch Descriptor

Catch descriptors control which exceptions are caught, and when. A function may have several catch clauses which each apply to a different subset of potentially-throwing function calls. One call site can have multiple catch descriptors, each with a different type.

If the type in the catch descriptor matches the thrown type, control is transferred to the *landing pad*, which is just a code fragment representing a catch block. Catch blocks implement catch clauses in the user's code. These blocks are only executed when an exception actually gets thrown. These blocks are generated for a function when the rest of the function is generated, and execute in the same stack frame as the function, but may be placed in a different section.



If bit R is 1, the type of the catch clause is a reference type represented by TYPE. If bit R is 0, the type is not a reference type.

The type field is either a reference to a `type_info` object (relocated via a R_C7X_PREL30 relocation), or one of two special values.

- The special value 0xFFFFFFFF (-1) means the "any" type [catch(...)].
- The special value 0xFFFFFFFE (-2) means the "any" type [catch(...)], and also indicates that the personality routine should immediately return `_URC_FAILURE`. In this case, the landing pad address should be set to 0. This idiom may be used to prevent exception propagation out of the code covered by that scope.

9.6.5 Function Exception Specification (FESPEC) Descriptor

FESPEC descriptors enforce throw() declarations in the user's code. If a throw declaration is used, a FESPEC descriptor will be created for this function to ensure that only those types listed are thrown. If a type not listed is thrown, the unwinder will typically call `std::unexpected`.

31-0

scope (long or short form)		
D	number of type info pointers	
reference to type_info object		
reference to type_info object		
…		
0	X	(if D == 1) PREL30 program address of landing pad

The first word of the descriptor consists of a 1-bit 'D' field and a 31-bit unsigned integer, which specifies the number of type_info fields that follow.

If bit D is 1, the `type_info` list is followed by a 32-bit word containing a PREL30 program address of a code fragment which is called if no type in the list matches the thrown type. Bit 31 of this word is set to 0.

If bit D is 0, and no type in the list matches the thrown type, the unwinding code should call `__cxa_call_unexpected`. If any descriptors match this form, the EXTAB section must contain a R_C7X_NONE relocation to `__cxa_call_unexpected`.

9.7 Special Sections

All of the exception handling tables are stored in two sections. The EXIDX table is stored in a section called `.c7xabi.exidx` with type `SHT_C7X_UNWIND`. The linker must combine all the input `.c7xabi.exidx` sections into one contiguous `.c7xabi.exidx` output section, maintaining the same relative order as the code sections they refer to. In other words, the entries in the EXIDX table are sorted by address. Each EXIDX section in a relocatable file must have the `SHF_LINK_ORDER` flag set to indicate this requirement.

The EXTAB is stored in a section called `.c7xabi.extab`, with type `SHT_PROGBITS`. The EXTAB is not required to be contiguous and there is no ordering requirement.

Exception tables can be linked anywhere in memory. For dynamically linked modules, the tables should be placed in the same segment as the code in order to facilitate position independence.

9.8 Interaction With Non-C++ Code

9.8.1 Automatic EXIDX entry generation

Functions which do not have an EXIDX entry will have one created for them automatically by the linker, so functions from a library compiled without exception-handling enabled (such as a C-only library) can be used in an application which uses TDEH. Automatically-generated entries will be `EXIDX_CANTUNWIND`, so if a function compiled without exception-handling support enabled calls a function which does propagate an exception, `std::terminate` will be called and the application will halt.

9.8.2 Hand-coded Assembly Functions

Hand-coded assembly functions can be instrumented to handle or propagate exceptions. This is only necessary if the function calls a function which might propagate an exception, and this exception must be propagated out of the assembly function. The user must create an appropriate EXIDX entry and an EXTAB containing at least the unwinding instructions.

9.9 Interaction with System Features

Shared Libraries

The exception-handling tables can propagate exceptions within an executable or shared libraries. Propagating an exception across calls between different load modules requires help from the OS.

Overlays

C++ functions which may propagate exceptions must not be part of an overlay. The EXIDX lookup table does not handle overlay functions, and it could not distinguish between the different possible functions at a particular location.

Interrupts

Interrupts, hardware exceptions, and OS signals cannot be handled directly by exceptions.

Because interrupt functions could happen anywhere, we cannot support propagating exceptions from interrupt functions. All interrupt functions will be `EXIDX_CANTUNWIND`. However, interrupt functions can call functions which might themselves throw exceptions, and thus interrupt functions must be in the EXIDX table and may have descriptors, but will never have unwinding instructions.

Applications that wish to use an exception to represent interrupts must arrange for the interrupt to be caught with an interrupt function, which must set a global volatile object to indicate that the interrupt has occurred, and then use the value of that variable to throw an exception after the interrupt function has returned.

If an OS provides signal, exceptions representing signals must be handled similarly.

9.10 Assembly Language Operators in the TI Toolchain

These implementation details pertain to the TI toolchain and are not part of the ABI.

The TI compiler uses special built-in assembler functions to indicate to the assembler that certain expressions in the exception-handling tables should get special processing.

- `$EXIDX_FUNC` — The argument is a function address to be encoded using the PREL30 representation.
- `$EXIDX_EXTAB` — The argument is an EXTAB label to be encoded using the PREL30 representation.
- `$EXTAB_LP` — The argument is a landing pad label to be encoded using the PREL30 representation.
- `$EXTAB_RTTI` — The argument is the label for the unique type_info object representing a type. (These objects are generated for run-time type identification.) The field is relocated with the PREL30 relocation.

DWARF

The C7000 uses the DWARF Debugging Information Format Version 4, also known as DWARF4, to represent information for a symbolic debugger in object files. DWARF4 is documented in http://www.dwarfstd.org/doc/DWARF4.pdf. This chapter augments that standard by specifying parts of the representation that are specific to the C7000.

In the compilation unit header, the `address_size` field specifies the size in bytes of an address on the target architecture. For C7000, the value of `address_size` shall be 8, so that 64-bit addresses can be represented.

Topic **Page**

10.1 DWARF Register Names

DWARF4 refers to registers using register name operators, as described in Section 2.6.1.1.2 of the DWARF4 standard. The operand of a register name operator is a register number representing an architecture register. Table 10-1 defines the mapping from DWARF4 register numbers to C7000 registers.

Table 10-1. DWARF4 Register Numbers for C7000

DWARF Register Range	C7X ISA Register Range	Description
0-15	A0-A15	A-side global 64-bit scalar registers
16-23	AL0-AL7	A-side LS-unit local 64-bit scalar registers
24-31	AM0-AM7	A-side MC-unit local 64-bit scalar registers
32-47	B0-B15	B-side global 64-bit scalar registers
48-63	VB0-VB15	B-side global 512-bit vector registers
64-71	BL0-BL7	B-side LS-unit local 64-bit scalar registers
72-79	VBL0-VBL7	B-side LS-unit local 512-bit vector registers
80-87	BM0-BM7	B-side MC-unit local 64-bit scalar registers
88-95	VBM0-VBM7	B-side MC-unit local 512-bit vector registers
96-111	D0-D15	A-side D-unit local 64-bit scalar registers
112-119	P0-P7	B-side 64-bit vector predicate registers
120-127		Reserved
128-131	CUCR0-CUCR3	C-unit 512-bit control registers
132-4095		Reserved
4096	CPUID	CPU ID register
4097	PMR	Power management register
4098	DNUM	DSP core number register
4099	TSC	Time-stamp counter register
4100	TSR	Task state register
4101	RP	Return pointer register
4102	BPCR	Branch Predictor Control Register
4103	FPCR	Floating-point control register
4104	FSR	Flag status register
4105	ECLMR	Event claim register
4106	EASGR	Event assign register
4107	EPRI	Event priority register
4108	EER	Event enable register
4109	EESET	Event enable set register
4110	EECLR	Event enable clear register
4111	DEPR	Debug event priority register
4112	EFR	Event flag register
4113	EFSET	Event flag set register
4114	EFCLR	Event flag clear register
4115	IESET	Internal exception event set register
4116	ESTP_SS	Event service table pointer register, secure supervisor
4117	ESTP_S	Event service table pointer register, supervisor
4118	ESTP_GS	Event service table pointer register, guest supervisor
4119	EDR	Event dropped register
4120	ECSP_SS	Event context save pointer register, secure supervisor
4121	ECSP_S	Event context save pointer register, supervisor
4122	ECSP_GS	Event context save pointer register, guest supervisor
4123	TCSP	Task context save pointer
4124	RXMR_SS	Returning execution mode register, secure supervisor

Table 10-1. DWARF4 Register Numbers for C7000 (continued)

DWARF Register Range	C7X ISA Register Range	Description
4125	RXMR_S	Returning execution mode register, supervisor
4126	AHPEE	Highest priority enabled event register, currently in service
4127	PHPEE	Highest priority enabled event register, pending
4128	IEBER	Internal event broadcast enable register
4129	IERR	Internal exception report register
4130	IEAR	Internal exception address register
4131	IESR	Internal exception status register
4132	IEDR	Internal exception data register
4133	TCR	Test count register
4134	TCCR	Test count config register
4135	GMER	Guest mode enable register
4136	UMER	User mask enable register
4137	SPBR	Stack pointer boundary register
4138-4141	LTBR0-LTBR3	Lookup table base address registers
4142-4145	LTCR0-LTCR3	Lookup table configuration registers
4146	LTER	Lookup table enable register
4147	DBGCTXT	Debug context (overlay) register
4148	PC_PROF	PC profile register
4149	GPLY	Galois polynomial register
4150	GFPGFR	Galois field polynomial generator function register
4151	GTSC	Global time stamp counter register
4152	STSC	Shadow time stamp counter register
4153	UFCMR	User flag clear mask register
4154-4326		Reserved
4327	PSA3	Streaming address predicate register
4328	PSA2	Streaming address predicate register
4329	PSA1	Streaming address predicate register
4330	PSA0	Streaming address predicate register
4331	STRACNTR3	Streaming address generator counter register
4332	STRACNTR2	Streaming address generator counter register
4333	STRACNTR1	Streaming address generator counter register
4334	STRACNTR0	Streaming address generator counter register
4335	STRACR3	Streaming address generator configuration register
4336	STRACR2	Streaming address generator configuration register
4337	STRACR1	Streaming address generator configuration register
4338	STRACR0	Streaming address generator configuration register
4339	SA3	Streaming address offset register
4340	SA2	Streaming address offset register
4341	SA1	Streaming address offset register
4342	SA0	Streaming address offset register
4343-4344		Reserved
4345	SE1	Streaming engine data register
4346	SE0	Streaming engine data register
4347	SCRB	Scoreboard bits register
4348	LCNTFLG	16-bit predicate flags register
4349	OLCNT	Outer loop counter initial value register
4350	ILCNT	Inner loop counter register

Table 10-1. DWARF4 Register Numbers for C7000 (continued)

DWARF Register Range	C7X ISA Register Range	Description
4351	PC	Program counter register

10.2 Call Frame Information

Debuggers often need to be able to view and modify the state of any subroutine activation on the call stack. An activation's state is comprised of values that were stored in registers and on the stack during that activation. Section 6.4 of the DWARF4 standard defines a way for a debugger to progressively recreate a previous state by interpreting the instructions of a byte-coded language. Each activation is represented by a base address, called the Canonical Frame address (CFA), and a set of values corresponding to the contents of the machine's registers during that activation.

Both the definition of the CFA and the set of registers comprising the state are architecture-specific.

For the CFA, the C7000 ABI follows the convention suggested in the DWARF4 standard, defining it as the value of SP (D15) at the call site in the previous frame (that of the calling function).

For simplicity, and because the representation is efficient, the set of registers includes all the registers listed in Table 10-1, indexed by their DWARF register numbers from the first column.

There is not a distinct column in the state table for the virtual return address as suggested in Section 6.4.4 of the DWARF4 standard. In accordance with the calling conventions, the return address is represented by the RP column of the state table.

The state table includes registers that are not present on all C7000 ISAs. Therefore a situation may arise in which the ISA executing the program has registers that are not mentioned in the call frame information. In this situation, the interpreter should behave as follows:

- Callee-saved registers should be initialized to the same-value rule.
- All other registers should be initialized to the undefined rule.

10.3 Vendor Names

The `DW_AT_producer` attribute is used to identify the toolchain that produced an object file. The operand is a string that begins with a vendor prefix. The following prefixes are reserved for specific vendors:

- **TI** — C7000 Code Generation Tools from Texas Instruments
- **GNU** — The GNU Compiler Collection (GCC)

10.4 Vendor Extensions

The DWARF standard allows toolchain vendors to define additional tags, attributes, and base type attributes for representing information that is specific to an architecture or toolchain. TI has defined some of each. This section serves to document the ones that apply generally to the C7000 architecture.

Unfortunately the set of allowable values is shared among all vendors, so the ABI cannot mandate standard values to be used across vendors. The best we can do is ask producers to define their own vendor-specific tags and attributes with the same semantics (using the same values if possible), and ask consumers to use the `DW_AT_producer` attribute in order to interpret vendor-specific values that differ from toolchain to toolchain..

Table 10-2. TI Vendor-Specific Tags

Name	Value	Description
DW_TAG_TI_assign_register	0x4082	Specifies mapping from DWARF reg to hardware reg.
DW_TAG_TI_ioport_type	0x4083	Used for types declared as ioport.
DW_TAG_TI_branch	0x4088	Used in conjunction with DW_AT_low_pc to identify a branch location.
DW_TAG_TI_indirect_call	0x4092	Declares what functions a function may indirectly call.

Table 10-3. TI Vendor-Specific Attributes

Name	Value	Class	Description
DW_AT_TI_symbol_name	0x2001	string	The mangled name of the symbol as it appears in the object file.
DW_AT_TI_begin_file	0x2003	string	Source position information.
DW_AT_TI_begin_line	0x2004	constant	Source position information.
DW_AT_TI_begin_column	0x2005	constant	Source position information.
DW_AT_TI_end_file	0x2006	string	Source position information.
DW_AT_TI_end_line	0x2007	constant	Source position information.
DW_AT_TI_end_column	0x2008	constant	Source position information.
DW_AT_TI_return	0x2009	flag	Specifies that a branch is a return.
DW_AT_TI_call	0x200a	flag	Specifies that a branch is a call.
DW_AT_TI_version	0x200b	constant	Version of TI DWARF attributes.
DW_AT_TI_asm	0x200c	flag	Function is hand coded assembly.
DW_AT_TI_indirect	0x200d	flag	Branch is indirect.
DW_AT_TI_category	0x200f	string	Category of code, typically used to mark RTS routines.
DW_AT_TI_plt_entry	0x2012	flag	Function is a Procedure Linkage Table entry. Similar to DW_AT_trampoline.
DW_AT_TI_max_frame_size	0x2014	constant	Activation record size. Indicates the amount of stack space required for activation of the function, in bytes.

Table 10-4. TI Vendor-Specific Base Type Attributes

Name	Value	Description
DW_ATE_TI_complex_signed	0x90	Complex integer
DW_ATE_TI_complex_float	0x91	Complex floating point

Object Files (Processor Supplement)

The C7000 ABI is based on the ELF object file format. The base specification for ELF is comprised of Chapters 4 and 5 of the larger System V ABI specification. This chapter contains the C7000 processor-specific supplement for Chapter 4 (Object Files). Section numbers given are based on the 10 June 2013 version of the System V ABI drafts.

C7000 has up to a 64-bit address space, so it uses the 64-bit form of ELF.

Topic **Page**

11.1 Registered Vendor Names

The compiler toolsets create and use vendor specific symbols. To potential avoid conflicts TI encourages vendors to define and use vendor-specific namespaces. The list of currently registered vendors and their preferred short-hand name is given in Table 11-1.

Table 11-1. Registered Vendors

Name	Vendor
cxa, __cxa	C++ ABI namespace. Applies to all symbols specified by the C++ ABI.
c7xabi, __c7xabi	Common namespace for symbols specified by the C7000 EABI.
C7000	Common namespace for symbols specified by the C7000 EABI.
TI, __TI	Reserved for symbols specific to the TI toolchain. This also represents a composite namespace for all TI processor ABIs. (see the note following this table)
gnu, __gnu	Reserved for symbols specific to the GCC toolchain.

NOTE: This specification defines names for processor-specific section types, special sections, and so on. Where there is commonality among different TI processors, we name such entities using 'TI' rather than defining distinct names for each processor. For example, the section type for C initialization tables is `SHT_TI_INITINFO` for all TI processors, rather than `SHT_C7X_INITINFO` for C7000, `SHT_MSP_INITINFO` for MSP, and so on.

11.2 ELF Header

The ELF header provides a number of fields that guide interpretation of the file. Most of these are specified in the System V ELF specification. This section augments the base standard with specific details for the C7000.

- **e_ident** — The 16-byte ELF identification identifies the file as an object file and provides machine-independent data with which to decode and interpret the file's contents. Table 11-2 specifies the values to be used for C7000 object files.

Table 11-2. ELF Identification Fields

Index	Symbolic Value	Numeric Value	Comments
EI_MAG0		0x7f	per System V ABI
EI_MAG1		'E'	
EI_MAG2		'L'	
EI_MAG3		'F'	
EI_CLASS	ELFCLASS64	2	64-bit ELF
EI_DATA	ELFDATA2LSB	1	little-endian
	ELFDATA2MSB	2	big endian
EI_VERSION	EV_CURRENT	1	
EI_OSABI	ELFOSABI_NONE	0	No platform
	ELFOSABI_C7X_ELFABI	64	bare-metal platform
	ELFOSABI_C7X_LINUX	65	Linux platform
EI_ABIVERSION		0	

The `EI_OSABI` field shall be `ELFOSABI_NONE` unless overridden by the conventions of a specific platform. The bare-metal dynamic linking model and Linux are two such platforms (Section 1.1) that define specific values for this field.

A value other than `ELFOSABI_NONE` represents an assertion that the file conforms to the conventions of the particular ABI variant corresponding to the specified value. Only such files are valid for that specific platform. Objects can be built for platforms other than the specific variants defined by the ABI; these should be identified as `ELFOSABI_NONE`, representing the lack of any assertion. The determination of whether such a file is compatible with a given environment is independent of the ABI.

- **e_type** — There are currently no C7000-specific object file types. All values between `ET_LOPROC` and `ET_HIPROC` are reserved to future revisions of this specification.
- **e_machine** — An object file conforming to this specification must have the value `EM_TI_C7X` (145, 0x91).
- **e_entry** — The base ELF specification requires this field to be zero if an application does not have an entry point. Nonetheless, some applications may require an entry point of zero (for example, via the reset vector).

  A platform standard may specify that an executable file always has an entry point, in which case e_entry specifies that entry point, even if zero.
- **e_flags** — This member holds processor-specific flags associated with the file. There is one C7000-specific flag.

Table 11-3. Processor-Specific File Header Flags

Name	Value	Comment
EF_C7X_REL	0x1	File contains static relocation information

The `EF_C7X_REL` flag is to indicate the presence of static relocation information in an executable file (`ET_EXEC`) or shared object (`ET_DYN`). A shared object with static relocation information is called a relocatable module and is generally used for libraries that can be linked either statically or dynamically.

11.3 Sections

There are no processor-specific special section indexes defined. All processor-specific values are reserved to future revisions of this specification.

11.3.1 Section Types

The ELF specification reserves section types 0x70000000 and higher for processor-specific values. TI has split this space into two parts: values from 0x70000000 through 0x7EFFFFFF are processor-specific, and values from 0x7F000000 through 0xFFFFFFFF are for TI-specific sections common to multiple TI architectures. The combined set is listed in Table 11-4.

Not all these section types are used in the C7000 ABI. Some are specific to the TI toolchain but outside the ABI, and some are used by TI toolchains for architectures other than C7000. They are documented here for completeness, and to reserve the tag values.

Table 11-4. C7000 Section Types

Name	Value	Comment
SHT_C7X_UNWIND	0x70000001	Unwind function table for stack unwinding
SHT_C7X_PREEMPTMAP	0x70000002	DLL dynamic linking preemption map
SHT_C7X_ATTRIBUTES	0x70000003	Object file compatibility attributes
SHT_TI_ICODE	0x7F000000	Intermediate code for link-time optimization
SHT_TI_XREF	0x7F000001	Symbolic cross reference information
SHT_TI_HANDLER	0x7F000002	Reserved
SHT_TI_INITINFO	0x7F000003	Compressed data for initializing C variables
SHT_TI_PHATTRS	0x7F000004	Extended program header attributes
SHT_TI_SH_FLAGS	0x7F000005	Extended section header attributes
SHT_TI_SYMALIAS	0x7F000006	Symbol alias table
SHT_TI_SH_PAGE	0x7F000007	Per-section memory space table

SHT_C7X_UNWIND identifies a section containing unwind function table for stack unwinding. See Section 9.7 for details.

SHT_C7X_PREEMPTMAP identifies a section containing a C7000 DLL dynamic linking preemption map.

SHT_C7X_ATTRIBUTES identifies a section containing object compatibility attributes, specified in Section 12.1.

SHT_TI_ICODE identifies a section containing a TI-specific intermediate representation of the source code, used for link-time recompilation and optimization.

SHT_TI_XREF identifies a section containing symbolic cross-reference information.

SHT_TI_HANDLER is not currently used.

SHT_TI_INITINFO identifies a section containing compressed data for initializing C variables. This section contains a table of records indicating source and destination addresses, and the data itself, usually in the compressed form. See Chapter 13.

SHT_TI_PHATTRS identifies a section containing additional properties for program segments in an executable or shared object file. See Chapter 14.

SHT_TI_SH_FLAGS identifies a section containing a table of TI-specific section header flags.

SHT_TI_SYMALIAS identifies a section containing a table that defines symbols as being equivalent to other, possibly externally defined, symbols. The TI linker uses the table to eliminate trivial functions that simply forward to other functions.

SHT_TI_SH_PAGE is used only on targets that have distinct, possibly overlapping, address spaces (“pages”). The section contains a table that associates other sections with page numbers. This section type is not used on C7000.

11.3.2 Section Attribute Flags

There are no processor-specific section attribute flags defined. All processor-specific values are reserved to future revisions of this specification.

11.3.3 Subsections

C7000 object files use a section naming convention that provides improved granularity while retaining the convenience of default rules for combining sections at link time. A section whose name contains a colon is called a subsection. Subsections behave as normal sections in all respects, but their name guides the linker when combining sections into output files. The root name of a subsection is the name up to, but not including, the colon. The suffix includes all characters following the colon. By default, the linker combines all sections with matching roots into a single section with that name. For example, `.text`, `.text:func1`, and `.text:func2` are combined into a single section called `.text`. The user may be able to override this default behavior in toolchain-specific ways.

If there are multiple colons, section combination proceeds recursively from the right-most colon. For example. unless the user specifies otherwise, the default rules combine `.bss:func1:var1` and `.bss:func1:var2`, which then combine into `.bss`.

Subsections whose root names match special sections have the same ABI-defined properties as the section they match, as defined in Section 11.3.4. For example `.text:func1` is an instance of a `.text` section.

11.3.4 Special Sections

The System V ABI, along with other base documents and other sections of this ABI, defines several sections with dedicated purposes. Table 11-5 consolidates dedicated sections used by the C7000 and groups them by functionality.

Section names are not mandated by the ABI. Special sections should be identified by type, not by name. However, interoperability among toolchains can be improved by following these conventions. For example, using these names may decrease the likelihood of having to write custom linker commands to link relocatable files built by different compilers.

The ABI does mandate that a section whose name does match an entry in the table must be used for the specified purpose. For example, the compiler is not required to generate code into a section called `.text`, but it is not allowed to generate a section called `.text` containing anything other than code.

Table 11-5. C7000 Sections

Prefix	Type	Attributes	Notes
Code Sections			
.text	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR	
.plt	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR	
Data Sections			
.bss	SHT_NOBITS	SHF_ALLOC+SHF_WRITE	
.data	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE	
.const	SHT_PROGBITS	SHF_ALLOC	
Dynamic Data Sections			
.got	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE	
Exception Handling Data Sections			
.c7xabi.exidx	SHT_C7X_UNWIND	SHF_ALLOC+SHF_LINK_ORDER	
.c7xabi.extab	SHT_PROGBITS	SHF_ALLOC	
Initialization and Termination Sections			
.init	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR	
.fini	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR	
.preinit_array	SHT_PREINIT_ARRAY	SHF_ALLOC+SHF_WRITE	
.init_array	SHT_INIT_ARRAY	SHF_ALLOC+SHF_WRITE	
.fini_array	SHT_FINI_ARRAY	SHF_ALLOC+SHF_WRITE	
ELF Structures			
.rel	SHT_REL	None	
.rela	SHT_RELA	None	
.symtab	SHT_SYMTAB	None	
.symtab_shndx	SHT_SYMTAB_SHNDX	None	
.strtab	SHT_STRTAB	SHF_STRINGS	
.shstrtab	SHT_STRTAB	SHF_STRINGS	
.note	SHT_NOTE	None	
Dynamic Loading Structures			
.dynamic	SHT_DYNAMIC	SHF_ALLOC	1
.dynsym	SHT_DYNSYM	SHF_ALLOC	1
.dynstr	SHT_STRTAB	SHF_ALLOC+SHF_STRINGS	1
.hash	SHT_HASH	SHF_ALLOC	1
.interp	SHT_PROGBITS	None	
Build Attributes			
.c7xabi.attributes	SHT_C7X_ATTRIBUTES	None	
Symbolic Debug Sections			
.debug_*	SHT_PROGBITS	None	
Symbol Versioning Sections			
.gnu.version	SHT_GNU_versym	SHF_ALLOC	1
.gnu.version_d	SHT_GNU_verdef	SHF_ALLOC	1
.gnu.version_r	SHT_GNU_verneed	SHF_ALLOC	1
Sections Reserved for Thread-Local Storage			
.tbss	SHT_NOBITS	SHF_ALLOC+SHF_WRITE+SHF_TLS	2
.tdata	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE+SHF_TLS	2
.tdata1	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE+SHF_TLS	2

Table 11-5. C7000 Sections (continued)

Prefix	Type	Attributes	Notes
TI Toolchain-Specific Sections			
.stack	SHT_NOBITS	SHF_ALLOC+SHF_WRITE	
.sysmem	SHT_NOBITS	SHF_ALLOC+SHF_WRITE	
.cio	SHT_NOBITS	SHF_ALLOC+SHF_WRITE	
.cinit	SHT_TI_INITINFO	SHF_ALLOC	
.binit	SHT_PROGBITS	SHF_ALLOC	
.const:handler_table	SHT_TI_HANDLER	SHF_ALLOC	
.ovly	SHT_PROGBITS	SHF_ALLOC	
.TI.crctab	SHT_PROGBITS	SHF_ALLOC	
.TI.icode	SHT_TI_ICODE	none	
.TI.phattrs	SHT_TI_PHATTRS	none	
.TI.preempt.map	SHT_C7X_PREEMPTMAP	SHF_ALLOC	
.TI.xref	SHT_TI_XREF	None	
.TI.section.flags	SHT_TI_SH_FLAGS	none	
.TI.symbol.alias	SHT_TI_SYMALIAS	none	
.TI.section.page	SHT_TI_SH_PAGE	none	
Sections Unused by the C7000 EABI			
.data1	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE	
.rodata1	SHT_PROGBITS	SHF_ALLOC	
.comment	SHT_PROGBITS	none	
.line	SHT_PROGBITS	none	

Notes:

1. Whether the `.dynamic` section and related sections are allocated into memory is platform-specific.
2. The ABI does not currently define a mechanism for thread-local storage (TLS). These names are reserved for future use.

The sections under the heading *TI Toolchain-Specific Sections* are used by the TI toolchain in various toolchain-specific ways. The ABI does not mandate the use of these sections (although interoperability encourages their use), but it does reserve these names.

The sections under the *Unused* heading are sections that are specified by the System V ABI, but not used or defined under the C7000 ABI.

11.3.5 Section Alignment

Sections containing C7000 instructions must be at least 64-byte aligned, and padded to 64-byte boundaries. The latter requirement is to avoid misinterpreting adjacent data as a fetch packet header.

Platform standards may set a limit on the maximum alignment that they can guarantee (normally the virtual memory page size).

11.4 Symbol Table

There are no processor-specific symbol types or symbol bindings. All processor-specific values are reserved to future revisions of this specification.

The C7000 ABI follows the ELF specification with respect to global and weak symbol definitions, and the meaning of symbol values.

11.4.1 Symbol Types

This specification adheres to the ARM ELF specification with respect to symbol types, namely:

- All code symbols exported from an object file (symbols with binding `STB_GLOBAL`) shall have type `STT_FUNC`.
- All extern data objects shall have type `STT_OBJECT`. No `STB_GLOBAL` data symbol shall have type `STT_FUNC`.
- The type of an undefined symbol shall be `STT_NOTYPE` or the type of its expected definition.
- The type of any other symbol defined in an executable section can be `STT_NOTYPE`.

11.4.2 Common Block Symbols

As described in the ELF specification, symbols with type `STT_COMMON` are allocated by the linker. The linker allocates such symbols into an uninitialized data section, typically `.bss`.

11.4.3 Symbol Names

A symbol that names a C or assembly language entity should have the name of that entity. For example, a C function called func generates a symbol called func. (There is no leading underscore as was the case in the former COFF ABI). Symbol names are case sensitive and are matched exactly by linkers.

C++ identifiers are mangled to encode type information, as described by the IA-64 C++ ABI. To support vectors in C++, vendor extended types are used as described by the IA-64 C++ ABI with the constraint that all vectors are mangled using their canonical (double leading underscore) name. For example, int8 will mangle as u6__int8. Similarly, __cdouble2 will mangle as u10__cdouble2.

The C7000 compiler follows the following naming convention for temporary symbols:

- Parser generated symbols are prefixed with `$P$`
- Optimizer generated symbols are prefixed with `$O$`
- Codegen generated symbols are prefixed with `$C$`

11.4.4 Reserved Symbol Names

The following symbols are reserved to this and future revisions of this specification:

- Local symbols (`STB_LOCAL`) beginning with `$`
- Global symbols (`STB_GLOBAL`, `STB_WEAK`) beginning with any of the vendor names listed in Table 11-1.
- Global symbols (`STB_GLOBAL`, `STB_WEAK`) ending with either `$$Base` or `$$Limit`
- Symbols matching the pattern `${Tramp}${I|L|S}[$PI]$$symbol`
- Compiler generated temporary symbols beginning with `$P$` , `$O$`, `$C$` (as described in Section 4.4)

11.4.5 Mapping Symbols

Mapping symbols are local symbols that serve to classify program data. Currently the ABI does not specify any behavior that uses mapping symbols. Nevertheless, the following two names are reserved for future use: `$code`, and `$data`.

11.5 Relocation

The ELF relocations for C7000 are defined such that the all information needed to perform the relocation is contained in the relocation entry, the object field, and the associated symbol. The linker does not need to decode instructions, beyond unpacking the object field, to perform the relocation.

Relocations are specified as operating on a relocatable field. Roughly speaking, the relocatable field is the bits of the program image that are affected by the relocation. The field is defined in terms of an addressable container whose address is given by the `r_offset` field of the relocation entry. The field's size and position within to the container, as well as the computation of the relocated value, are specified by the relocation type. The relocation operation consists of extracting the relocatable field, performing the operation, and re-inserting the resultant value back into the field.

ELF relocations can be of type `Elf_Rela` or `Elf_Rel`. The `Elf_Rela` entries contain an explicit addend which is used in the relocation calculation. Entries of type `Elf_Rel` use the relocatable field itself as the addend. Certain relocations are identified as `Elf_Rela` only. For the most part these correspond to values split between multiple instructions or extension words, where the resultant value depends on carry propagation from lower bits that are not available in the field. Where `Elf_Rela` is specified, an implementation must honor this requirement. Where not specified, an implementation may choose to use `Elf_Rel` or `Elf_Rela`.

The C7000 has several PC-relative addressing modes. Instructions that use these modes encode the effective address as an offset between the address of their fetch packet and the target address.

The C7000 ISA has several encodings in which immediate values, including addresses, use a *constant extension* (CE) field in the execute packet of which they are part. In some cases the value is split between two or three fields: one in the instruction itself, and one or two in separate constant extension field(s). Even though the multiple fields logically encode a single value, each requires a distinct relocation entry. Such relocations are always `Elf_Rela`.

When a PC-relative offset is split between two or more fields, the fields are always in the same execute packet but may be in different fetch packets. After relocation, each field that contains part of the encoded offset should be relative to the fetch packet of the instruction itself, not the fetch packet(s) that contain the CE fields. This presents a challenge because the 'P' value used to compute the relocation is necessarily the CE's fetch packet – there is no information in the CE's encoding to indicate the address of the instruction itself. To solve this, the offset between the CE and the instruction is encoded in the addend field of the CE's relocation. In this way the computed offset is adjusted to be relative to the instruction's fetch packet and not the CE's.

11.5.1 Relocation Types

Relocations are described using two tables. Table 11-6 gives numeric values for the relocation types and summarizes the computation of the relocated value. Following the table is a description of the relocation types and examples of their use. Table 11-7 is a reference table that describes, for each type, the exact computation, including extraction and insertion of the relocation field, overflow checking, and any scaling or other adjustments.

The following notations are used in Table 11-6:

- **S** — The value of the symbol associated with the relocation, specified by the symbol table index contained in the `r_info` field in the relocation entry
- **A** — The addend used to compute the value of the relocatable field. For `Elf_Rel` relocations, A is encoded into the relocatable field according to Table 11-7. For `Elf_Rela` relocations, A is given by the `r_addend` field of the relocation entry.
- **PC** — The address of the container containing the field being relocated.
- **FP(x)** — The address of the fetch packet containing the instruction or field at address x; that is: FP(x) := x & ~0x3F.
- **P** — The fetch packet containing the instruction or field being relocated; that is: P := FP(PC)

Table 11-6. C7000 Relocation Types

Name	Value	Operation	Constraints
R_C7X_NONE	0	none	
R_C7X_PCR16	4	S + A – P	
R_C7X_ABS16	16	S + A	
R_C7X_ABS32	17	S + A	
R_C7X_ABS64	18	S + A	
R_C7X_MVK32_LO5	19	S + A	Rela only
R_C7X_MVK32_HI27	20	S + A	Rela only
R_C7X_MVK_LO10	21	S + A	Rela only
R_C7X_MVK64_MID27	22	S + A	Rela only
R_C7X_MVK49_HI12	23	S + A	Rela only
R_C7X_MVK64_HI27	24	S + A	Rela only
R_C7X_PCR_OFFSET_LO5	25	S + A – P	Rela only
R_C7X_PCR_OFFSET_HI27	26	S + A – P	Rela only
R_C7X_PCR_BRANCH_LO19	27	S + A - P	
R_C7X_PCR_BRANCH_LO24	28	S + A - P	
R_C7X_PCR_EBRANCH_LO19	29	S + A - P	Rela only
R_C7X_PCR_EBRANCH_HI27	30	S + A - P	Rela only
R_C7X_PREL30	31	S + A - PC	
R_C7X_PCR_OFFSET_ADDKPC_LO5	32	S + A - P	Rela only
R_C7X_PCR_OFFSET_ADDKPC_HI27	33	S + A - P	Rela only

The `R_NONE` relocation performs no operation. It is used to create a reference from one section to another, to insure that if the referring section is linked in, so is the referee.

The `R_C7X_ABS*` relocations directly encode the relocated address of a symbol into 32- or 64-bit fields. They are commonly used for initialized data, not for instructions. The signedness of the field is unspecified; that is, they can be used for both signed and unsigned values.

```
.field    X,64      ; R_C7X_ABS64
.field    X,32      ; R_C7X_ABS32
```

The `R_C7X_MVK_*` relocations encode an absolute constant split into three fields for use in MVK instructions. The `LO5` and `LO10` relocations apply to the instruction itself. The `MID` and `HI` relocations apply to the constant extension field(s). `MVK49` is the same instruction as `MVK64`, but the `MVK49_HI12` relocation constrains the result to fit in 49 bits, ensuring its validity as an address.

```
MVK64    sym,reg   ; R_C7X_MVK_LO10
                   ; + R_C7X_MVK64_MID27
                   ; + R_C7X_MVK64_HI27

MVK49    sym,reg   ; R_C7X_MVK_LO10
                   ; + R_C7X_MVK64_MID27
                   ; + R_C7X_MVK49_HI12
```

The `R_C7X_PCR_BRANCH` and `R_C7X_PCR_EBRANCH` relocations encode a code address as a PC-relative offset for branches and calls. The examples here show CALL instructions; branch instructions are encoded in the same way. For the extended branch, the 46-bit offset is split into two fields. The `LO19` relocation applies to the instruction itself. The `HI27` relocation applies to the constant extension field.

```
CALL   func        ; R_C7X_PCR_BRANCH_LO19

CALL   func        ; R_C7X_PCR_BRANCH_LO24

CALLE  func        ; R_C7X_PCR_EBRANCH_LO19
                   ; + R_C7X_PCR_EBRANCH_HI27
```

The R_C7X_PCR_OFFSET relocations encode an address as a PC-relative offset for load and store instructions, and for the ADDKPC instruction. The offset is an unscaled byte offset. The offset is split into two fields. The LO5 relocation applies to the instruction itself. The HI27 relocation applies to the constant extension.

```
LDB  *PC($PCR_OFFSET(sym)),reg  ; R_C7X_PCR_OFFSET_LO5
                                ; + R_C7X_PCR_OFFSET_HI27

ADDKPC $PCR_OFFSET(label),reg   ; R_C7X_PCR_OFFSET_ADDKPC_LO5
                                ; + R_C7X_PCR_OFFSET_ADDKPC_HI27
```

R_C7X_PREL30 is used to encode addresses in exception handling tables. See Section 9.2. Note that although R_C7X_PREL30 is position-relative, it relocates data, not code. As such there is no fetch-packet adjustment.

11.5.2 Relocation Operations

The following table provides detailed information on how each relocation is encoded and performed. It uses the following additional notations:

- **F** — The relocatable field. The field is specified using the tuple [CS, O, FS], where CS is the container size, O is the starting offset from the LSB of the container to the LSB of the field, and FS is the size of the field. All values are in bits.
- **R** — The arithmetic result of the relocation operation
- **EV** — The encoded value to be stored back into the relocation field
- **SE(x)** — Sign-extended value of x

For relocations for which overflow checking is enabled, an overflow occurs if the encoded value, (including its sign, if any) cannot be encoded into the relocatable field. That is:

- A signed relocation overflows if the encoded value falls outside the half-open interval [-2^{FS-1}... 2^{FS-1}].
- An unsigned relocation overflows if the encoded value falls outside the half-open interval [0 … 2^{FS}].
- A relocation whose signedness is indicated as ‘either’ overflows if the encoded value falls outside the half-open interval [-2^{FS-1}… 2^{FS}].

Table 11-7. C7000 Relocation Operations

Relocation Name	Signed-ness	Field (F) [CS,O,FS]	Addend (A)	Result (R)	Overflow Check	Encoded Value (EV)
R_C7X_NONE	none	[0, 0, 0]	none	none	no	none
R_C7X_PCR16	signed	[16, 0, 16]	SE(F)	S + A - P	no	R
R_C7X_ABS16	either	[16, 0, 16]	SE(F)	S + A	no	R
R_C7X_ABS32	none	[32, 0, 32]	F	S + A	no	R
R_C7X_ABS64	none	[64, 0, 64]	F	S + A	no	R
R_C7X_MVK32_LO5	signed	[32, 0, 0]	SE(F)	S + A	yes	R
R_C7X_MVK32_HI27	signed	[32, 0, 32]	SE(F)	S + A	yes	R
R_C7X_MVK_LO10	signed	[32, 0, 0]	SE(F)	S + A	yes	R
R_C7X_MVK64_MID27	signed	[32, 0, 0]	SE(F)	S + A	yes	R
R_C7X_MVK49_HI12	signed	[32, 0, 49]	SE(F)	S + A	yes	R
R_C7X_MVK64_HI27	signed	[32, 0, 64]	SE(F)	S + A	yes	R
R_C7X_PCR_OFFSET_LO5	signed	[32, 0, 0]	SE(F)	S + A - P	yes	R
R_C7X_PCR_OFFSET_HI27	signed	[32, 0, 32]	SE(F)	S + A - P	yes	R
R_C7X_PCR_BRANCH_LO19	signed	[32, 8, 19]	SE(F)	S + A - P	yes	R >> 2
R_C7X_PCR_BRANCH_LO24	signed	[32, 8, 24]	SE(F)	S + A - P	yes	R >> 2
R_C7X_PCR_EBRANCH_LO19	signed	[32, 0, 0]	SE(F)	S + A - P	yes	R >> 2
R_C7X_PCR_EBRANCH_HI27	signed	[32, 0, 46]	SE(F)	S + A - P	yes	R >> 2
R_C7X_PREL30	signed	[32, 0, 30]	SE(F)	S + A - PC	yes	R >> 2

Table 11-7. C7000 Relocation Operations (continued)

Relocation Name	Signed-ness	Field (F) [CS,O,FS]	Addend (A)	Result (R)	Overflow Check	Encoded Value (EV)
R_C7X_PCR_OFFSET_ADDKPC_LO5	signed	[32, 0, 0]	SE(F)	S + A - P	yes	R
R_C7X_PCR_OFFSET_ADDKPC_HI27	signed	[32, 0, 32]	SE(F)	S + A - P	yes	R

11.5.3 Relocation of Unresolved Weak References

A relocation that refers to an undefined weak symbol is satisfied as follows:

- When used in a PC-Relative address offset relocation for the ADDKPC instruction (R_C7X_PCR_OFFSET_ADDKPC_*), the reference resolves to zero.
- When used in a PC-Relative address offset relocation for a branch or call instruction (R_C7X_PCR_[E]BRANCH_*), the instruction to be relocated is replaced with a NOP.

All other cases are non-conformant with the ABI.

Build Attributes

This chapter describes the build attributes and build attribute tags specified for the C7000 ABI.

Topic **Page**

12.1 Overview

The ABI specification for the ARM ABIv2 defines a mechanism called *Build Attributes* to capture build-time options used to create relocatable files so that a linker can enforce compatibility. The C7000 ABI uses the same structure to encode the build attributes in the ELF file as documented in ARM ABIv2 build attributes specifications in ARM Addenda to, and Errata in, the ABI for the ARM Architecture (document number ARM IHI0045A), which was released on 13 November 2007.

Build attributes are classified as vendor-specific or ABI-specific. This section documents build attributes that are ABI-specific. Vendors are free to implement additional toolchain-specific attributes.

Every ABI-conforming relocatable file must contain the build attributes section of type `SHT_C7X_ATTRIBUTES` (0x70000003), conventionally named `.c7xabi.attributes`. An executable file can optionally contain the build attributes section. A conforming tool should only use the section type to recognize the build attribute section.

The build attributes section consists of a one-byte version specifier with the value

'A'	vendor subsection	vendor subsection	. . .

Each subsection has the following format.

length	vendor name	0	vendor data
uint32	char[ ]	uint8	

The length field specifies the length in bytes of the entire subsection. The vendor name `c7xabi` is reserved for ABI-specified attributes, described in the following section. The format and interpretation of vendor data in other subsections is vendor-specific.

12.2 C7000 ABI Build Attribute Subsection

Attributes that are specified by this ABI are recorded in the subsection with the vendor string `c7xabi`. Toolchains should determine compatibility between relocatable files using solely these attributes; vendor-specific information should not be used other than as permitted by the `Tag_Compatibility` attribute which is provided for this purpose.

The vendor data in the `c7xabi` subsection contains any number of attribute vectors. Attribute vectors begin with a scope tag that specifies whether they apply to the entire file or only to listed sections or symbols. An attribute vector has one of the following three formats:

1	length	(omitted)		attributes	Apply to file
2	length	section numbers	0	attributes	Apply to specified sections
3	length	symbol numbers	0	attributes	Apply to specified symbols
ULEB128	uint32	ULEB128[ ]	ULEB128	see below	

The length field specifies the length in bytes of the entire attribute vector, including the other fields. The symbol and section number fields are sequences of section or symbol indexes, terminated with 0.

Attributes in an attribute vector are represented as a sequence of tag-value pairs. Tags are represented as ULEB128 constants. Values are either ULEB128 constants or NULL-terminated strings.

The effect of omitting a tag in the file scope is identical to including it with a value of 0 or "" (empty string), depending on the parameter type.

To allow a consumer to skip unrecognized tags, the parameter type is standardized as ULEB128 for even-numbered tags and a NULL-terminated string for odd-numbered tags. Tags 1, 2, 3 (the scope tags) and 32 (`Tag_ABI_Compatibility`) are exceptions to this convention.

As the ABI evolves, new attributes may be added. To enable older toolchains to robustly process files that may contain attributes they don't comprehend, the ABI adopts the following conventions:

- Tags 0-63 must be comprehended by a consuming tool. A consuming tool may choose to generate an error if an unknown tag in this range is encountered.
- Tags 64-127 convey information a consumer can ignore safely.
- For N >= 128, tag N has the same property as tag N modulo 128.

12.3 C7000 ABI Build Attribute Tags

Tag_ISA (=4), ULEB128

Tag_ISA specifies the C7000 ISA(s) that can execute the instructions encoded in the file. The following values are defined:

- **0** — No ISA specified
- **1** — C71x
- **other** — Reserved

Tag_ABI_PIC, (=6), ULEB128

- **0** — Addressing conventions unsuitable for a shared object
- **1** — Addressing conventions suitable for a shared object

`Tag_ABI_PIC` indicates that the object follows the addressing conventions required for a shared object, in particular that all references to imported variables are addressed via the GOT.

When linking a shared library, the linker should enforce the presence of this tag on all the objects that comprise the library.

The name `Tag_ABI_PIC` may be misleading. The term position independence may imply several related properties, which may or may not equate to the properties required for a shared object. Hence this attribute is defined in terms of the latter set.

Table 12-1 summarizes the build attribute tags defined by the ABI.

Table 12-1. C7000 ABI Build Attribute Tags

Tag	Tag Value	Parameter Type	Compatibility Rules
Tag_File	1	uint32_t	
Tag_Section	2	uint32_t	
Tag_Symbol	3	uint32_t	
Tag_ISA	4	ULEB128	See description above
Tag_ABI_PIC	6	ULEB128	Warn if absent when building shared library; combine using min value.
Tag_ABI_compatibility	32	ULEB128 char[]	See description in text.
Tag_ABI_conformance	67	char[]	Unspecified

Copy Tables and Variable Initialization

This chapter describes the copy table format, data compression formats, and initialization models used with the C7000 ABI.

Topic **Page**

13.1 Overview

Copy tables is the term for a general capability in the TI toolchain to facilitate moving data from offline storage to online storage. Offline storage generally refers to where the program is loaded; it could be ROM, slower memory, and so on. Online storage generally refers to where the data resides when the program runs. The data being copied can be either code or variables. The term copy table refers to a table of source and destination addresses in which objects to be copied are registered. There is also a runtime component in the form of library functions that read the table and perform the copying in response to calls in the program.

There are numerous applications for copy tables, but the two most common are:

- **Initialization** — In a ROM-based bare-metal system, initialized read-write variables must be copied from ROM to RAM at program startup time.
- **Overlays** — As the program runs, different code and data components are swapped in and out of a region of memory.

The copy table mechanism is not part of the ABI. The means by which initialized variables get their initial values is by contract between the linker and the runtime library, which are required to be from the same toolchain. However, there may be advantages for other toolchains to follow the TI mechanism, or there may be a need for downstream tools to recognize the format, so we document it here.

This chapter is organized as follows: first there is a general description of the mechanism, followed by a specification of the data structures involved. Finally there is a description of how the implementation of variable initialization in the TI toolchain builds upon the basic copy table functionality.

Figure 13-1 is an illustration of the general mechanism. An object file contains an initialized section, `.mydata` in the example. At link time, the user specifies that `.mydata` is to have separate load and run addresses, and specifies that a copy table entry be created for it. The linker vacates the data from `.mydata`, making it an uninitialized section, and assigns its address as its run location. It creates a new initialized section called `.mydata.load` [4] which contains `.mydata`'s data in encoded form, and places it at the load location. It links in a function called `copy_in` from the runtime library to decode and copy the data at runtime, as well as additional format-specific helper functions. Finally, it creates a section (`.ovly1` in the example) that contains a *copy table*, which is a sequence of *copy records* that point to the source data and the destination address, and a *handler table* (not shown) that the copy function uses to choose the right decode helper function.

At runtime, the application invokes `copy_in` to decompress and copy the data. The argument to `copy_in` is the address of the copy table associated with the section. The function parses the table and executes the specified copy operations.

Multiple objects can be encoded and registered for copy-in. Each generates its own copy table in the `.ovly1` section.

[4] Section names for copy table sections and compressed source data are arbitrarily chosen by the linker.

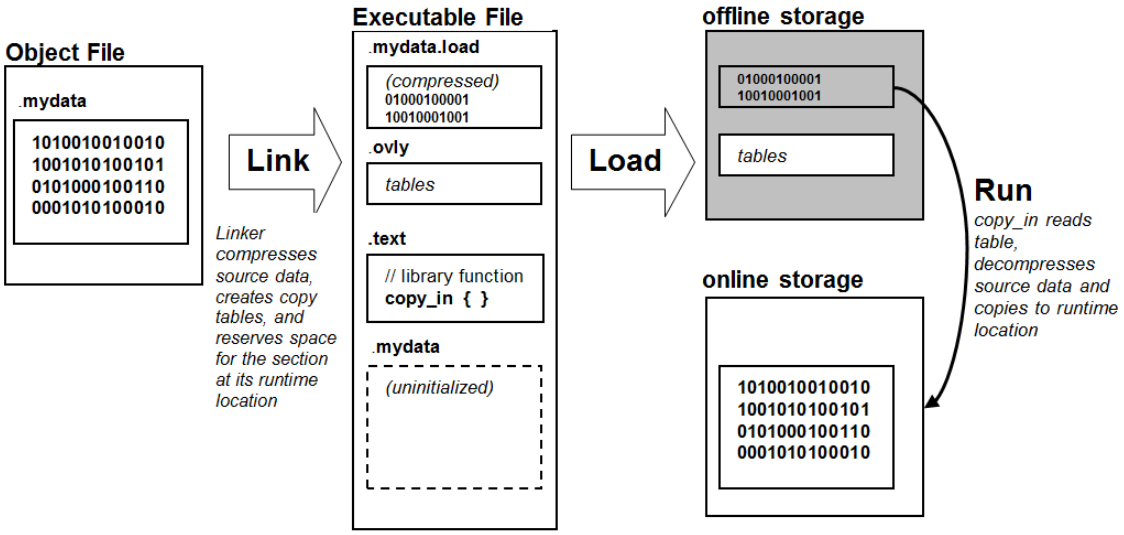


Figure 13-1. Copy Table Overview

A few variations are possible:

- **Multiple objects.** Multiple sections can be registered into a single copy table. This is so that all the code and data associated with an overlay can be copied in with a single invocation, without the application having to be aware of the number of separate components that comprise the overlay. A copy table can contain multiple copy records. Each copy record controls the copy-in of a contiguous chunk of code or data.
- **No compression.** The compression is optional. If compression is not enabled, there is no need for a separate load version of the section. The linker simply assigns separate load and run addresses to the initialized section.
- **Initialization.** Initialization of variables is a special case of the general mechanism. Copy records for initialization have a slightly different format, are stored in a different section called `.cinit`, and support zero-initialization as well as copy-in. These details are covered in Section 13.4.
- **Boot-Time Copy-In.** A special section called `.binit` contains copy tables that are automatically invoked at application startup time. This is similar to the initialization case, but whereas initialization is part of the language implementation and is therefore built-in to the toolchain, boot-time copy-in is strictly an application level operation.

13.2 Copy Table Format

A copy table has the following format:

```
typedef struct
{
    uint16_t     rec_size;
    uint16_t     num_recs;
    COPY_RECORD  recs[num_recs];
} COPY_TABLE;
```

`rec_size` is a 16-bit unsigned integer that specifies the size in bytes of each copy record in the table.

`num_recs` is a 16-bit unsigned integer that specifies the number of copy records in the table.

The remainder of the table consists of a vector of copy records, each of which has the following format:

```
typedef struct
{
    void*       load_addr;
    void*       run_addr;
    uint32_t    size;
} COPY_RECORD;
```

The `load_addr` field is the address of the source data in offline storage.

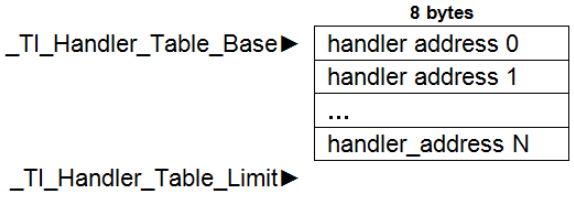
The `run_addr` field is the destination address to which the data will be copied.

The `size` field is overloaded. If `size` is non-zero, the source data is the exact image of the data to copy; in other words, it's not compressed. The copy-in operation is to simply copy size bytes from the load address to the run address.

If `size` is zero, the load data is compressed. The source data has a format-specific encoding that implies its size. In this case the first byte of the source data encodes the compression format. The format is encoded as an index into the *handler table*, which is a table of pointers to handler routines for each format in use.

The rest of the source data is format-specific. The copy-in routine reads the first byte of the source data to determine its format/index, uses that value to index into the handler table, and invokes the handler to finish decompressing and copying the data.

The handler table has the following format:



The copy-in routine references the table via special linker-defined symbols as shown. The assignment of handler indexes is not fixed; the linker reassigns indices for each application depending on what decompression routines are needed for that application. The handler table is generated into the `.cinit` section of the executable file.

The run-time support library in the TI toolchain contains handler functions for all the supported compression formats. The first argument to the handler function is the address pointing to the byte after the 8-bit index. The second argument is the destination address.

The following example provides a reference implementation of the copy_in function:

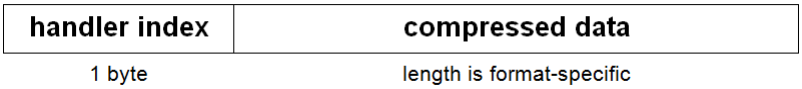
```
typedef void (*handler_fptr)(const unsigned char *src, unsigned char *dst);
extern int __TI_Handler_Table_Base;

void copy_in(COPY_TABLE *tp)
{
       unsigned short I;
       for (I = 0; I < tp->num_recs; I++)
       {
              COPY_RECORD crp = tp->recs[i];
              const unsigned char *ld_addr = (const unsigned char *)crp.load_addr;
              unsigned char       *rn_addr = (unsigned char *)crp.run_addr;

              if (crp.size)       // not compressed, just copy the data.
                  memcpy(rn_addr, ld_addr, crp.size);
              else                // invoke decompression routine
              {
                     unsigned char index = *ld_addr++;
                     handler_fptr  hndl  = ((handler_fptr *)(&__TI_Handler_Table_Base))[index];
                                           (*hndl)(ld_addr, rn_addr);
              }
        }
}
```

13.3 Compressed Data Formats

Abstractly, compressed source data has the following format:



The handler index specifies the decode function, which interprets the rest of the data. There are currently two supported compression formats for copy tables: Run-length encoding (RLE) and Lempel-Ziv Storer and Szymanski compression (LZSS).

RLE Format

The data following the 8-bit index is compressed using run length encoded (RLE) format. The C7000 uses a simple run length encoding that can be decompressed using the following algorithm:

1. Read the first byte and assign it as the delimiter (D).
2. Read the next byte (B).
3. If B != D, copy B to the output buffer and go to step 2.
4. Read the next byte (L).
5. If L > 0 and L < 4 copy D to the output buffer L times. Then go to step 2.
6. If L = 4 read the next byte (B'). Copy B' to the output buffer L times. Then go to step 2.
7. Read the next 16 bits (LL).
8. Read the next byte (C).
9. If C != 0 copy C to the output buffer L times. Go to step 2.
10. End of processing.

The RLE handler function in the TI toolchain is called `__TI_decompress_rle`.

LZSS Format

The data following the 8-bit index is compressed using LZSS compression. The LZSS handler function in the TI toolchain is called `__TI_decompress_lzss`. Refer to the implementation of this function for details on the format.

13.4 Variable Initialization

As described in Section 4.1, initialized read-write variables are collected into dedicated section(s) of the object file, for example .data. The section contains an image of its initial state upon program startup.

The TI toolchain supports two models for loading such sections. In the so-called *RAM model*, some unspecified external agent such as a loader is responsible for getting the data from the executable file to its location in read-write memory. This is the typical direct-initialization model used in OS-based systems or, in some instances, boot-loaded systems.

The other model, called the *ROM model*, is intended for bare-metal embedded systems that must be capable of cold starts without support of an OS or other loader. Any data needed to initialize the program must reside in persistent offline storage (ROM), and get copied into its RAM location upon startup. The TI toolchain implements this by leveraging the copy table capability described above. The initialization mechanism is conceptually similar to copy tables, but differs slightly in the details.

Figure 13-2 depicts the conceptual operation of variable initialization under the ROM model. In this model, the linker vacates the data from sections that contain initialized variables. The sections become uninitialized sections, allocated into RAM at their runtime address (much like, say, `.bss`). The linker encodes the initialization data into a special section called `.cinit` (for C Initialization), where the startup code from the runtime library decodes and copies it to its run address.

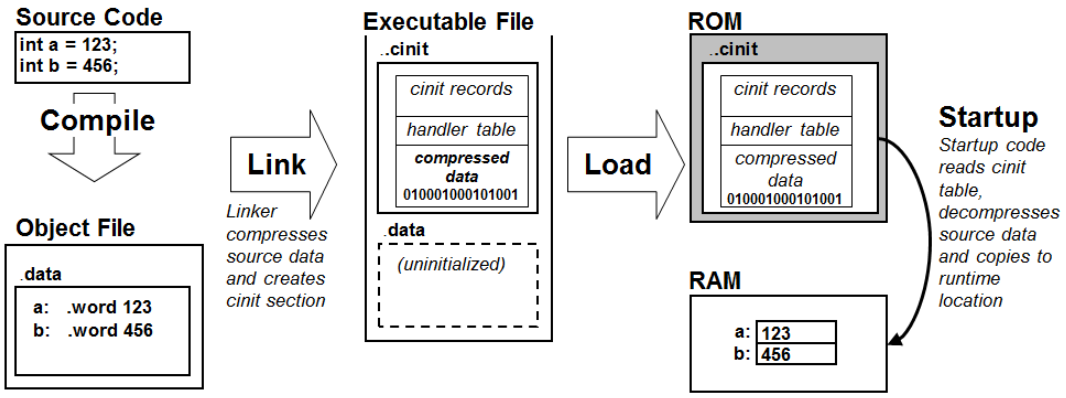


Figure 13-2. ROM-Based Variable Initialization via cinit

Like copy tables, the source data in the `.cinit` tables may or may not be compressed. If it is compressed, the encoding and decoding scheme is identical to that of copy tables so that the handler tables and decompression handlers can be shared.

The `.cinit` section contains some or all of the following items:

- The *cinit* table consists of *cinit records*, which are similar to copy records.
- The *handler table*, which consists of pointers to decompression routines, as described in Section 13.2. The handler table and handlers are shared by initialization and copy tables.
- The *source data*, which consists of compressed or uncompressed data used to initialize variables.

These items may be in any order. Figure 13-3 is a schematic depiction of the `.cinit` section.

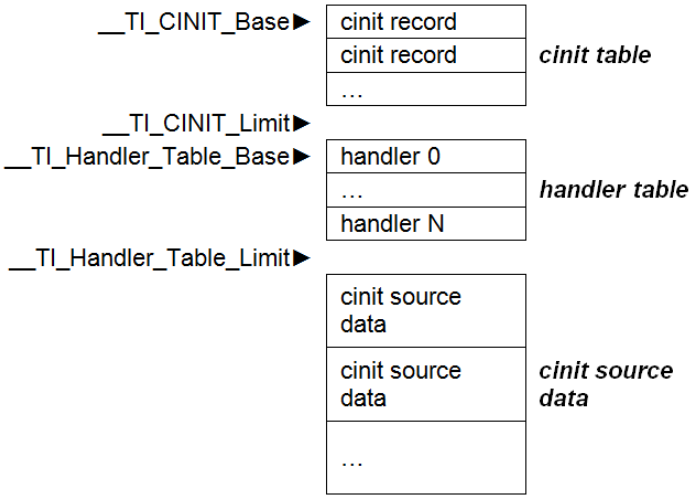


Figure 13-3. The .cinit Section

The `.cinit` section has the section type `SHT_TI_INITINFO` which identifies it as being in this format. Tools should rely on the section type and not on the name `.cinit`.

Two special symbols are defined to delimit the cinit table: `__TI_CINIT_Base` points to the cinit table, and `__TI_CINIT_Limit` points one byte past the end of the table. The startup code references the table using these symbols.

Records in the cinit table have the following format:

```
typedef struct
{
    void*      source_data;
    void*      dest;
} CINIT_RECORD;
```

The `source_data` field points to the source data in the cinit section. The `dest` field points to the destination address. Unlike copy table records, cinit records do not contain a size field; the size is always encoded in the source data.

The source data has the same format as compressed copy table source data described above, and the handlers have the same interface.

In addition to the RLE and LZSS formats, there are two additional formats defined for cinit records: uncompressed, and zero-initialized.

The explicit *uncompressed* format is required because unlike a copy table record, there is no overloaded size field in a cinit record. The size field is always encoded into the source data, even when no compression is used. The encoding is as follows:

handler index	padding	size	data
1 byte	3 bytes	4 bytes	'size' bytes

The encoded data consists of a `size` field, which is aligned on the next 4-byte boundary following the handler index. The `size` field specifies how many bytes are in the data payload, which begins immediately following the `size` field. The initialization operation copies `size` bytes from the data field to the destination address. The TI runtime library contains a handler called `__TI_decompress_none` for the uncompressed format.

The *zero-initialization* format is a compact format used for the common case of variables whose initial value is zero. The encoding is as follows:

handler index	padding	size
1 byte	3 bytes	4 bytes

The `size` field is aligned on the next 4-byte boundary following the handler index. The initialization operation fills `size` consecutive bytes at the destination address with zero. The TI runtime library contains a handler called `__TI_zero_init` for this format.

As an optimization, the linker is free to coalesce initializations of adjacent objects into single cinit records if they can be profitably encoded using the same format. This is typically significant for zero-initialized objects.

Extended Program Header Attributes

This chapter describes encoding and attributes used in the program header table.

Topic **Page**

14.1 Overview

ELF executable objects and shared libraries contain a program header table. Each entry in the program table describes a single segment. Along with the other metadata, the program table allows limited processor-specific extension of the segment attributes: 8 OS-specific flags and 4 processor-specific flags.

These flags can be used by a processor-specific ABI to represent additional segment properties. However, there are very few available flags, and they cannot be used to express attributes with parameters.

TI anticipates a need to specify additional system/device/application specific segment properties in the ELF program header table. The segment flags are not sufficient to represent all our segment attribute needs, so we have extended the ELF format to include *extended program header attributes*. A C7000 EABI conforming tool can choose to implement support for extended program header attributes as a quality-of-implementation issue. Support for extended program header attributes is not required to be C7000 EABI compliant.

Extended program header attributes are encoded in a processor specific section of type `SHT_TI_PHATTRS` (0x7F000004) and name `.TI.phattrs`. This section is contained in a segment specified by a segment of type `PT_C7X_PHATTR` (0x70000000).

14.2 Encoding

The program header attributes are encoded as <segment id, tag, value> triplets. Each attribute has the following representation:

```
typedef struct
{
       Elf_Half  pha_seg_id;        /* Segment id */
       Elf_Half  pha_tag_id;        /* Attribute kind id */
       Elf_Word  pha_value;            /* Tag value */
} Elf_TI_PHAttrs;
```

Both the segment id and the tag id are encoded as 2-byte unsigned integers in the byte order of the ELF file. The pha_value field is encoded as 4-byte unsigned integer in the byte order of the ELF file. This representation is modeled after the <tag, value> representation of dynamic tags.

The value of the tag can be an inlined 32-bit constant or an offset into the `.TI.phattrs` section that points to a fixed length binary data (FLBD) block or a null terminated byte string (NTBS). The fixed-length binary data size must be 4-byte aligned.

If the extended program header attributes segment is present, it is terminated by a `PHA_NULL` tag.

Attribute tag values and properties are assigned and maintained by TI and are processor-specific. All the undefined values are reserved for future use.

The attribute tag determines how the value of `pha_value` is interpreted. Each attribute has predefined behavior. The `pha_value` field can be interpreted as either the value itself or a pointer to the value as either NTBS or FLBD. If the interpretation is FLBD, the length of the field is pre-defined.

14.3 Attribute Tag Definitions

TI has introduced two attributes in support of native ROMing support.

Table 14-1. Extended Program Header Attributes

Name	Tag ID	pha_value	Length
PHA_NULL	0x0	ignored	none
PHA_BOUND	0x1	ignored	none
PHA_READONLY	0x2	ignored	none

The attribute PHA_BOUND indicates that the segment's address is bound to the final address and cannot change during downstream re-linking, dynamic linking, or dynamic loading steps. This property applies to segments that are either themselves located in ROM, or referred to using absolute addresses from code in ROM.

PHA_BOUND also indicates to the static or dynamic linker that this address is allocated and not available for further allocation.

PHA_READONLY indicates that the section contains "true" constant data; that is, the static and dynamic linkers are not allowed to perform any relocations on the contents or change the contents in any way. PHA_READONLY segments shall not have any relocation entries. The dynamic loader can use this as a hint to avoid relocation processing for such segments.

14.4 Extended Program Header Attributes Section Format

The extended program header attributes section contains three parts:

Program header attributes	Fixed-length binary data (FLBD)	Null-terminated byte strings (NTBS)

The first part is a vector of Elf_TI_PHAttrs, terminated by PHA_NULL. This is followed by the FLBD part and the NTBS part. If used, pha_value shall point into the FLBD or NTBS parts using byte offsets relative to the beginning of the section. FLBD and NTBS can be empty if there are no tags that point to additional data.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265