C6000 Embedded Application Binary Interface

Application Report

TEXAS INSTRUMENTS

Contents

Copyright © 2011–2014, Texas Instruments Incorporated

Copyright © 2011–2014, Texas Instruments Incorporated

List of Figures

List of Tables

C6000 Embedded Application Binary Interface

ABSTRACT

This document is a specification for the ELF-based Embedded Application Binary Interface (EABI) for the C6000 family of processors from Texas Instruments. The EABI defines the low-level interface between programs, program components, and the execution environment, including the operating system if one is present. Components of the EABI include calling conventions, data layout and addressing conventions, object file formats, and dynamic linking mechanisms. The purpose of the specification is to enable tool providers, software providers, and users of the C6000 to build tools and programs that can interoperate with each other.

1   Introduction

This document specifies the ELF-based Application Binary Interface (ABI) for the C6000 family of processors from Texas Instruments. The ABI is a broad standard that specifies the low-level interface between tools, programs, and program components.

1.1   ABIs for the C6000

Prior to release 7.0 of TI's C6000 Compiler Tools in 2009, the one and only ABI for C6000 was the original COFF-based ABI. It was strictly a bare-metal ABI; there was no execution-level component, and although various systems implement aspects of dynamic linking, there was no standardization or tools support for such mechanisms.

Release 7.0 of the TI Compiler Tools introduced a new ABI called the C6000 EABI. It 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 processor-specific aspects of the ABI, such as data layout and calling conventions, are largely unchanged from the COFF ABI, although there are some differences. Needless to say, the COFF ABI and the EABI are incompatible; that is to say, all of the code in a given system must follow the same ABI. TI's compiler tools support both the new EABI and the older COFF ABI, although we encourage migration to the new ABI as support for the COFF ABI may be discontinued in the future.

A *platform* is the software environment upon which a program runs. The ABI has platform-specific aspects, particularly in the area of conventions related to the execution environment, such as the number and use of program segments, addressing conventions, visibility conventions, pre-emption, dynamic linking, program loading, and initialization. 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, an implementation may provide 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 does not, 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 C6000.

1.2 Scope

Parts of the ABI Specification 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 chapter of this ABI specification.

The components in the bottom area relate to object-level interoperability.
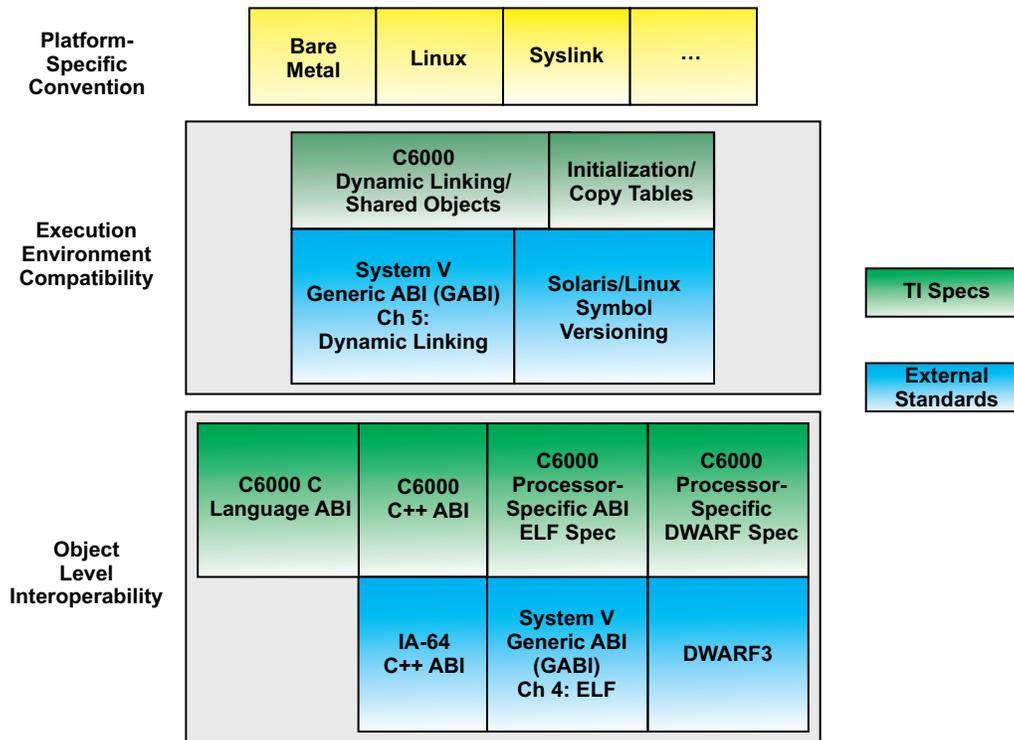
Parts of the ABI Specification

The **C Language ABI** (Section 2, Section 3, Section 4, Section 5, Section 8 and Section 9) specifies function calling conventions, data type representations, addressing conventions, and the interface to the C run-time library.

The **C++ ABI** (Section 10) specifies how the C++ language is implemented; this includes details about virtual function tables, name mangling, how constructors are called, and the exception handling mechanism (Section 11). The C6000 C++ ABI is based on the commonly prevalent IA-64 (Itanium) C++ ABI.

The **DWARF** component (Section 12) specifies the representation of object-level debug information. The base standard is the DWARF3 standard. This specification details processor-specific extensions.

The **ELF** component (Section 13) specifies the representation of object files. This specification extends the System V ABI specification with processor specific information.

Build Attributes (Section 17) 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.

The components in the central area of the diagram relate to execution-time interoperability. The **dynamic linking** components (Section 6 and Section 14.3) specify a mechanism whereby separately linked modules can interoperate, including the sharing of their code. Part of the dynamic linking mechanism is a method for data addressing such that separately linked modules can address each other's data without relocation.

Parts of the ABI Specification (continued)

Thread-Local Storage (Section 7) allows for the creation of thread-specific variables with static storage duration. The specification, representation, and access of thread-local variables is described in this document.

Symbol versioning (Section 16) is a mechanism whereby symbolic references include a minimum version, such that they are dynamically resolved with definitions having at least that version, in order to prevent run-time incompatibilities. This ABI adopts the standard GCC/Linux model, with no changes.

The components in the top part of Parts of the ABI Specification augment the ABI with platform-specific conventions that define the requirements for executables to be compatible with an execution environment, such as the number and use of program segments, addressing conventions, visibility conventions, pre-emption, program loading, and initialization. **Bare-Metal** refers to the absence of any specific environment. The only other environment currently covered by the ABI is the Linux platform (Section 15).

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 (Section 18) refers to the mechanism whereby initialized variables obtain their initial value. 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 (e.g. 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 (Section 19) 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. Some of the variants are incompatible with each other. For example, if any object uses the DBST PIC model, then all must. In such cases, toolchains are expected to use build attributes to prevent incompatible objects from being combined.

This section describes some of the more 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 Parts of the ABI Specification. 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, procedure linkage table (PLT) stubs, or a global offset table (GOT).

- **Bare Metal—Dynamic Linking**. This model refers to a system in which an executable may dynamically link to separately linked modules, but not within the controlled environment of an OS. Addressing may or may not be position-independent, depending on the environment. The environment may impose additional conventions on addressing or placement. This model would use the dynamic linking components of Parts of the ABI Specification. Specifics of the bare-metal dynamic linking model are detailed in Section 14.4.

- **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 ABI solves this through two related structures: position-independent addressing, and the data segment base table (DSBT) mechanism.

- **Position Independence**. This refers to a means of addressing without the use of address constants, enabling code and/or data to be loaded and run at any address without relocation. The term *PIC* generally means Position Independent Code, but position independence can refer to code, data, or both. Shared libraries require position independent data so that multiple clients can have private copies; in the context of shared libraries, the term PIC sometimes connotes this narrower definition. Libraries in ROM may require position independent addressing to reference other objects if their addresses are not bound when the ROM is created. Position-independent data relies on the Data Page register (B14). When multiple modules are involved, such as with dynamic linking, the *DSBT* (Data Segment Base Table) model is a mechanism that can be used to reset the DP when calling from one module to another.

- **Linux**. Executables and Shared Libraries built for the Linux environment must follow certain conventions. They have dynamic linking information. They require position independence using the DSBT model. Objects built for Linux have the ELFOSABI_C6000_LINUX flag in the EI_OSABI field of the ELF header. Augmentations to the ABI for the Linux platform are detailed in Section 15.

- **ROMing**. It may be desirable to build a separately linked module that will reside in ROM. Once linked, its addresses are permanently bound. It may be subsequently linked against other modules, either statically or dynamically. For this purpose, the ABI defines a special class of ELF file that presents both a static and dynamic linking view and a handful of section flags to indicate sections whose addresses are permanently bound. ROM modules typically use PIC addressing to make them independent of the placement of other modules they reference.

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 C6000 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 run-time libraries that implement part of the language support provided by the toolchain.

The library format used by the C6000 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 the following 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.

* A **relocatable module**. A relocatable module is a shared object file that also contains static linking information: that is, a static symbol table, section table, and static relocation entries. It is intended for ROMable libraries that can be either statically or dynamically linked.

This specification uses the terms *static link unit* and *load module* interchangeably to refer to executables and shared libraries (including relocatable modules).

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 C6000 Architecture Overview

The TMS320C6000, familiarly C6000 or C6x, is a family of 32-bit VLIW Digital Signal Processors from Texas Instruments. The family includes both fixed-point (integer) and floating-point devices. The architecture is capable of issuing up to 8 32-bit instructions per cycle for a high level of parallelism. Table 1 lists the members of the C6000 family covered by this ABI.

Table 1. C6000 ISAs

ISA	Data Format	Description
C62x	Fixed-point	Original ISA
C64x	Fixed-point	C62x with additional instructions and registers
C64x+	Fixed-point	Additional instructions and compact instruction encoding
C67x	Floating-point	Original floating-point ISA
C67x+	Floating-point	C67x with additional instructions and registers
C6740	Fixed/float	Union of C64x+ and C67x+ plus additional instructions
C6600	Fixed/float	C6740 with 128-bit data path plus additional instructions

Most family members are backwards compatible; that is, newer CPUs can correctly execute object code built for older devices. Specific cases are specified under the Tag_ISA build attribute in Section 17.2.

C6000 devices are byte-addressable. Memory can be configured as big-endian or little-endian. Most devices have no general memory-management unit so CPU addresses refer to actual physical memory locations (no virtual memory).

The pipeline of the C6000 is unprotected. That is, when the CPU reads the destination of a previously-issued computation which is still in the pipeline and has not yet been written, the read will obtain the old value rather than stalling to wait for the new one. The implication is that the programmer (or compiler) must manage pipeline latencies and schedule operations so as to obtain the correct result. Operations with multi-cycle latencies include loads (4 cycles), branches (5 cycles), and certain multiplies (2 cycles).

The C6000 has a minimum of 32 general-purpose registers, designated A0-A15 and B0-B15. Members of the C64 family extend this to 64 registers: A0-A31 and B0-B31. Two of these registers are assigned by convention for use with addressing and linkage under the ABI. B15 is designated as the Stack Pointer, usually denoted as **SP**; and B14 is designated as the Data Page Pointer, denoted as **DP**. The DP is used as a base address for the data segment, providing a means for both position independence and efficient access to (near) data.

1.9 Reference Documents

Document Title	Link or URL
TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide	SPRU732
TMS320C6000 Optimizing Compiler User's Guide	SPRU187
TMS320C6000 Assembly Language Tools	SPRU186
ELF Specification—GABI Chapters 4/5	http://www.caldera.com/developers/gabi/2003-12-17/contents.html
IA64 (Itanium) C++ ABI	http://refspecs.linux-foundation.org/cxxabi-1.83.html
IA64 (Itanium) Exception Handling ABI	http://www.codesourcery.com/public/cxx-abi/abi-eh.html
Application Binary Interface for the ARM Architecture	http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.swdev.abi/index.html
C Library ABI for the ARM Architecture	http://infocenter.arm.com/help/topic/com.arm.doc.ihi0039b/IHI0039B_clibabi.pdf
DWARF DEBUGGING Format Version 3	http://dwarfstd.org/Dwarf3.pdf
C Language Standard	http://www.open-std.org/jtc1/sc22/wg14, ISO/IEC 9899:1990
C99 Language Standard	http://www.open-std.org/jtc1/sc22/wg14, ISO/IEC 9899
C++ Language Standard	http://www.open-std.org/jtc1/sc22/wg21, ISO/IEC 14882:1998

1.10 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 often used:

sym	The symbol being referenced
label	A symbol referring to a code address
func	A symbol referring to a function
tmp	A temporary register (also tmp1, tmp2, etc)
reg, **reg1**, **reg2**	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.

2 Data Representation

This section describes the representation in memory and registers of the standard C data types. Other languages may be supported but presumably their objects would correspond to these types.

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

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 gives the size and alignment of C data types in bits.

Table 2. Data Sizes for Standard Types

Type	Generic Name	Size	Alignment
signed char	char	8	8
unsigned char	uchar	8	8
char	plain char	8	8
bool (C99)	uchar	8	8
_Bool (C99)	uchar	8	8
bool (C++)	uchar	8	8
short, signed short	int16	16	16
unsigned short	uint16	16	16
int, signed int	int32	32	32
unsigned int	uint32	32	32
long (32 bits), signed long	int32	32	32
unsigned long	uint32	32	32
long (40 bits)	int40	40	64
long long, signed long long	int64	64	64
unsigned long long	uint64	64	64
enum	--	varies (see Section 2.8)	32
float	float32	32	32
double	float64	64	64
long double	float64	64	64
pointer	--	32	32

The generic names in the table are used in this specification to identify these types in a language-independent way.

The alignment values listed are the default alignment values. They apply in all cases except when a type is used inside a "packed" structure. All structure member types inside of a "packed" structure have an alignment of 8 bits.

The *char* type is signed.

The integral types have complementary unsigned variants. The generic names are prefixed with 'u' (e.g. uint32).

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

In the former COFF ABI for C6000, the C type *long* was a 40-bit integer, corresponding to the longest native integer type of the original 62x hardware. By default, the EABI changes long to 32 bits to be compatible with universal unwritten convention. However, toolchains may wish to support compatibility with legacy codes developed under the COFF ABI by supporting an optional 40-bit type, designated as long or otherwise, so the representation is described here. Notice that this type has a size of 40 bits but an alignment of 64 bits. If the 40-bit *long* type is used for a member of a "packed" structure, the type has an 8-bit alignment but a 64-bit container size.

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

```
typedef unsigned int    size_t;
typedef int             ptrdiff_t;
typedef unsigned int    wchar_t;
typedef unsigned int    wint_t;
typedef char *          va_list;
```

2.2   Data in Registers

In general, implementations are free to use registers as they see fit. The standard register representations specified in this section apply only to values passed to or returned from functions.

Objects whose size is 32 bits or less can reside in single registers. Numeric values in registers are always right justified; that is, bit 0 of the register contains the least significant bit of the value. Signed integral values smaller than 32 bits are sign extended into the upper bits of the register. Unsigned values smaller than 32 bits are zero extended.

Objects whose size is between 32 and 64 bits use register pairs. Register pairs consist of an even-numbered register, which holds the least significant part of the value, and the next consecutive odd-numbered register, which holds the most significant part. Register pairs are denoted as R_o:R_e where R_o is the odd register and R_e is the even one (for example, A5:A4). Numeric values in register pairs are right justified into the even register; that is, bit 0 of the even register contains the least significant bit and bit 0 of the odd register contains bit 32 of the value. Signed integral values are sign extended into the upper bits of the odd register. Unsigned values are zero extended.

Objects larger than 64 bits have no designated register representation.

2.3   Data in Memory

The C6000 can be configured to run in either big or little endian mode. Endianness refers to the memory layout of multi-byte values. 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.

Scalar variables are aligned such that they can be loaded and stored using the native instructions appropriate for their type: LDB/STB for bytes, LDH/STH for halfwords, LDW/STW for words, and so on. These instructions correctly account for endianness when moving to and from memory.

Forty-bit integers have 5 bytes, designated 0 (LSB) through 4 (MSB). In memory, 40-bit values are padded to 64 bits (8 bytes). If the address of the value in memory is N, then Representation of 40-Bit Values in Memory gives the storage layout:

location	little-endian	big-endian
N	byte 0 (lsb)	padding
N+1	byte 1	
N+2	byte 2	
N+3	byte 3	byte 4 (msb)
N+4	byte 4 (msb)	byte 3
N+5	padding	byte 2
N+6		byte 1
N+7		byte 0 (lsb)

Representation of 40-Bit Values in Memory

2.4 Complex Types

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. This leaves the ABI little flexibility. Accordingly, the C6000 representation for complex types is as in Table 3:

Table 3. Complex Types

Type	Generic Name	Size	Alignment	External Alignment
float _Complex	complex32	64	32	64
double _Complex	complex64	128	64	128
long double _Complex	complex64	128	64	128

Variables with type complex or array of complex with external visibility have stricter alignment requirements than that required by their type. The external alignment column of the table gives the minimum alignment for such variables.

2.5 Structures and Unions

Structure members are assigned offsets starting at 0. Each member is assigned the lowest available offset that satisfies its alignment. Padding may be required between members to satisfy this alignment constraint.

Union members are all assigned an offset of 0.

The underlying representation of a C++ class is a structure. Elsewhere in this document the term *structure* applies to classes as well.

The alignment requirement of a structure or union is equal to the most strict alignment requirement among its members, including bit field containers as described in the next section. The size of a structure or union in memory is rounded up to a multiple of its alignment by inserting padding after the last member. Structures and unions passed by value on the stack have special alignment rules as specified in Section 3.3.

Structures having size 64 bits or less may reside in registers or register pairs while being passed to or returned from functions.

In little-endian mode a structure in a register is always right justified; that is, the first byte occupies the LSB of the register (the even register if a pair) and subsequent bytes of the structure are filled into the increasingly significant bytes of the register(s).

In big-endian mode the following rules govern the layout of structures in registers:

- A 1-byte structure occupies the LSB of a single register.
- For a 2-byte structure, the first byte occupies byte 1 of the register and the second byte occupies byte

0 (the LSB)

- For a 3- or 4-byte structure the first byte occupies byte 3 (the MSB) of the register and the remaining bytes fill the register towards the LSB. A three byte structure has one byte of padding in the LSB of the register.

- For a 5- to 8-byte structure the first byte occupies byte 3 (the MSB) of the upper (odd) register and the remaining bytes fill the decreasingly significant bytes. The 5- to 7-byte structures have padding in the LSBs of the lower (even) register.

Big-Endian Layout for Structures or Unions in Registers depicts the big-endian register representation of structures with sizes of one through eight bytes.

structure size	high (odd) register				low (even) register			
1 byte structure								byte 0
2 byte structure							byte 0	byte 1
3 byte structure					byte 0	byte 1	byte 2	
4 byte structure					byte 0	byte 1	byte 2	byte 3
5 byte structure	byte 0	byte 1	byte 2	byte 3	byte 4			
6 byte structure	byte 0	byte 1	byte 2	byte 3	byte 4	byte 5		
7 byte structure	byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	
8 byte structure	byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7

Big-Endian Layout for Structures or Unions in Registers

The rationale for this layout is to allow the structure to be copied between registers and memory using the smallest appropriate memory reference; for example, a 2-byte struct uses a 16-bit reference, a 3-byte struct uses a 32-bit reference, a 5-byte struct uses a 64-bit reference, and so on. The structure is left-justified within the size of the containing memory reference.

2.6 Arrays

The minimum alignment for an object with the *array* type is that specified by the type of its elements.

File-scope array variables with external visibility have stricter requirements. Depending on the target ISA, the alignment of such a variable is the maximum of its element alignment and for:

C62x, C67x	4 bytes
All others	8 bytes

2.7 Bit Fields

The C6000 EABI adopts its bit field layout from the IA64 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 refer to it in this document. The container type is the major determinant of how bit fields are packed and aligned.

The C89, C99, and C++ language standards have different requirements for the declared type:

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 structure 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 IA64 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 C6000 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 described previously.
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 mode, containers are filled from LSB to MSB. In big-endian mode, containers are filled from MSB to LSB.

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

A declared type of plain *int* is treated as a *signed int* by C6000 EABI.

2.7.1 Volatile Bit Fields

A volatile bit field is one declared with the C *volatile* keyword. When a volatile bit field is read, its container must be read exactly once using the appropriate access for the entire container.

When a volatile bit field with a size less than its container 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.

When a volatile bit-field with a size exactly equal to the container size is written, it is unspecified whether the read takes place. Because such reads are unspecified, it is not safe to interlink object files compiled with different implementations if they both write to a volatile bit-field with exactly the width of its container. For this reason, using volatile bit-fields in external interfaces should be avoided.

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.

2.8 Enumeration Types

Enumeration types (C type *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.)

The C standard requires enumeration constants to fit in type "signed int", so enum types may only be int or unsigned int in strict ANSI mode. Wider enum types are possible in C++. The TI compiler also allows wider enum types in relaxed and GCC modes.

3   Calling Conventions

3.1   *Call and Return*

The C6000 has different instructions that can be used to effect a function call depending on the ISA variant and the context of the call. In any case, a function call is executed by saving the return address in register B3 and branching to the called function. The called function returns by executing an indirect branch to the address that was in B3 when it was called.

3.1.1   Return Address Computation

On 64x targets, a call is a two-instruction sequence: an ADDKPC instruction calculates the return address into B3 using PC-relative addressing, followed by a B (branch) instruction.

```
        ADDKPC  return_label,B3    ; B3 := return_label
        B       func               ; goto func
  return_label:
```

On non-64x targets, the address can be computed using other methods such as absolute, PC-relative, or GOT-based addressing (as described in Section 5.1).

3.1.2   Call Instructions

The call itself is generated as a simple PC-relative branch that transfers control to the callee:

```
        B       func               ; goto func
```

The displacement is a 21-bit signed word offset. If the destination is unreachable, the linker generates a trampoline, which is a stub function that uses absolute, PC-relative, or GOT-indirect addressing to address the destination function. For more information about trampolines see Section 5.3.

For an indirect call, the destination is a register:

```
        B       reg                ; goto address in reg
```

For branches that implement calls, the TI toolchain uses the **CALL** pseudo-instruction, which encodes as a branch but annotates the debug information so that profilers, debuggers, or other analysis tools can identify the instruction as a function call (see Section 5.3). So the previous direct call would actually appear in the assembly source as:

```
        CALL  func                 ; encodes as B func
```

The C64+ ISA has a composite instruction CALLP that single-handedly implements a call. CALLP integrates these steps:

*   Load the address of the next execute packet into B3 as the return address
*   NOP 5 to fill the delay slots of the branch

A call using CALLP is simply:

```
        CALLP func,    B3
```

3.1.3   Return Instruction

A function return is executed by branching to the address passed in B3. The callee is free to move the address and store it elsewhere, typically required for nested calls. Assuming the address is still in B3 the instruction would be:

```
        B       B3                 ; return
```

If the function is an interrupt handler function, this must be a branch to IRP instead:

```
        B       IRP                ; return from interrupt handler
```

The TI toolchain uses the **RET** pseudo instruction to designate branches that implement function returns.

3.1.4   Pipeline Conventions

On the C6000, there are five delay slots between the fetch of a branch instruction—including a call or return—and the cycle in which it executes. Instructions may be scheduled in the delay slots subject to the following: a caller is responsible for ensuring that the effects of all instructions that could affect the callee are complete before the E1 phase of the callee's first instruction. Similarly, for a return instruction, the callee is responsible for ensuring that all instructions that could affect the caller are complete before the E1 phase of the instruction at the return address.

3.1.5   Weak Functions

A weak function is a function whose symbol has binding STB_WEAK. A program can successfully link without a definition of a weak function, leaving references to it unresolved.

The linker handles unresolved weak function calls differently depending on whether static linking or dynamic linking is used.

When static linking is used, the linker replaces calls to undefined weak functions with what effectively becomes a NOP. But to enable optimizations such as tail call elimination in which the callee does not return to the call site, the replacement must preserve some aspects of the call's behavior. Therefore the replacement is not with NOP, but with a surrogate return instruction:

```
        B.S2   B3                    ; replacement for unresolved weak call
```

This behavior imposes these additional requirements on calls to weak functions:

- The S2 functional unit must be available. This is trivially ensured if the original call is encoded on S2.
- The return address must be available in B3 when the call instruction reaches the E1 phase of the pipeline. In other words, the compiler cannot schedule the return address computation in the delay slots of the call.

  The ABI supports calls to imported weak functions; that is, potentially defined in a different static link unit.

When dynamic linking is used, the global offset table (GOT) is used to decide whether to leave the unresolved weak function with a value of 0 or patch in the address of _ _c6xabi_weak_return(), which simply returns to its caller. If the value of 0 is used (for the Linux platform), a guard is required. If the _ _c6xabi_weak_return() function is used (for the bare metal platform), no guard is not required.

3.2   Register Conventions

The C6000 has at least 32 general-purpose 32-bit registers. Registers may contain integers, floating-point values, or pointers. The general purpose registers are divided into two register files, designated as A and B.

B15 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 (see Section 4.3).

B14 is designated as the Data Page Pointer (DP). It points to the beginning of the data segment for the currently active object.

The ABI does not designate a dedicated Frame Pointer (FP) register. However, the TI compiler uses A15 as a frame pointer in some circumstances.

GCC supports lexically nested functions as a language extension. The implementation uses a register, called the static chain register, to provide the parent function's activation context to the child. The choice of register is largely toolchain-specific, unless the call is interceded in some way (by a trampoline, for example). For this reason, the ABI designates A2 as the recommended choice for the static chain register. The calling conventions support this designation by including A2 as one the registers involved in function linkage, requiring its value to be preserved between a call site and the entry point of the callee.

The ABI designates A10-A15 and B10-B15 as *callee-saved* registers. That is, a called function is expected to preserve them so they have the same value on return from a function as they had at the point of the call. Note that this set includes the SP (B15) and DP (B14).

In addition, the ILC and RILC are callee-saved registers. These are control registers used by the C64+'s SPLOOP mechanism.

All other registers are *caller-save* registers. 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.

The Address Mode Register (AMR) is a user-writable control register that enables circular addressing. At function call boundaries bits 0-15 of the AMR must be 0 so that circular addressing is disabled.

Table 4 lists the registers and their role in the ABI.

Table 4. C6000 Register Conventions

Register	Alias	Preserved by Callee	Role in Calling Convention
A0		no	
A1		no	
A2		no	Static chain register for nested functions
A3		no	Address for returned-by-reference structure
A4		no	First argument; return value (LSW)
A5		no	First argument; return value (MSW)
A6		no	Third argument (LSW)
A7		no	Third argument (MSW)
A8		no	Fifth argument (LSW)
A9		no	Fifth argument (MSW)
A10		yes	Seventh argument (LSW)
A11		yes	Seventh argument (MSW)
A12		yes	Ninth argument (LSW)
A13		yes	Ninth argument (MSW)
A14		yes	
A15	FP	yes	Frame pointer
A16-A31		no	
B0		no	Dynamic reloc offset argument to lazy binder; see Section 15.6
B1		no	Dynamic reloc offset argument to lazy binder; see Section 15.6
B2		no	
B3		no	Return address
B4		no	Second argument (LSW)
B5		no	Second argument (MSW)
B6		no	Fourth argument (LSW)
B7		no	Fourth argument (MSW)
B8		no	Sixth argument (LSW)
B9		no	Sixth argument (MSW)
B10		yes	Eighth argument (LSW)
B11		yes	Eighth argument (MSW)
B12		yes	Tenth argument (LSW)
B13		yes	Tenth argument (MSW)
B14	DP	yes	Data page pointer
B15	SP	yes	Stack pointer
B16-B29		no	
B30-B31		no!	Trampoline scratch registers; see Section 3.7

3.3 Argument Passing

The first ten arguments to a function are passed in registers. Arguments are assigned, in declared order (except arguments passed in register quads), to registers in the following sequence:

A4, B4, A6, B6, A8, B8, A10, B10, A12, B12

Arguments whose size is between 32 and 64 bits are passed in register pairs, using the even registers from the previous list for their least-significant part and the corresponding odd register for their most-significant part. For example, in the following example 'a' is passed in A4 and 'b' in B5:B4:

```
func1(int a, double b);
```

Arguments of type float complex are passed in register pairs. The ordering is endian dependent. In little-endian mode the real part is passed in the even register and the imaginary part in the odd. For big-endian mode this ordering is reversed.

Arguments of type double complex are passed in register quads, using the first available quad from the following list: A7:A6:A5:A4, B7:B6:B5:B4, A11:A10:A9:A8, B11:B10:B9:B8. In little-endian mode the real part is passed in the lower-numbered pair (e.g. A5:A4) and the imaginary part is passed in the higher-numbered pair (A7:A6). For big-endian this ordering is reversed. Any register that is bypassed for a quad-register argument is available for subsequent arguments. For example, in the following function 'w' is passed in A4, 'x' is passed in B4, 'y' is passed in A11:A10:A9:A8, and 'z' backfills into A6:

```
func2(int w, int x, double complex y, int z);
```

Any remaining arguments are placed on the stack at increasing addresses, starting at SP+4. Each argument is placed at the next available address correctly aligned for its type. Thus if the first stack argument requires 64-bit alignment, its address will be SP+8.

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

Structures and unions with size 64 bits or less are passed by value, either in registers or on the stack as described in the list that follows. Structures and unions larger than 64 bits are passed by reference, as described in Section 3.5.

Any arguments not passed in registers are placed on the stack at increasing addresses, starting at SP+4. Each argument is placed at the next available address correctly aligned for its type, subject to the following additional considerations:

- The stack alignment of a scalar is that of its declared type.
- Regardless of the alignment required by its members, the stack alignment of a structure passed by value is the smallest power of two greater than or equal to its size. (This cannot exceed 8 bytes, which is the largest allowable size for a structure passed by value.)
- Each argument reserves an amount of stack space equal to its size rounded up to the next multiple of its stack alignment.

Note that SP+4 is not 8-byte aligned, so if the first argument requires 8 byte alignment, it will be stored in memory at SP+8.

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.

3.4 Return Values

Scalars and structures whose size is 32 bits or less are returned in A4. Scalars and structures between 32 and 64 bits are returned in A5:A4.

Objects of type float complex are returned in A5:A4, with the real part in the odd register and the imaginary part in the even register.

Objects of type double complex are returned with the real part in A5:A4 and the imaginary part in A7:A6.

Aggregates larger than 64 bits are returned by reference.

3.5 Structures or Unions Passed and Returned by Reference

Structures (including classes) and unions larger than 64 bits are passed and returned by reference. To pass a structure or union 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 the called function returns a structure or union larger than 64 bits, the caller must pass an additional argument in A3 containing a destination address for the returned value, or NULL if the returned value is not used.

The callee returns the object by copying it to the address in A3, 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 A3.

3.6 Conventions for Compiler Helper Functions

The ABI specifies so-called *helper functions* that the compiler uses to implement language features. Generally, these functions adhere to the standard calling convention, but an exception is made for a handful of them to enable better performance. See Section 8.3 for helper functions that use modified conventions.

3.7 Scratch Registers for Inter-Section Calls

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 function code by omitting the save and restore around the call. This arises when the definition has been seen, or when calling helper functions with special conventions as described in Section 8.3.

However, the registers B30 and B31 are designated as potentially modified by any call that crosses a section boundary, even if the definition has been seen or when calling helper functions. This is so that if the call requires a far-call trampoline (Section 5.3), B30 and B31 are available as scratch registers in the trampoline.

Additionally, the lazy binding mechanism for Linux requires caller-save registers to be available for the stub functions that implement lazy binding. The compiler must not optimize a call site when the callee may be imported and therefore potentially called via lazy binding.

Calls within the same section never require trampolines; for such intra-section calls B30 and B31 are treated no differently than other caller-save registers.

3.8 Setting Up DP

An exported function compiled under the DSBT model for shared objects may need to set up the DP upon entry and restore it upon entry. This is discussed in Section 6.7.1.

4   Data Allocation and Addressing

4.1   *Data Sections and Segments*

In a relocatable object file that is 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 1. The ABI designates static data sections as *near* or *far*. Near sections can be addressed using efficient near DP-relative addressing, but their size and placement is constrained. Conventions for placement of static variables into sections and for how they are addressed are covered in Section 4.2.2.

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 sections accessed with near DP-relative addressing must be grouped such that they fall within the unsigned 15-bit addressing range of the static base address as defined by _ _c6xabi_DSBT_BASE.
- All data within a given segment is subject to the same segment attributes (see Section 19)
- Within a segment, initialized data must precede uninitialized data. This is a structural constraint of ELF.
- Any additional restrictions imposed by the platform-specific conventions.

A segment is designated as DP-relative if it is accessed using DP-relative addressing. A single DP-relative segment may contain a mixture of near and far addressing, provided it meets the constraints listed previously.

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

Figure 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 1. Data Sections and Segments (Typical)

The .const and .fardata:.const sections contain read-only constants. The .const section contains position-independent constants. Depending on the platform, the .const section may be located in read-only memory, and may be addressed using absolute addressing, or in position-independent models, relative to code via PC-relative addressing. On some platforms such as Linux that share read-only segments, const objects whose initializers contain address constants cannot be shared. Accordingly, they are placed into a distinct section called .fardata:.const, so named because it can be considered part of .fardata, in the data segment.

The .rodata section contains read-only constants addressable with near DP-relative addressing.

The .neardata and .fardata sections contain initialized read-write variables. These sections correspond to the .data section commonly found on other architectures.

The .bss and .far sections contain uninitialized variables.

The .common and .scommon sections contain common-block symbols allocated by the linker. These are not actual sections in the object files. Instead, the section names are a convention in the linker command file for placing variables. These sections should not be used for other purposes.

The .got and .dsbt sections contain data structures related to dynamic linking. See Section 6.

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.

A data segment designated as a *DP-relative segment* is addressed using DP-relative addressing. Upon entry to any code in the load module, the DP is initialized to point to a process-private copy of the load module's DP-relative segment with the lowest address. The linker defines the symbol _ _c6xabi_DSBT_BASE to point to this address.

DP-relative addressing has two forms. Near DP-relative addressing applies when the DP-relative offset can be encoded into a single instruction as a 15-bit unsigned constant. Far DP-relative addressing applies when it cannot, necessitating additional instructions. When a variable is addressed using the near form, its placement is constrained to be within 32 KB of the DP.

DP-relative segments are identified by the PF_C6000_DPREL flag in the program header (see Section 14.1).

Some platforms (Linux, in particular) may constrain load modules to have no more than one DP-relative segment.

Additional data segments containing static variables are referred to as *absolute data segments*, and are addressed using either absolute or GOT-based addressing. There are no restrictions on their number, size, or placement.

If a program is dynamically linked and has shared libraries, the data segments of each load module are independent from those of other load modules. In particular, each load module has its own data segments, including DP-relative segment(s), and therefore its own DP. If multiple executables share a library, they each get a private copy of that library's data segment(s). The model for managing multiple data segments in the absence of virtual address translation is called the DSBT model; it is described in Section 6.7.

4.2.1 Addressing Methods for Static Data

The ABI supports the following fundamental schemes for addressing static data: DP-relative, Absolute, GOT-indirect, and PC-relative. Which one is used in a given situation depends on a variety of factors, including the variable's declaration, the execution platform, whether the module is being built as an executable or shared library, visibility conventions, and so on. Since the compiler generates the addressing it must be aware of this context, usually via command-line options and/or visibility directives in the source code. Other sections of this ABI provide details on *when* each form of addressing applies; this section specifies *how* the addressing is performed.

4.2.1.1 Near DP-Relative Addressing

This is the default addressing for static variables in the near DP segment. The DP offset is a 15-bit unsigned value, limiting this form of addressing to objects within 32 KB of the DP.

```
LDW    *+DP(sym),dest            ;reloc R_C6000_SBR_U15_W (also _B and _H)
```

4.2.1.2 Far DP-Relative Addressing

This is a position-independent way of addressing *far* data; that is, data in segments other than the near DP segment. The 32-bit DP-relative offset is loaded into a register using 2 MVK instructions, which is then added to the DP using indexed addressing. The offset must be appropriately scaled for the size of the access. The TI toolchain uses special assembly language operators to indicate the scale factor.

```
MVKL   $DPR_word(sym),tmp        ;reloc R_C6000_SBR_L16_W
MVKH   $DPR_word(sym),tmp        ;reloc R_C6000_SBR_H16_W
LDW    *+DP(tmp),dest

MVKL   $DPR_hword(sym),tmp       ;reloc R_C6000_SBR_L16_H
MVKH   $DPR_hword(sym),tmp       ;reloc R_C6000_SBR_H16_H
LDH    *+DP(tmp),dest

MVKL   $DPR_byte(sym),tmp        ;reloc R_C6000_SBR_L16_B
MVKH   $DPR_byte(sym),tmp        ;reloc R_C6000_SBR_H16_B
LDB    *+DP(tmp),dest
```

4.2.1.3 Absolute Addressing

The following instructions use absolute addressing.

```
MVKL    sym,tmp                 ;reloc R_C6000_ABS_L16
MVKH    sym,tmp                 ;reloc R_C6000_ABS_H16
LDW     *tmp,dest
```

Because this addressing mode encodes an address, it is position-dependent.It can be used to access far data. Compared to the Far DP-relative scheme described previously, an actual access has the same cost, but computing the address of a variable (&sym) does not require adding the DP, saving one instruction. There is no scaling in this case because the loaded constant is the actual address rather than an offset.

4.2.1.4 GOT-Indirect Addressing

The Global Offset Table (GOT) is a position-independent mechanism used to dynamically resolve addresses that cannot be known at static link time. Addresses in the GOT are resolved by a dynamic loader. GOT-based addressing is discussed in Section 6.6.

4.2.1.5 PC-Relative Addressing

This is a position-independent way of addressing *far* data in the code segment. The data is assumed to be located at a (link-time) constant offset from the code that accesses it. Examples include label tables for switch statements, and read-only constant variables that can be placed into the code segment (.const). The addressing mechanism is the same whether addressing code or data; it is described in Section 5.1.

4.2.2 Placement Conventions for Static Data

Interoperability between toolchains requires that addressing generated by one is consistent with placement generated by another, especially with respect to near DP-relative addressing. Any variable addressed with near DP-relative addressing must be allocated in a section that is placed within 32 KB of the DP.

This requires the ABI to establish some conventions. Some of these conventions depend on toolchain-specific behavior, such as code generation models supported, or even user behavior, such as command line options selected or language extensions applied. For this reason, the ABI takes a two-pronged approach:

- To achieve consistency, the ABI defines some abstract conventions for placement and addressing, that map to toolchain behavior in some toolchain-specific way. These conventions make it possible to build compatible object files with different toolchains, but cannot precisely specify how to do so.
- To enforce consistency, the ABI requires the linker to either link the program in such a way that the addressing constraints are satisfied, or refuse to link the program.

The toolchain generating the addressing may only have visibility to a variable's declaration and not its definition. Therefore, the conventions must be based only on information available at both points. This excludes, for example, the use of array dimensions.

4.2.2.1 Abstract Conventions for Placement

The abstract conventions designate variables as either near or far, as follows:

- Any variable declared with a toolchain-specific keyword, attribute, or pragma that designates it as near or far assumes that designation.
- Any variable declared with a toolchain-specific keyword, attribute, or pragma to be in a section other than .bss, .rodata, or .neardata is designated as far.
- Any remaining variable is designated according to one of three models, typically controlled by command-line options.
  - Near model—all variables not otherwise designated are designated as near
  - Far model—all variables not otherwise designated are designated as far
  - Far-aggregate model—variables with scalar type are designated as near; variables with aggregate type (that is: arrays, classes, structs, and unions) are designated as far. This should be the default

model for a toolchain.

Toolchains may support other models but must minimally support these three. Interoperability with other toolchains may or may not be achievable if other models are used.

In the cases where the designation depends on toolchain-specific aspects like command-line options or language extensions, the onus is on the programmer to use these constructs consistently wherever the variable is declared, but on the linker to catch errors (see Section 4.2.2.3).

The ABI establishes conventional assignments of variables to sections. A variable's assignment is a function of its near/far designation and its initialization category, as determined by the first matching condition from the following list.

* A variable is uninitialized if it has no initializer, or is initialized via a constructor call at startup.
* A variable is const if its type is const-qualified.
* A variable is initialized if it has an initializer.

The conventional section assignment is given by Table 5:

Table 5. Conventional Assignments of Variables to Sections

Designation	Initialization category		
	Uninitialized	Initialized	Const
near	.bss	.neardata	.rodata
far	.far	.fardata	.const

The conventional assignments may be overridden in toolchain-specific ways. For example, variables may be assigned to user-defined sections. However, the toolchain must not allow users to place variables designated as far into any of the three near sections.

4.2.2.2   Abstract Conventions for Addressing

How a variable is addressed depends on its designation as near or far, its visibility, and the code generation model (for example, position-independent vs. position-dependent).

Only objects designated as near can be addressed with near DP-relative addressing. Near objects may also be addressed in other ways, for example with absolute addressing (position dependent) or through the GOT (position-independent), but none of these ways are inconsistent with near placement.

Variables designated as far cannot be addressed using near DP-relative addressing.

4.2.2.3   Linker Requirements

The linker is responsible for ensuring that variables addressed using near DP-relative addressing are placed such that they within the required 15-bit range of the DP, as established the _ _c6xabi_DSBT_BASE symbol. The linker can detect such accesses as being marked by R_C6000_SBR_* relocation entries. If the linker cannot satisfy this constraint (perhaps due to conflicting instructions from the user), it must fail to link the program.

4.2.3   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 Section 18. Other toolchains may adopt a compatible mechanism, a different mechanism, or none at all.

4.3   Automatic Variables

Local variables of a procedure, i.e. 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 either via the stack pointer (B15), or in cases where the offset is too large, via a temporary frame pointer register (A15) that points to the activation frame and can support greater offsets.

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 2-word (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

There are at least two cases that require a standardized layout for the local frame and ordering of callee-saved registers. They are exception handling and debugging.

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

The stack grows toward zero. The SP points to the word above the topmost allocated word; that is, the word at *(SP+4) is allocated, but *SP is not.

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

A compiler is free to allocate one or more "frame pointer" registers to access the frame. The TI compiler uses A15 as a frame pointer (FP). If FP is allocated, its value is the value of SP before the function's frame is created. In other words, FP points to the bottom of the current frame, and the top of the caller's. Objects in the frame are accessed via FP with negative offsets. Incoming arguments are accessed via FP with positive offsets.

Insofar as a frame pointer is not part of the linkage between functions, the choice of whether to use a frame pointer, which register to use, and where it points is up to the discretion of the toolchain. However, some of the virtual instructions used for stack unwinding assume that A15 points to the frame as described in the preceding paragraph. If a function has no frame pointer, or uses a different convention as to which register is used or where it points, then these unwinding instructions cannot be used and a less efficient sequence may be required.

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-saved area** stores registers modified by the function that must be preserved. If exceptions or debugging is 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 3.3. The size of the section is the maximum required for any single call.

(stack grows towards zero)
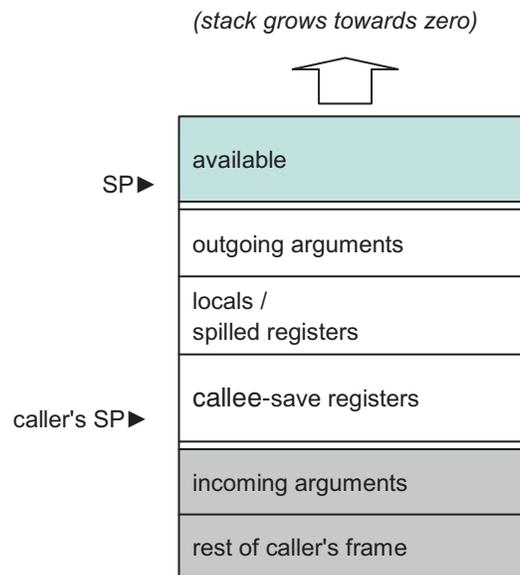
Figure 2. Local Frame Layout

4.4.1 Stack Alignment

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 atomic adjustment to SP must be a multiple of 8 bytes.

The double word (8 bytes) at the bottom of the frame spans a frame boundary. That is, the first word is in the callee's frame but the second is in the caller's frame, so neither can use it to store a double word. This is unfortunate from the point of view of saving and restoring registers using double word loads and stores, but is a historical carryover from prior architectures that did not have double word support. In the diagrams that follow, double word boundaries are indicated with heavier lines.

Before the first instruction in a function, the stack looks like this:
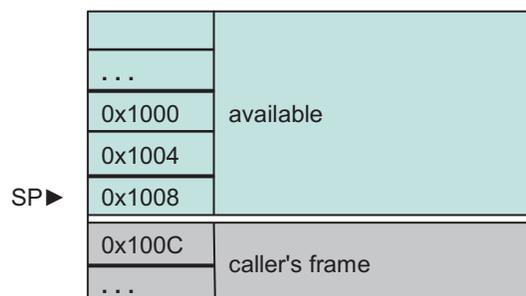
If this function needs one word on the stack to store something, it will need to allocate a frame of 2 words (because SP must always remain 8-byte aligned). The allocation is performed by decrementing SP by 8. Now the stack looks like this:
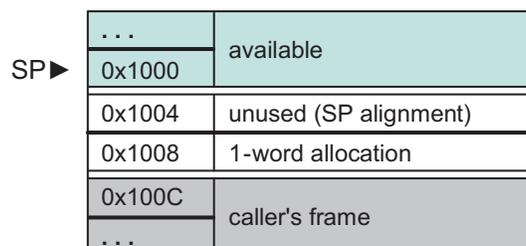
4.4.2   Register Save Order

As discussed in Section 3.2, functions are responsible for preserving the contents of registers designated as *callee-saved*, normally accomplished by saving modified registers in the local frame upon entry to the function and restoring them before exit. Usually, the order and locations of the callee-saved registers on the stack do not matter, as long as they are restored from the same location as they were saved. In most cases, the compiler saves registers in an arbitrary order. However, there are some features which require a known ordering:

- **Safe Debug**. The safe debug convention applies when symbolic debugging is enabled (often indicated by the -g option). In this mode, the compiler saves and restores the registers in a fixed order on the stack.

- **Exception Handling**. The stack unwinding process for exception handling needs to know exactly where each register is so that it can simulate the function epilog. To efficiently encode this information using a bit vector, we defined a fixed order. Exception handling re-uses the *callee-saved register safe debug order* for encoding the bit vectors, so the orderings are generally the same, with certain exceptions as follows.

The callee-saved register safe debug order is A15, B15, B14, B13, B12, B11, B10, B3, A14, A13, A12, A11, A10.

When using safe debug, and in the absence of a special stack layout (see Section 4.4.3 and Section 4.4.4), the compiler will always save registers in that relative order, starting at the bottom (highest address) of the frame. 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.

4.4.2.1   *Big-Endian Pair Swapping*

For targets that have double-word (64-bit) LDDW and STDW instructions, it is more efficient to save registers belonging to even-odd pairs arranged on the stack so that the pair can be read with one LDDW. Note that the safe-debug ordering for little-endian frequently places registers so that this is true; this is not entirely by coincidence. However, for big-endian, the order of each pair would need to be reversed. When compiling for big-endian, the compiler looks for register pairs that occupy the same aligned double word on the stack, and swaps the order. This is still considered safe debug ordering, despite the fact that the ordering is not the same as little endian, and the big-endian order can vary for functions which save different registers. This swap occurs even on C6x targets which do not support LDDW or STDW.

Keep in mind that the safe debug ordering is consulted first for placing the registers on the stack; the ordering for a given even-odd pair is swapped only if the offset is evenly divisible by 8. If the save offset is not aligned, the registers will be saved individually in the original order.

4.4.2.2   Examples

If all 13 callee-saved registers are saved by a function compiled for C62x, the save area looks like this. Bold entries in the big-endian column indicate swapped pairs.

SP▶	little-endian	big-endian
. . .	rest of callee's frame	
0x1004		
0x1008	A10	**A11**
0x100C	A11	**A10**
0x1010	A12	**A13**
0x1014	A13	**A12**
0x1018	A14	A14
0x101C	B3	B3
0x1020	B10	**B11**
0x1024	B11	**B10**
0x1028	B12	**B13**
0x102C	B13	**B12**
0x1030	B14	**B15**
0x1034	B15	**B14**
0x1038	A15	A15
0x103C	caller's frame	
. . .		

Figure 3. C62x Save Area When All Callee-Saved Registers Are Saved by Function

If only registers B13, B12, A12, A11, and A10 are saved:

SP▶	little-endian	big-endian
. . .	rest of callee's frame	
0x1004		
0x1008	A10	**A11**
0x100C	A11	**A10**
0x1010	A12	A12
0x1014	B12	B12
0x1018	B13	B13
0x103C	caller's frame	
. . .		

Figure 4. C62x Save Area When Only Registers B13, B12, A12, A11, and A10 Are Saved

Observe that B13:B12 is not swapped because its offset is not correctly aligned.

4.4.3 DATA_MEM_BANK

The pragma DATA_MEM_BANK creates a hole on the stack to guarantee a specific alignment for local variables. To achieve this, it stores the old SP on the stack and clears the low bits of the SP.

If a FP has been allocated, the SP is simply restored from the FP, and we do not need to save the old SP on the stack:

```
aligned
   SP►  | available              |
        | outgoing arguments     |
        | locals                 |
        | spilled registers      |
        | hole                   |
        | callee-save registers  |
   FP►  |                        |
        | caller's frame         |
```
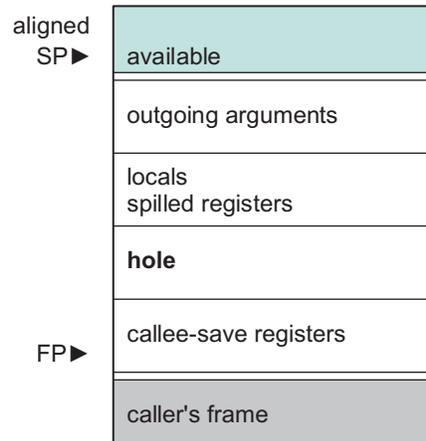
If a FP has not been allocated, then we have to save the old value of SP on the stack:

```
aligned
   SP►  | available              |
        | outgoing arguments     |
        | locals                 |
        | spilled registers      |
        | callee-save registers  |
        | old SP                 |
        | hole                   |
        | caller's frame         |
```
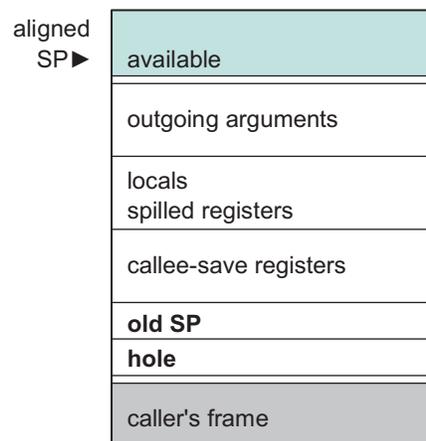
4.4.4 C64x+ Specific Stack Layouts

In order to aggressively reduce code size, there are special stack layouts used for some functions on C64x+ and C674x.

4.4.4.1 _ _c6xabi_push_rts Layout

Many functions save and restore all the callee-saved registers, and the code to do this is fairly large. Instead of having the code to do this in the prolog and epilog of every function, there are functions in the runtime library that can be called instead. These functions use a special calling convention to avoid corrupting the registers they will save. The call to save all of the callee-saved registers looks like this:

```
        CALLP __c6xabi_push_rts, A3     ; CALLP puts the return address in A3
```

The code to restore them is:

```
        CALLP __c6xabi_pop_rts, A3      ; A3 is unused
                        ; Returns to the location saved from B3 by __c6xabi_push_rts
```

Before _ _c6xabi_push_rts is called, the stack looks like this:

SP► 0x1080	available
0x1084	caller's frame
. . .	

The _ _c6xabi_push_rts stores all the callee-saved registers, resulting in:

SP►

	little-endian	big-endian
. . . 0x1048	available	
0x104C	unused (SP alignment)	
0x1050	X (pushed with B3)	B3
0x1054	B3	X (pushed with B3)
0x1058	A10	A11
0x105C	A11	A10
0x1060	B10	B11
0x1064	B11	B10
0x1068	A12	A13
0x106C	A13	A12
0x1070	B12	B13
0x1074	B13	B12
0x1078	A14	A15
0x107C	A15	A14
0x1080	B14	B14
0x1084 . . .	caller's frame	

4.4.4.2   Compact Frame Layout

In order to encourage the use of compressible instructions, the register save/restore code is slightly different for some C64x+ functions.

Ordinarily, the compiler allocates the entire frame with one SP decrement and then saves the callee-saved registers with SP-relative writes.

```
STW    B14,      *SP--[12]
STDW   A15:A14,  *SP[5]
STDW   A13:A12,  *SP[4]
STDW   A11:A10,  *SP[3]
STDW   B13:B12,  *SP[2]
STDW   B11:B10,  *SP[1]
```

In compact frame mode, the compiler instead generates a series of SP-auto-decrementing stores for each register pair.

```
STW    B14,      *SP--[2]
STDW   A15:A14,  *SP--
STDW   A13:A12,  *SP--
STDW   A11:A10,  *SP--
STDW   B13:B12,  *SP--
STDW   B11:B10,  *SP--
```

For this case, the stack layout is the same. However, the stack layout can be different for different saved register subsets. For instance, suppose we need to save A10, A11, B10, B11, and B3, and we choose to use the compact frame layout to save code size. The traditional layout will make the most efficient use of stack space:

```
STW   A11, *SP--[6]
STW   A10, *+SP[5]
STW   B3,  *+SP[4]
STW   B11, *+SP[3]
STW   B10, *+SP[2]
```

SP▶	0x1050	available
	0x1054	unused (SP alignment)
	0x1058	B10
	0x105C	B11
	0x1060	B3
	0x1064	A10
	0x1068	A11
	0x106C	caller's frame

However, the compact frame layout will leave multiple holes in the register save area in order to use more compressible SP decrement instructions. Since every push must decrement SP by 8 (for interrupt safety), the compiler tries to push members of register pairs together; if it cannot, it must push a single register into a double word, making a hole in the save area.

```
STW   A11, *SP--[2]
STW   A10, *SP--[2]
STDW  B11:B10, *SP--
STW   B3,  *SP--[2]
```

	. . . 0x1048	available
SP▶		
	0x104C	unused (SP alignment)
	0x1050	B3
	0x1054	unused
	0x1058	B11
	0x105C	B10
	0x1060	A10
	0x1064	unused
	0x1068	A11
	0x106C	caller's frame

4.5   Heap-Allocated Objects

Dynamically allocated objects, such as via C's malloc() or C++'s operator "new", 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.

5 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.

Except for the compact instruction encoding format of the C64x+, all instructions on the C6000 are 32 bits wide. Labels that represent function addresses, as well as most other labels, are always aligned on 32-bit boundaries. Considerations for compact instructions are discussed in Section 5.4.

There are three ways a code object can be referenced: computing its address, as a branch destination, or by calling it as a function.

5.1 Computing the Address of a Code Label

An assembly code section needs to compute a code address to:

- Perform a call or branch
- Create a function pointer
- Form the return address for a call
- Fill switch tables
- Use in a trampoline or PLT entry generated by the linker

There are three basic ways to form the address of a code object: Absolute, PC-relative, and GOT-based addressing. Absolute addressing is position dependent; the PC-relative and GOT forms are position independent.

5.1.1 Absolute Addressing for Code

The basic approach is to simply encode the destination as an absolute constant:

```
MVKL label, B5       ; B5 := lower 16 bits of label
MVKH label, B5       ; B5 := upper 16 bits
```

Any code that encodes such a constant directly becomes position dependent, having the undesirable property of requiring patching if relocated, for example at load time.

5.1.2 PC-Relative Addressing

This is a position independent way of addressing code (or constant data in the code segment). The address is computed as the sum of the address of the current fetch packet and a constant.

```
base:  MVC   PCE1,tmp1                      ; address of current fetch packet
       MVK   $PCR_OFFSET(label,base),tmp2   ; label-base, reloc R_C6000_PCR_L16
       MVKH  $PCR_OFFSET(label,base),tmp2   ; label-base, reloc R_C6000_PCR_H16
       ADD   tmp1,tmp2,tmp2                  ; &label
```

The $PCR_OFFSET assembly operator evaluates to the offset between the fetch packet containing the instruction labeled by base (MVC) and the target symbol.

5.1.3 PC-Relative Addressing Within the Same Section

When the referenced label is within the same section as the reference, the offset is an assembly-time constant. Assuming the offset can be encoded in 15 or fewer bits, it can be added directly to the base address using ADDK:

```
base:  MVC PCE1,tmp                      ; address of current fetch packet
       ADDK  label-(base & ~0x1F),tmp    ; no reloc; tmp == &label
       ...
label: ; must be in same section
```

Here no relocation is needed; the assembler encodes the offset directly. The expression "(base & ~0x1F)" represents the address of the fetch packet containing base.

(If the offset is too large to encode with ADDK, MVK/MVKH/ADD must be used as described in Section 5.1.2.)

5.1.4   Short-Offset PC-Relative Addressing (C64x)

C64x and newer architectures have an optimized instruction to implement PC-relative addressing for nearby labels:

```
        ADDKPC label, B5            ; B5 := label, position independent
```

This form constrains the label to be within a signed 7-bit constant offset (+/- 64 words) of the fetch packet containing the ADDKPC instruction.

ADDKPC is useful for computing return addresses, which are typically only a few instructions away.

5.1.5   GOT-Based Addressing for Code

Another position independent form for computing a code address is to load it from the global offset table as described in Section 6.5.

5.2   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.

The encoding uses a 21-bit signed offset scaled by 2, yielding a range of $\pm 2^{22}$ bytes (4MB). This effectively limits the size of any given function to 4MB.

5.3   Calls

The C6x does not have a specific call instruction. [1] A call is generated by generating the return address into a register (B3) and executing a branch. The return address calculation is covered in Section 3.1. A direct call to a named function generates a PC-relative branch, and therefore subject to the 4MB limit.

5.3.1   Direct PC-Relative Call

If the direct call's target function is placed at a location that is unreachable with the offset in a direct CALL instruction, the static linker rewrites the CALL instruction so that it instead calls a helper stub function called a *trampoline*. The trampoline simply calls the target function. The linker is responsible for placing the trampoline within the reach of the CALL instruction.

```
        CALL  sym                  ; reloc R_C6000_PCR_S21
```

Note: CALL is a pseudo-up; this instruction encodes as a B.

5.3.2   Far Call Trampoline

If the call target is defined in the same static link unit, but is unreachable with a 21-bit word offset, the static linker generates a trampoline, which is a stub function that uses an alternate form of addressing to reach the target. This example illustrates a trampoline using absolute addressing: [2]

```
    $Tramp$$sym2:
        MVKL sym,tmp               ;reloc R_C6000_ABS_L16
        MVKH sym,tmp               ;reloc R_C6000_ABS_H16
        B tmp
```

Alternatively, the trampoline may compute the address of the destination function using other forms of addressing as described in Section 5.1.

If the call target is not defined in the same static link unit, the static linker generates a PLT entry, which is similar to a trampoline. This case is covered in Section 6.5.

Depending on the addressing used, trampolines typically require either one or two scratch registers (e.g. tmp in the previous sequence).

[1]   The C64x+ has CALLP, which combines ADDKPC, B, and NOP 5.
[2]   The name of the trampoline label may vary by convention.

For C64x and newer targets, B30 and B31 are available for use by any trampoline. Callers are prevented from assuming that B30 and B31 are preserved by calls, even when the called function is known not to modify them (see Section 3.7).

For the older C62 and C67 targets, trampolines must save and restore any registers they use.

5.3.3 Indirect Calls

An indirect call through a function pointer generates a branch with a register operand. For example:

5.4 Addressing Compact Instructions

The C64+ ISA and some later variants have a feature known as compact instructions, an encoding format that packs pairs of 16-bit instructions into 32-bit words of program memory. In big-endian mode, the instructions are stored in memory in the opposite order from their lexical order in the source program, and opposite from the order in which they execute. This section clarifies the conventions for representing addresses in the face of this discrepancy.

A 16-bit instruction may be considered to have two addresses:

- Its **physical address** is the address in which it physically resides in program memory.
- Its **logical address** is the address at which it appears to reside from the point of view of the program's control flow. The logical address can be thought of as the program counter value corresponding to the instruction. The program executes instructions in logical address order, corresponding to the lexical order of the program. Branch targets and displacements are computed according to logical addresses.

In the little-endian configuration, logical addresses are the same as physical addresses. In big-endian, pairs of 16-bit instructions are swapped in program memory such that if address A represents the physical address of the 32-bit word containing the pair, the first logical instruction is stored A+2 and the second at A.

The code fragment in Figure 5 illustrates the distinction between logical and physical addresses. The program is shown in its lexical order. The first column shows the little-endian physical address, which is also the logical address for both little- and big-endian. The second column shows the big-endian physical address. The dashed lines represent 32-bit boundaries in program memory.

Physical Address		Opcode	Source Code	
Little-Endian	Big-Endian			
			code:	
0000	0002	40CE	MV.S1	A1,A2
0002	0000	41B0	|| ADD.L1	A2,A3,A3
0004	0006	58E7	NEG.L2	B1,B1
0006	0004	614F	|| MV.S2	B2,B3
0008	000A	814F	MV.S2	B2,B4
			local:	
000A	0008	A14F	MV.S2	B2,B5
000C	000E	C14F	MV.S2	B2,B6
000E	000A	40CE	MV.S1	A1,A2
0010	0012	41B0	|| ADD.L1	A2,A3,A3
0012	0010	58E7	|| NEG.L2	B1,B1

Figure 5. Addressing Compact Instructions

The ABI specifies that all program addresses in the object file are represented as logical addresses. This includes branch displacements, symbol values, addresses in unwinding tables, and addresses in debug information.

Referring to Figure 5, the value of the label code in the symbol table is 0x0000 even though the instruction it labels (opcode 0x40CE) is stored at 0x0002. Similarly, the value of the label local is 0x000A even though the instruction it labels (opcode 0xA14F) is stored at 0x0008.

For the most part the distinction between logical and physical addresses is transparent to both the programmer and the toolchain. To preserve this transparency, the following conditions are imposed:

1. Branches must be to labeled instructions. It is possible to construct a branch to an unlabeled instruction but the results are undefined.

2. With certain exceptions, labels must be aligned on 32-bit boundaries. The exceptions are:

   (a) A label that can be determined not to be a branch destination (e.g. a DWARF label)

   (b) A label that is the target of an intra-section branch from one fetch packet to another (the compact form of BNOP encodes a half-word offset).

3. A relocatable field cannot occur within a 16-bit instruction.

Conditions (1) and (2) taken together exclude indirect branches to addresses that are not 32-bit aligned. Note an instance of (2b) does not require a relocation since the offset is an assembly-time constant. This in turn enables condition (3), sidestepping the need to translate between logical and physical addresses to access a relocatable field.

6   Addressing Model for Dynamic Linking

In the most basic scenario for building a standalone program for a bare-metal environment, a program is statically linked and bound to run at a specific address. The linker simply patches all references with their final resolved address, and the program is ready to run. This scenario is simple and efficient.

Increasingly, even embedded systems consist of multiple components that are separately linked. This naturally leads to the dynamic linking model common on general-purpose systems: dynamic-link libraries (DLLs) on Windows or dynamic shared objects (DSO) on Unix-based platforms, including Linux. This section describes a set of conventions for a base-level dynamic linking and shared object mechanism for the C6000. Object file mechanisms relating to dynamic linking are covered in Section 14.3. Specific execution platforms, such as Linux, may specify additional conventions; see Section 15.

6.1   Terms and Concepts

Static linking is the traditional process of combining relocatable object files and static libraries into a *static link unit*: either an ELF *executable file* (.exe) or and ELF *shared object* (.so). Within this document, we use the term *load module* (or simply module) to refer to a static link unit and *shared library* (or library) to refer to a shared object..

A *program* consists of exactly one executable file and any additional shared libraries that it depends on to satisfy any undefined references. If multiple executables depend on the same library, they can share a single copy of its code (hence the shared in shared object), thereby significantly reducing the memory requirements of the system.

When such a program consisting of multiple objects is loaded, references between its component modules (some of which may already be loaded as part of other applications) must be resolved. This process is called *dynamic linking*, and is handled by a run-time component known as the *dynamic linker*. Because the state of the system varies with respect to which objects are loaded at any given time, the dynamic linker may wish to control their memory allocation and placement dynamically. The ability to assign a program's location and relocate it at load time is sometimes referred to as *dynamic loading*. Although dynamic linking and dynamic loading are somewhat independent capabilities, in that either may be useful without the other, the mechanisms that enable each are tightly related. In this document we use the term dynamic linking to refer to the composite capability, and the terms dynamic linker and dynamic loader interchangeably to refer to the component that performs these operations.

An object's *own* functions and variables (collectively called *symbols*) are those that are defined within it. When a module (executable or library) references a symbol that is undefined within that module but defined in another module, it is said to *import* that symbol. The defining module is said to *export* the symbol.

In general, the addresses of a dynamically linked module's code and data are not known at static link time. Furthermore, the addresses of any imported symbols are also unknown, until they are resolved by the dynamic linker. Therefore when the dynamic linker loads a module, it may need to patch its code and/or data according to its assigned address, as well as the addresses of any symbols it imports. Relocations performed at dynamic link time are called *dynamic relocations*. A design goal of most dynamic linking mechanisms is to minimize the number and complexity of dynamic relocations. Dynamic relocations, and the associated symbolic information, are contained in special sections in the ELF object file.

A fundamental issue with shared libraries is that each executable that shares a library must still have its own **private** (not shared) copy of the library's data. This implies that shared code cannot use absolute addressing to access data. The term *Position Independent Data (PID)* applies to code that accesses data in a shareable way, typically via either relative or GOT-based addressing.

The broader term *Position Independent Code (PIC)* refers to code that does not use absolute addressing in any way, and is therefore independent from both its own placement and that of other load modules. Position independent code requires no load-time patching to the code segment, thereby speeding load time and/or allowing it to be located in ROM. Typical approaches for PIC rely on PC-relative addressing, virtual memory, indirection, and/or relative addressing from a base pointer register, such as the C60's DP (B14).

An additional consideration applies to modules that will be located in ROM. Obviously code in ROM cannot be patched at load time, so it has many similar requirements for position independence.

6.2   Overview of Dynamic Linking Mechanisms

The ABI addresses these issues through several related mechanisms:

The general **ELF Dynamic linking** mechanisms define the object file representations to support load-time symbol resolution and relocation. Most of these are target independent and are specified by the GABI. Target specific aspects are described in Section 14.3.

A *Procedure Linkage Table (PLT) Entry* is a linker-generated stub used to resolve calls to imported functions.

The *Global Offset Table (GOT)* is an addressing method for referencing imported objects that supports position independence and privatization by placing address constants in a table in the data section rather than encoding them into the code. These benefits come at the cost of an extra indirection for a GOT-based reference, plus the additional data space for the table.

The *Data Segment Base Table (DSBT) model* is a software convention that allows each component to have its own dedicated data segment, so references to its own data can be statically resolved without regard to other components. The DSBT mechanism enables position-independent code without virtual memory, enabling a single instance of shared code to address multiple copies of dynamically-bound private data.

6.3   DSOs and DLLs

Systems differ in the dynamic linking models they support. In UNIX systems, including Linux, dynamic linking is designed to be transparent from the application's point of view. That is, a program or library can be written and compiled without regard to whether any unresolved references will be resolved statically or dynamically. For example if a program declares "extern int f()" and then calls f, the compiler generates code that enables f to be resolved either statically or dynamically. The main advantage to this approach is flexibility: programs can be written and compiled without regard to how they will be linked. The main drawback is that it may be less efficient, as the compiler must assume that any extern reference may not be resolved statically, and generate the appropriate addressing code to support dynamic linking.

Unix refers to dynamically linked libraries as Dynamic Shared Objects, or *DSOs.*

In Windows and various embedded systems such as Symbian and PalmOS, dynamic linking is explicitly specified in the source code for a symbol declaration via a language extension, usually _ _declspec(import). The advantage of this approach is that the compiler explicitly knows when to generate the special addressing required. These systems commonly have a post-link phase that replaces dynamic linkage via symbolic references with a symbol indexing scheme. These systems refer to shared libraries as Dynamic Link Libraries, or *DLLs.*

6.4   Preemption

When an object refers to a global symbol defined in another object, it is said to import that symbol, and the defining object is said to export it. Suppose two different objects define and export the same symbol. One of the definitions takes priority and *preempts* the other. Preemption enables dynamic linkage to behave identically to static linkage: the executable's definition preempts that of the library, so the library's instance is not linked in. In the dynamic linking case the library may already be loaded, and a definition in a shared instance may be needed by one client but not by another.

Preemption means that even though at static link time a symbol appears to be defined within the module, in fact it may be replaced by a different definition at dynamic link time. This has implications for the compiler, which must generate code as if the symbol were imported. For this reason preemption is expensive, even when it does not actually occur. The performance impact is discussed in Section 6.8. Linux uses a technique called import-as-own, discussed in Section 15.9, to alleviate the penalty for the executable.

The symbol visibility field in the ELF symbol table indicates a symbol's preemptability. Symbols marked as STV_HIDDEN or STV_INTERNAL are not exported (and therefore not preemptable). Symbols marked STV_PROTECTED are exported, but cannot be preempted. Symbols marked as STV_DEFAULT can be preempted.

Different platform and toolchain-specific conventions apply to which symbols can be preempted and how the programmer specifies visibility.

6.5   PLT Entries

Typically when the compiler sees a call to an extern function, it simply generates a CALL instruction without regard to where the called function is. During static linking, if the function is defined in another source file or within a statically-linked library, the linker simply relocates the displacement field in the CALL instruction to resolve the reference.

If the function is imported from a shared library, its address is unknown at static link time, eventually being resolved at dynamic link time. Additional instructions may be required to address and call imported functions. For this possibility, and to avoid having to patch the call at dynamic link time, the static linker instead generates a position-independent stub to call the function, and patches the original call to go through the stub. This stub is called a *PLT entry*. PLT stands for *Procedure Linkage Table*. (The designation of the PLT as a table is historical; its entries are independently generated code fragments and are not collected into any cohesive entity.) A PLT entry is conceptually similar to a far-call trampoline (see Section 3.1). Whereas the purpose of a trampoline is to call a *far-away* function, a PLT entry calls an *imported* function.

6.5.1   Direct Calls to Imported Functions

PLT stubs are generated into the code segment where the call occurs. The PLT encodes the address of the destination function according to the considerations in Section 5.1.

6.5.2   PLT Entry Via Absolute Address

```
$sym$plt:
        MVKL    sym,tmp                   ;reloc R_C6000_ABS_L16
        MVKH    sym,tmp                   ;reloc R_C6000_ABS_H16
        B       tmp
```

With one subtle distinction discussed in the following section involving the choice of tmp, this code sequence is identical to a far-call trampoline.

6.5.3   PLT Entry Via GOT

If the function can be preempted, the function's address cannot be encoded in the PLT entry, even in a position independent way. The address must be addressed indirectly through the GOT.

```
$sym$plt:
        LDW     *+DP($GOT(sym)),tmp       ;reloc R_C6000_SBR_GOT_U15
        B       tmp
```

Certain compiler helper functions have non-standard register preservation conventions (Section 8.3), affecting the choice of which register is used for tmp. Furthermore, lazy binding (Section 15.6) may affect additional registers beyond those directly mentioned in the PLT entry. For this reason the ABI specifies that functions with non-standard conventions cannot be imported; that is, they cannot be called via a PLT entry. With this stipulation the linker is free to modify any caller-save register not involved in the function-call interface in the PLT entry.

A compiler may choose to inline the PLT entry for calls to functions that it knows or suspects are imported. This has the advantage of reducing the latency for the additional branch, at the expense of code size.

If the dynamic loader uses lazy binding as described in Section 15.6, inlined PLT entries must follow the conventions described there. Alternately, inlined PLTs can generate GOT relocations that are excluded from the DT_JMPREL part of the dynamic relocation table (see *Dynamic Section* in Chapter 5 of the System V ABI) so that they are not subject to lazy binding.

6.6   The Global Offset Table

Full position independence implies that code is independent of its own location, the location of its own data, and the location of any imported code or data, without requiring relocation patches at load time. In this context the word own means part of the same static link unit as the reference. Let's examine the implications of each case:

- References to own code (Section 5.1): PC-relative addressing or GOT-based addressing must be used. No absolute addresses may be used. This case affects trampolines, switch tables, and return address calculations.
- References to own data (Section 4.2): DP-relative, PC-relative, or GOT-based addressing must be used. No absolute addresses may be used. Generally the choice must be made at compile time. This case affects references to near and far data.
- References to imported code: No absolute or PC-relative addresses may be used. This case applies to the call generated in a PLT entry.
- References to imported data: no absolute or DP-relative addresses may be used. This case applies to any reference to imported data.

To avoid encoding position-dependent absolute addresses into the code segment, they are generated into a table called the Global Offset Table (GOT) which is part of each static link unit's data segment. Instead of accessing the object directly, a program reads the symbol's address from the GOT and addresses it indirectly. The GOT is part of the data segment and is always addressed DP-relative using offsets that are fixed at static link time. It is generated by the linker in response to special GOT-generating relocations emitted by the compiler. The addresses in the GOT are patched at dynamic link time when the addresses are known.

A GOT-based access involves two memory references: one to load the address from the GOT, and another to reference the variable itself. The first reference, to access the GOT itself, is essentially the same as a normal DP-relative data access (see Section 4.2.1). The vast majority of the time, we expect the GOT to be in a near-DP segment, and therefore accessible using near DP-relative addressing.

6.6.1   GOT-Based Reference Using Near DP-Relative Addressing

A complete GOT-based reference using near DP-relative addressing form looks like this:

```
LDW    *+DP($GOT(sym)),tmp      ;reloc R_C6000_SBR_GOT_U15
LDW    *tmp,dest
```

The relocation indicated here causes the static linker to allocate a GOT entry and evaluate to its DP-relative offset. The table entry itself is marked with a dynamic relocation that evaluates to the address of the symbol.

6.6.2   GOT-Based Reference Using Far DP-Relative Addressing

For completeness, the ABI also supports GOT-based addressing when the GOT itself is far; that is, outside the 15-bit offset range of the DP. In this case far DP-relative addressing is used to reach the GOT:

```
MVKL   $DPR_GOT(sym),tmp        ;reloc R_C6000_SBR_GOT_L16
MVKH   $DPR_GOT(sym),tmp        ;reloc R_C6000_SBR_GOT_H16
LDW    *+DP[tmp],tmp2
LDW    *tmp2,dest
```

6.7   The DSBT Model

Each executable that shares a library's code must allocate its own private copy of the library's data. Furthermore, each static link unit's own data (including the GOT) is addressed using DP-relative addressing, with offsets that are fixed at static link time. (On systems with MMU's, this is typically accomplished by using PC-relative addressing to achieve position-independent virtual offsets, and using address translation to instantiate multiple physical copies of the data segment at the same (virtual) address.) Systems without an MMU, like the C6000, typically rely on some form of a static base pointer of some kind (the DP) and offset addressing.

All addressing from a given static link unit is relative to its data segment and is therefore independent of any other static link unit. The result is a model where a given program, comprised of an executable and one or more (possibly shared) libraries, has multiple data segments, each having a different address on which DP-relative offsets are based. When control transfers from one module to another, the DP must be changed to the base address of the new module's data segment.

The general issues with this model, common to most static-base addressing schemes, are:

* Who changes the DP: the caller, or the callee
* How is the new DP value determined
* How are indirect calls handled

Various solutions have been adopted for other architectures, such as FDPIC, XFLAT, and DSBT. We have chosen to adopt the DSBT model as the best compromise between efficiency, compatibility, and flexibility.

When a call to an imported function is made, the callee is responsible for setting the DP to point to its data segment (more precisely, the underlying executable's private copy of the data segment for the module containing the callee), and for restoring it upon return.

Before we explain how the callee achieves this, consider two observations. First, each module has its own data segment(s), with its own base address, and if shared among multiple executables, each has a private copy of that segment, with a different base address. Furthermore, these addresses are dynamically determined. So obviously, much like addresses stored in the GOT, the base address cannot be absolute and therefore must be stored in the data segment.

Second, although the *callee* is responsible for changing the DP, upon entry to the *callee* the DP is still pointing to the caller's data segment. Thus the callee has only the context of the caller to somehow set up its *own* DP.

The solution is that the first few words of each data segment contain a copy of a table, called the Data Segment Base Table (DSBT), listing the base addresses for all the other segments of the other modules that comprise the program. Each shared library is assigned a unique index, starting with 1. Index 0 is reserved for the executable. The callee uses its assigned index to lookup its own base address in the caller's copy of the table, and assigns that value to the DP. Within the private data segments for a given executable and its shared libraries, each copy of the DSBT is identical, enabling any callee to use any caller's table to find its own base address.

The DSBT approach has the desirable characteristic that the penalty for dynamic linking is isolated to exported functions. There is no effect on the ABI for bare-metal programs that do not use dynamic linking, or for an executable without exported functions. By judiciously using toolchain-specific declaration constructs to explicitly identify externally-accessible functions (see Section 6.7.2), the programmer can minimize the overhead. In functions that do need to adjust the DP, the overhead is typically only 3 instructions.

The drawback of the DSBT model is the requirement to coordinate the assignment of the library indexes and to enforce agreement on the maximum number of modules, which determines the size of the table in each data segment.

The DSBT is allocated by the static linker in the .dsbt section, and must be located at the base address of each module's first DP-relative segment so that the DP points to it. The dynamic linker initializes the table entries when the module is loaded.

An executable always accesses the table using index 0; library indexes start at 1, 2, or some other index as specified for a specific platform. A library's index may be assigned in one of two ways:

- It can be statically assigned at static link time (or equivalently, by a static post-link tool) via a command line option or other directive. This method must be used when the library is ROM-resident and cannot be relocated at dynamic load time.

- It can be dynamically assigned by the dynamic linker. This requires relocating (patching) the library's code segment when it is loaded, in order to update the indexes, so libraries with dynamically-assigned indexes are not considered position independent.

Each object's DSBT must be at least as large as the largest index assigned to any module that is dynamically loaded as part of the program. The dynamic linker is responsible for ensuring that all modules have a large enough DSBT; if not, it must fail to load the program. The size of the DSBT is specified at static link time (or to a static post-link tool) via a command line option or environment variable. Embedded systems generally require a small number of dynamic libraries; so a typical size for the DSBT is 5 or less.

The module's dynamic section contains C6000-specific tags that specify the size of its DSBT table, and its index if assigned. These are detailed in Section 14.3.2.

6.7.1 Entry/Exit Sequence for Exported Functions

The following code sequences illustrate how an exported function changes the DP to point to its data segment by indexing into its caller's DSBT. Any function that changes DP is responsible for restoring it upon return (the DP is callee-save).

Entry Sequence to Setup DP

```
func:
        MV      DP,somewhere               ;typically the stack
        LD      *+DP[$DSBT_index(func)],DP ;reloc R_C6000_DSBT_INDEX
        ; body of function
```

The expression $DSBT_index(func) evaluates to the unique library index of the current object and generates a special relocation to indicate this. The index will be bound either at static link time or dynamically.

Exit Sequence

```
        MV      somewhere, DP
        RET
```

The exit sequence simply restores the caller's DP.

An exported function may choose not to change the DP if it does not use any DP-relative addressing, *and* it does not call any functions that use DP-relative addressing.

6.7.2 Avoiding DP Loads for Internal Functions

Only functions that can be called from another link unit need to adjust the DP. Functions that can be called only from within their static link unit do not need to adjust the DP, since they can rely on their caller to have done so. A function's ability to be externally called is known as its *visibility*. (Note that visibility also applies to an object's ability to be preempted; see Section 6.4.)

An external call from another link unit can be direct, in which case the function is called by name, or indirect, in which case the function's address is taken and passed to the external caller, who calls it through this address. ELF provides four levels of visibility, which cover various possibilities for direct and indirect calls from other modules as summarized in Table 6:

Table 6. Interpretation of ELF Visibility Attributes

Name	Directly Callable	Indirectly Callable	Preemptable
STV_DEFAULT	yes	yes	yes
STV_PROTECTED	yes	yes	no
STV_HIDDEN	no	yes	no
STV_INTERNAL	no	no	no

A function's visibility is determined by a combination of its declaration and a set of compiler and platform specific conventions. For example, the Linux model is that an external function has STV_DEFAULT visibility unless otherwise indicated by augmenting its declaration with an _ _attribute_ _((visibility)) modifier; but for bare-metal platforms a default visibility of STV_HIDDEN or STV_INTERNAL may be more appropriate.

6.7.3   Function Pointers

In general since callees are responsible for setting up their own DP, no special handling is required for function pointers. Exported functions can safely be called indirectly from inside or outside the module where they are defined. (This is a major advantage of the DSBT model over some of the other MMU-less approaches.)

However, there is a potential pitfall in the use of function pointers. If a function with internal visibility has its address taken, passed to another static link unit, and then indirectly called, it likely will not set the DP properly and the program will fail.

Taking the address of an *internal* function and making it available to another module is, strictly speaking, a programming error since it violates the assumptions implied by the visibility declaration. To aid in detecting such violations the toolchain may choose to have the compiler issue a warning when the address of a non-exported function is taken. Users who are doing so legitimately can disable the warning.

Taking the address of an *external* function and passing it to another module is always legitimate. To permit comparison of function pointers computed in different modules to work as expected, the ABI requires that an expression representing the address of a function evaluates to a unique value across all modules. Some ABIs for other architectures adopt a convention that, within an executable, references to the address of a function may resolve to the PLT entry, allowing for static resolution of those references. (References from a shared object must resolve dynamically, due to preemption).

The C6000 ABI does not adopt that convention because it leads to a problem with cross-module calls through a function pointer. If such a pointer could resolve to a PLT entry, then when an indirect call lands at that PLT entry the DP value may be that of a different static link unit, preventing the PLT entry from being able to access the GOT. In effect the PLT entry is an internal function so it must not be called indirectly from outside the module.

Therefore, the convention for the C6000 ABI is that a reference to the address of a function must resolve to the function's actual address. The implication is that for imported objects, such references cannot be statically resolved; they must be resolved at load time by the dynamic linker.

6.7.4   Interrupts

In a standalone application with no shared libraries, the DP never changes. Assuming this convention holds throughout the system, an interrupt service routine could reliably assume the DP points to the one and only RW segment.

In the presence of dynamic linking, an interrupt routine cannot assume anything about the DP. It must save, setup, and restore the DP for itself like any other exported function.

6.7.5   Compatibility With Non-DSBT Code

The DSBT model is provided as a variant to the ABI in order to support position independence and shared libraries. Many embedded systems do not require these features, and therefore can avoid the added complexity and performance overhead. Code that uses the DSBT model is not binary compatible with code that does not. A build attribute in the object file indicates that it is built using the DSBT model; linkers and loaders should prevent DSBT code from being mixed with non-DSBT code.

6.8 *Performance Implications of Dynamic Linking*

There is a performance penalty for dynamic linking. Imported functions called via the PLT incur the overhead of an additional call, similar to a trampoline. If the function's address is accessed through the GOT, there also the overhead of an indirect access to load its address.

There is no penalty for near data addressed via DP. For far data, DP-relative addressing requires three instructions, versus two for position-dependent absolute addressing. For objects addressed via the GOT, there is the overhead of an additional reference to the GOT to load the address.

Symbol preemption significantly exacerbates the GOT penalty. Any symbol that may be preempted—that is, any global symbol defined in a shared library—must be treated by the compiler and static linker as if it were imported. Even a locally defined function must be called via the PLT, thereby precluding inlining or specialization. A locally defined variable must be accessed indirectly via the GOT. These restrictions apply to the code generated by the compiler so the losses generally cannot be recovered even if the symbol is not ultimately preempted.

The penalty due to preemption applies only to shared libraries. Symbols defined in an executable (that is, not a library), cannot be preempted.

Systems employ a handful of techniques to mitigate these effects. In some systems that follow the DLL model (Windows, Palm, Symbian) defined symbols are not considered exported unless specifically declared so.

In UNIX systems (including Linux), all external symbols are potentially dynamically linked, meaning a compiler must generate the inefficient GOT indirection for all such symbols. To alleviate this effect, the UNIX model adopts the import-as-own model, described in Section 15.9.

Toolchains may adopt additional vendor-specific ways of alleviating the preemption penalty, such as options or declaration specifiers that alter the default visibility of extern symbols.

The DSBT model introduces overhead in that exported functions must save and restore the DP, a cost of 3 instructions and 2 memory references. There is also the data size overhead of the table itself, which adds N+1 words to the data segment of each executable and library, where N is the maximum index of any library used by the application.

7 Thread-Local Storage Allocation and Addressing

Multi-threaded programming is common in many embedded systems that use the C6000 family of processors. Given the increase in the number of C6000 CPU based multi-core devices, multi-threaded programming is expected to be even more widely adopted to leverage the multiple cores. Also, multi-core programming models like OpenMP and OpenCL rely on underlying multi-threading support.

Complex multi-threaded programs can be better structured and easier to develop if the threads can use variables with static storage duration and that are specific to the thread. That is, other threads cannot see or access such thread-specific variables with static storage duration. Consider the following C code:

```
int global_x;

foo()  {
    int local_x;
    static int static_x = 0;
    ...
}
```

The global_x and static_x variables are allocated once per process, and all threads share the same instance. In contrast, local_x is allocated from the stack. Since each thread gets its own stack, the variable local_x is thread specific, while static_x is not. However, there is no easy way to define a global/static variable on a per thread basis. The POSIX thread interface allows creating thread-specific static storage variables using pthread getspecific and pthread setspecific. But this interface is cumbersome to use.

To solve this issue, Thread-Local Storage (TLS) is a class of storage that allows a program to define thread-specific variables with static storage durations. A TLS variable or "thread-local" is a global/static variable that is instanced once per thread.

Memory used for TLS is allocated statically for the full time the program runs. Each thread has its own instance of *all* the thread-local variables (even the ones it doesn't declare or use) that are defined by all of the dynamic modules that are loaded at the time a thread is created. When a thread is created, its TLS block is allocated and initialized by the underlying OS thread support library. A thread's TLS block is reinitialized if a thread completes and then runs again within the same program run. TLS variables are not re-initialized if the thread is suspended or blocked by other threads and then resumes execution after it becomes un-blocked.

The way a TLS variable is accessed depends on how the OS or RTOS creates and manages thread-local storage for each thread. Linux systems need to support TLS allocation for multiple dynamic libraries and libraries loaded during runtime using dlopen(). Also, Linux systems may require allocating TLS storage lazily only when the thread-local is accessed. This requires sophisticated TLS storage management and affects how the thread-local is accessed. On the other hand, a static executable that includes an RTOS needs only to manage a single TLS block and the access can be simple.

After an overview of thread-local concepts, this document describes how thread-locals are specified in source code and how they are represented in the ELF object file (Section 7.3). Then it describes how thread-locals are accessed for C6x Linux, static executable, and bare-metal dynamic linking TLS models (Section 7.4) and how weak references to thread-local variables are resolved. (Section 7.5).

The C6000 TLS mechanism is based on industry-standard conventions, for example the mechanism described in the ELF Handling for Thread-Local Storage paper by Ulrich Drepper.

7.1 Terms and Concepts

Thread-local variables are thread-specific and have static storage duration. They must be allocated similarly to global and static variables that are allocated in the .neardata or .fardata section if they are initialized and .bss if uninitialized. Global and static variables have only one copy per process, whereas thread-locals need a separate instance per thread.

Threads are created by a thread manager when a program calls for the creation of a thread. For example, a parallel region in an OpenMP application makes an OS thread library call to create worker threads; these worker threads will join/merge at the bottom of the parallel region.

During thread creation, the storage for thread-locals must be allocated and initialized. This means there needs to be an initialization image for use in initializing per thread TLS storage. The output of the static linker, the static link unit, must contain a TLS initialization image if thread-local storage is used. The static link unit is referred to as a *module*.

The TLS initialization image for a single module is called a *TLS Image*. TLS is allocated for each thread as part of the thread creation and is initialized with the data from the TLS Image. The memory allocated per thread for thread-local variables from a single module is called a *TLS Block*.

In the static executable model, the static linker produces an executable that is loaded and executed from the start address. RTOS and/or thread libraries are linked in as part of the executable. In this case, there is only one module and hence only one TLS Image and one TLS Block. This simplifies TLS access. The main thread is usually created at program initialization; other threads are created by the threads library. The main thread's TLS Block should be allocated and initialized by the program loader (see Section 14.2). It is the responsibility of the thread library to allocate and initialize TLS for the threads it creates.

In a C6x Linux system, a program (process) is created by loading multiple modules: an executable and zero or more dynamic libraries. Each module can have a TLS Image. The program's TLS Image consists of all the modules' TLS Images. This is called a *TLS Template*. Normally, the executable and all dependent modules are loaded at process startup. These are called *initially loaded modules*. A Linux program can also load a dynamic library after startup by calling the dlopen() system function. Modules loaded after startup are called *dlopened modules*. During thread creation, TLS blocks are created based on the TLS Template. The run-time structure consisting of TLS Blocks from all modules is called *TLS*.

In the case of bare-metal dynamic linking, by default, there are only initially loaded modules and they can be consecutively placed to form the TLS Template.

See Section 14 for more information on C6x program loading and dynamic linking.

7.2   User Interface

Over the years, C and C++ have been extended to allow definition of thread-local variables:

* Compilers for Linux systems (GCC, Sun, IBM, and Intel) support the __thread storage qualifier as a C/C++ language extension. This is not an official language extension, however.
* Compilers for Windows (MS VC++, Intel, Borland) support the __declspec(thread) storage attribute extension.
* The latest C++ standard, C++11 (ISO/IEC 14882:2011), introduces the thread_local storage class specifier.
* The latest C standard, C11, introduces the _Thread_local storage class specifier.

The language extension used to support thread-local storage is toolchain-specific and outside the scope of the ABI.

Thread-local variables can be initialized or uninitialized. Uninitialized thread-local variables are initialized to zero, as with uninitialized global and static variables. Allocation and initialization of thread-local variables occurs when the thread is created, whether statically or dynamically.

7.3   ELF Object File Representation

The ELF specification (www.sco.com/developers/gabi/) provides details on how thread-local storage is represented in ELF relocatable object files and ELF modules.

To summarize the relevant portions of the ELF specification, thread-local variables are represented in the object files and ELF modules similarly to static data. The difference is that ELF requires that thread-local variables be allocated in sections with the SHF_TLS flag set in relocatable files. Also, the ELF specification requires that the section names .tdata and .tbss be used for initialized and uninitialized thread-local storage, respectively. These sections have read-write permission.

In modules, ELF requires that the TLS segment have the PT_TLS segment type. This segment is read-only. The PT_TLS segment is the TLS Image.

Thread-local symbols have the symbol type STT_TLS.

7.4   TLS Access Models

Each thread has its own instance of *all* the thread-local variables (even the ones it doesn't declare or use). An access to a thread-local variable should access the current thread's instance of that thread-local variable. This means the thread-local access needs to find the current thread's TLS and access the variable using an offset into the TLS block where the variable is defined.

There are six models for accessing TLS data, depending on factors such as whether DLLs/DSOs are supported (separate linking), whether dlopen() is supported, and whether the access is to own or imported data. Table 7 summarizes various characteristics of the TLS access models.

Table 7. Thread-Local Storage Addressing Models

Model	General Dynamic	Local Dynamic	Initial Exec	Local Exec	Static Exec	Bare Metal Dynamic
System has DLLs or DSOs	yes	yes	yes	yes	no	yes
Can access another module's TLS data	yes	no	yes	no	no	yes
TLS access from DLLs or DSOs	yes	yes	yes	no	no	yes
TLS access from dlopened modules	yes	yes	no	no	no	no
TLS initialization performed by	loader	loader	loader	loader	linker	loader
Supports weak references	yes	yes	no	no	no	no
Use case	Access another module's TLS data	Access own TLS data from DLL or DSO	Access TLS data of module loaded at load-time	Access own TLS data from executable	No DLLs or DSOs	Access TLS data of module loaded at load-time
Section	Section 7.4.1.1	Section 7.4.1.2	Section 7.4.1.3	Section 7.4.1.4	Section 7.4.2	Section 7.4.3

The C6x Linux TLS access model needs to satisfy more constraints and can be complex. It needs to conform to already-established conventions. The static executable access model, on the other hand, is simple—there is only one TLS block, and any thread-local variable can be accessed using Thread Pointer Relative (TPR) addressing. It is useful to first describe the more complex C6x Linux TLS model (Section 7.4.1) and then describe the static executable TLS model as a simpler case (Section 7.4.2). Finally, the bare-metal dynamic linking case is described (Section 7.4.3).

There are four widely-used TLS access models discussed in the literature. [1] [2] These are:

- General Dynamic TLS access model (Section 7.4.1.1)
- Local Dynamic TLS access model (Section 7.4.1.2)
- Initial Exec TLS access model (Section 7.4.1.3)
- Local Exec TLS access model (Section 7.4.1.4)

The full list of relocations used for TLS are listed in Table 30 and Table 31. The sections that follow show the use of these relocations.

7.4.1   C6x Linux TLS Models

In some dynamic linking models, including Linux, a module can be loaded during run-time using dlopen(). The TLS block from the dlopened module is dynamically allocated and so cannot be allocated at a fixed offset from the TP for all the threads. Hence the access to a thread-local variable is by reference using the module identifier and the offset of the thread-local variable in the module's TLS block.

[1]   Ulrich Drepper, *ELF Handling for Thread-Local Storage*, http://www.uclibc.org/docs/tls.pdf, 2005, Version 0.20

[2]   Alexandre Olivia and Glauber de Oliveira Costa, *Speeding Up Thread-Local Storage Access in Dynamic Libraries in the ARM Platform*, http://www.fsfla.org/~lxoliva/writeups/TLS/paper-lk2006.pdf, 2006.

Figure 6 shows the C6x Linux TLS run-time representation. Each thread has an instance of this run-time TLS structure.
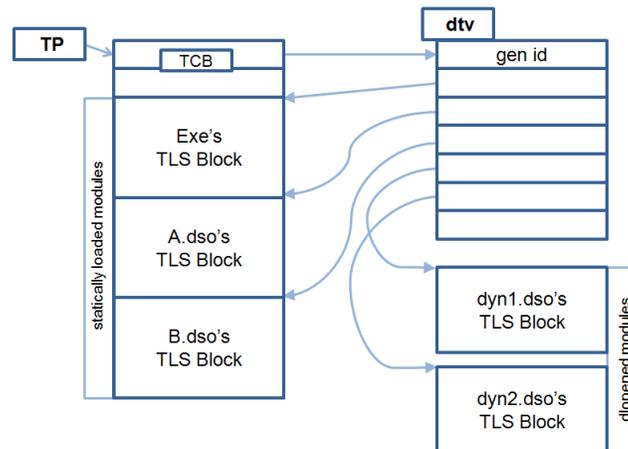
Figure 6. C6x Linux TLS Run-Time Representation

For each thread, the Thread Pointer (TP) points to the Thread Control Block (TCB). The executable's TLS block, if it exists, is placed after the TCB after adjusting for the alignment. TLS blocks from other non-dynamic modules are placed subsequently honoring their alignment requirement. The TCB and the TLS blocks that follow for the static modules constitute the program's static TLS. The static TLS for a thread is created as part of the thread creation.

The TCB is 64 bits wide. The first 32 bits point to the Dynamic Thread Vector (dtv). The remaining 32 bits are reserved.

The dtv pointed to by the TCB is a vector of 32-bit elements. The dtv[0] element is the generation ID, which is used to manage the dynamic growth of the dtv as dlopened modules are loaded. The dtv[n] elements, where n != 0, are 32-bit pointers to the TLS block for module n. When a module with TLS data is loaded, a module ID is assigned to that module. This module ID is process-specific. A dynamic shared library that is shared by multiple processes can have different module IDs in each process. Module ID 1 is always assigned to the executable.

The main thread is created by the dynamic loader, and subsequent threads are created by the thread library. When the main thread is created, the dtv array needs to contain only pointers to the initially loaded modules.

When a thread dlopens a new module, the module's TLS block should be allocated for all threads in the process. This is needed in case the other threads access this new module's thread-local data. However, allocating the TLS block of the dlopened module can be deferred until the first time the storage it is accessed. This can be done by initializing the appropriate dtv[module-id] to TLS_DTV_UNALLOCATED. The __tls_get_addr() function can check to see if dtv[module_id] is TLS_DTV_UNALLOCATED; if so, it allocates and initializes the TLS block for the current thread.

7.4.1.1   General Dynamic TLS Access Model

This is the most generic TLS access model. Objects using this access model can be used to build any Linux module: executables, initially loaded modules, and dlopened modules. The generated code for this model cannot assume the module-id or the offset is known during static linking.

With this access model, a dynamic module can be loaded at run time. To allow for this possibility, the thread library's thread management architecture must provide a way for TLS blocks to be added and removed as dynamic modules are loaded and unloaded at run-time.

The compiler generates a call to __tls_get_addr() to get the address of the thread-local variable. The module-id and the thread-local variable's offset in the module's TLS block are passed as parameters. The code obtains the module-id and offset from the Global Offset Table (GOT) entries to ensure position independence (PIC) and symbol preemption.

The simplest way for the __tls_get_addr() function to pass the module-id and offset is as follows:

```
void * __tls_get_addr(unsigned int module_id, ptrdiff_t offset);
```

Note that both are 32-bit arguments, and the GOT entries are also 32-bit entries. As an optimization, we can load these two GOT entries as a 64-bit double word if the ISA supports this. The two GOT entries must be allocated consecutively and aligned to a 64-bit boundary. This GOT entity can be thought of as the following struct:

```
struct TLS_descriptor
{
    unsigned int module_id;
    ptrditt_t offset;
} __attribute__ ((aligned (8)));
```

Then the __tls_get_addr() interface becomes:

```
void * __tls_get_addr(struct TLS_descriptor);
```

In this EABI, a struct of size 64 bits or less is passed by value, resulting in passing the TLS descriptor in the A5:A4 register pair. In little-endian mode, the module-id is passed in A4 and the offset is in A5. In big-endian mode, the registers are swapped as per the C6x EABI calling conventions. The examples in this section use little-endian mode.

Using this interface, the thread-local access becomes the following (for C64 and above):

```
      LDDW   *+DP($GOT_TLS(X)), A5:A4  ;reloc R_C6000_SBR_GOT_U15_D_TLS
   || CALLP  __tls_get_addr,B3          ; A4 has the address of X at return
      LDW    *A4, A4                    ; A4 has the value of X
```

The relocation R_C6000_SBR_GOT_U15_D_TLS causes the linker to create GOT entries for the module-id and offset for x as follows:

```
64-bit aligned address:
      GOT[n]              ;reloc R_C6000_TLSMOD  (symbol X)
      GOT[n+1]            ;reloc R_C6000_TBR_U32 (symbol X)
```

The linker then resolves the R_C6000_SBR_GOT_U15_D_TLS relocation with the DP-relative offset of the GOT entity. The dynamic loader resolves R_C6000_TLSMOD to the module-id of the module where x is defined. It resolves R_C6000_TBR_U32 to the offset of x in the module's TLS block.

The C6x ISA does not currently have an instruction to load the 64-bit TLS descriptor directly. However, we define the __tls_get_addr() interface using the 64-bit descriptor in anticipation of a future ISA having such support.

```
void * __tls_get_addr(struct TLS_descriptor);
```

The linker is required to allocate the GOT entries of a thread-local variable's module-id and offset consecutively and align the first entry to a 64-bit boundary when the R_C6000_SBR_GOT_U15_D_TLS relocation is found.

Lacking support for a DP-relative 64-bit load, the following sequence can be used on current ISAs:

```
      LDW    *+DP($GOT_TLSMOD(X)), A5   ;reloc R_C6000_SBR_GOT_U15_W_TLSMOD
      LDW    *+DP($GOT_TBR(X)), A4      ;reloc R_C6000_SBR_GOT_U15_W_TBR
   || CALLP  __tls_get_addr,B3          ; A4 has the address of X at return
      LDW    *A4, A4                    ; A4 has the value of X
```

The relocations R_C6000_SBR_GOT_U15_W_TLSMOD and R_C6000_SBR_GOT_U15_W_TBR cause the linker to create GOT entries for the module-id and offset respectively for x. This access mode does not require these GOT entries to be consecutive and 64-bit aligned. If the linker does not also see a DW_TLS relocation for the same symbol, it is free to define the module-id and offset GOT entries separately without 64-bit alignment. However, if it sees DW_TLS in addition to the TLSMOD/TBR relocations for the same symbol, 64-bit aligned consecutive GOT entries must be defined and reused for the TLSMOD/TBR relocations.

If the GOT must be addressed using far-DP addressing, then the general dynamic addressing becomes:

```
      MVKL $DPR_GOT_TLSMOD(X),  A5       ;reloc R_C6000_SBR_GOT_L16_W_TLSMOD
      MVKH $DPR_GOT_TLSMOD(X),  A5       ;reloc R_C6000_SBR_GOT_H16_W_TLSMOD
      ADD  DP, A5, A5
      LDW  *A5, A5
```

```
          MVKL $DPR_GOT_TPR(X),  A4        ;reloc R_C6000_SBR_GOT_L16_W_TBR
          MVKH $DPR_GOT_TPR(X),  A4        ;reloc R_C6000_SBR_GOT_H16_W_TBR
          ADD  DP, A4, A4
          LDW  *A4, A4

       || CALLP  __tls_get_addr,B3         ; A4 has the address of X at return
          LDW    *A4, A4                   ; A4 has the value of X
```

__tls_get_addr() can calculate the thread-local address as follows:

```
    void * __tls_get_addr(struct TLS_descriptor desc)
    {
        void *TP  = __c6xabi_get_tp();
        int  *dtv = (int*)(((int*) TP)[0]);
        char *tls = (char *)dtv[desc.module_id];
        return tls + desc.offset;
    }
```

7.4.1.2 *Local Dynamic TLS Access Model*

This access model is an optimization of the General Dynamic Model to access a module's own data. If the compiler knows it is accessing a module's own thread-local storage, then this access model can be used. If the thread-local variable is defined in the same module where it is accessed, then the TLS offset is known at static link time. However, the module-id is not known at static link time.

A call to __tls_get_addr() with an offset argument of zero returns the base address of that module's TLS block. This base address can be used to access all the thread-local data belonging to that module.

At compile time, the thread's own data is identified using the symbol binding and visibility. Symbols with static scope or hidden/protected visibility are own data. In this model, thread-local x can be accessed as follows:

```
          LDW  *+DP($GOT_TLSMOD(x)), A4     ; reloc R_C6000_SBR_GOT_U15_W_TLSMOD
          MVK  $TBR_word(x), A5             ; reloc R_C6000_TBR_U15_W
       || CALLP    __tls_get_addr,B3   ; A4 has the address of x at return
```

As mentioned previously, the own TLS base can be obtained once and reused to access other own thread-local variables as follows:

```
          LDW  *+DP($GOT_TLSMOD()), A4     ; reloc R_C6000_SBR_GOT_U15_W_TLSMOD w/ Symbol=0
          MVK  0x0, A5                     ;
       || CALLP    __tls_get_addr,B3 ; A4 has the module's own TLS base

          MVK  $TBR_byte(x), A5            ; reloc R_C6000_TBR_U15_B; Get x's scaled TLS offset
          LDB  *A4[A5], A6                 ; A6 has the value of thread-local char x

          MVK  $TBR_hword(y), A5           ; reloc R_C6000_TBR_U15_H; Get y's scaled TLS offset
          LDH  *A4[A5], A6                 ; A6 has the value of thread-local short y

          MVK  $TBR_word(z), A5            ; reloc R_C6000_TBR_U15_W; Get z's scaled TLS offset
          LDW  *A4[A5], A6                 ; A6 has the value of thread-local int z

          MVK  $TBR_dword(l), A5           ; reloc R_C6000_TBR_U15_D; Get l's scaled TLS offset
          LDDW *A4[A5], A7:A6              ; A7:A6 has the value of thread-local long long l
```

The relocation R_C6000_SBR_GOT_U15_W_TLSMOD resolves to the module's own module-id when the symbol is zero. The TBR_U15 relocations encode a 15-bit unsigned offset from the module's TLS Base for near TB (TLS Block Base) addressing. They are scaled according to the access width. The previous addressing can access a TLS block of size 32 KB. This specification limits the size of each module's TLS block to 32 KB, a limit that is expected to be sufficient for most use cases. Hence the far TB relative address is not defined. Far TBR addressing may be defined, but it will use up 8 new relocations, and it is better to conserve the limited number of relocations (256) ELF allows.

The static linker resolves all the TBR relocations using static-only relocations. That is, these relocations cannot be in the dynamic relocation table.

7.4.1.3   Initial Exec TLS Access Model

Objects that are used to build initially-loaded modules can use this access model. Modules that use this access model cannot be dlopened.

Since the module will always be initially loaded and the dynamic loader can allocate TLS blocks from initial modules consecutively after the executable's TLS block, the offset from the thread pointer is known at dynamic link time. The thread-local variable can be accessed using *(TP + offset), where the offset is loaded from the GOT to ensure PIC and symbol preemption. Modules built with this type of addressing cannot be dlopened. Such modules are marked with the dynamic flag DF_STATIC_TLS, and the dynamic loader will refuse to dlopen modules marked DF_STATIC_TLS.

7.4.1.3.1   Thread Pointer

The addressing used for the Initial Exec model needs a way to obtain the thread pointer of the current thread. A new c6xabi function, __c6xabi_get_tp(), returns the thread pointer value for the current thread. This function does not modify any register other than the return register A4. This function can be called via the PLT, so the caller should assume that the B30 and B31 registers are modified by the call to this function. This function has the following signature:

```
void * __c6xabi_get_tp(void);
```

The thread library is responsible for providing a definition of this function.

7.4.1.3.2   Initial Exec TLS Addressing

In the Initial Exec model, the thread-local variable is accessed as follows:

```
callp  __c6xabi_get_tp()          ;Returns TP in A4; Can be CSEed
LDW  *+DP($GOT_TPR_byte(x)), A5   ;reloc R_C6000_SBR_GOT_U15_W_TPR_B
LDB    *A4[A5], B4                ;

LDW  *+DP($GOT_TPR_hword(x)), A5  ;reloc R_C6000_SBR_GOT_U15_W_TPR_H
LDH    *A4[A5], B4                ;

LDW  *+DP($GOT_TPR_word(x)), A5   ;reloc R_C6000_SBR_GOT_U15_W_TPR_W
LDW    *A4[A5], B4                ;

LDW  *+DP($GOT_TPR_dword(x)), A5  ;reloc R_C6000_SBR_GOT_U15_W_TPR_D
LDDW   *A4[A5], B4                ;
```

The relocation R_C6000_SBR_GOT_U15_W_TPR_[B|H|W] causes the linker to create a GOT entry for x's TPR offset:

```
GOT[m]          ;reloc R_C6000_TPR_U32_B  (symbol x)
GOT[n]          ;reloc R_C6000_TPR_U32_H  (symbol y)
GOT[o]          ;reloc R_C6000_TPR_U32_W  (symbol z)
GOT[p]          ;reloc R_C6000_TPR_U32_D  (symbol z)
```

The _TPR_U32_[B|H|W|DW] relocations are resolved by the dynamic loader with the offset of x from the thread pointer. These relocations are scaled as per the access width.

If the GOT must be accessed using far-DP addressing, the sequence is as follows:

```
callp  __c6xabi_get_tp()          ;Returns TP in A4; Can be CSEed
MVKL  $DPR_GOT_TPR_byte(x), A5    ;reloc R_C6000_SBR_GOT_L16_W_TPR_B
MVKH  $DPR_GOT_TPR_byte(x), A5    ;reloc R_C6000_SBR_GOT_H16_W_TPR_B
ADD   DP, A5, A5
LDW   *A5, A5
LDB   *A4[A5], A6

MVKL  $DPR_GOT_TPR_hword(x), A5   ;reloc R_C6000_SBR_GOT_L16_W_TPR_H
MVKH  $DPR_GOT_TPR_hword(x), A5   ;reloc R_C6000_SBR_GOT_H16_W_TPR_H
ADD   DP, A5, A5
LDW   *A5, A5
LDH   *A4[A5], A6

MVKL  $DPR_GOT_TPR_word(x), A5    ;reloc R_C6000_SBR_GOT_L16_W_TPR_W
MVKH  $DPR_GOT_TPR_word(x), A5    ;reloc R_C6000_SBR_GOT_H16_W_TPR_W
```

```
        ADD   DP, A5, A5
        LDW   *A5, A5
        LDW   *A4[A5], A6

        MVKL  $DPR_GOT_TPR_dword(x), A5    ;reloc R_C6000_SBR_GOT_L16_W_TPR_D
        MVKH  $DPR_GOT_TPR_dword(x), A5    ;reloc R_C6000_SBR_GOT_H16_W_TPR_D
        ADD   DP, A5, A5
        LDW   *A5, A5
        LDDW  *A4[A5], A6
```

7.4.1.4 Local Exec TLS Access Model

This is an optimization of the Initial Exec model. When the program's initial TLS image (normally called the static TLS image) is created, the TLS block is always placed at a known offset from the thread pointer. Normally this is the Thread Control Block (TCB) plus the TLS Block Base offset. Hence the executable's own thread-local variable has a thread pointer relative offset that is a static link time constant. In this case, thread-local variables can be accessed using an inline constant offset; a GOT entry is not needed. Objects using this access model cannot be used to build a dynamic library.

```
        CALLP  __c6xabi_get_tp()   ; Returns TP in A4. Can be CSEed.
        MVK    $TPR_byte(x), A5     ; reloc R_C6000_TPR_U15_B
        LDB    *A4[A5], A4          ; A4 contains the value of thread-local char x

        MVK    $TPR_hword(y), A5    ; reloc R_C6000_TPR_U15_H
        LDH    *A4[A5], A4          ; A4 contains the value of thread-local short y

        MVK    $TPR_word(z), A5     ; reloc R_C6000_TPR_U15_W
        LDW    *A4[A5], A4          ; A4 contains the value of thread-local int z

        MVK    $TPR_dword(l), A5    ; reloc R_C6000_TPR_U15_D
        LDDW   *A4[A5], A7:A6       ; A7:A6 contains the value of thread-local long long l
```

The TPR_U15 relocations encode 15-bit unsigned TPR offsets (offset from the address to which the TP points) for near TPR addressing. They are scaled according to the access width. The previous addressing can access a TLS block of size 32 KB. This specification limits the size of the total static TLS to 32 KB, because this limit is expected to be sufficient for most use cases. Hence the far TPR address is not defined. Far TBR addressing may be defined, but doing so would use up 8 new relocations, and it is better to conserve the limited number of relocations (256) ELF allows.

7.4.2 Static Executable TLS Model

The static executable TLS model can be supported by a C6x EABI conforming compiler as a Quality of Implementation (QoI) item. It is not required for C6x EABI compliance.

In the case of a static executable, there is only one TLS block, and the TLS offset of each thread-local variable is known at static link time. The access to thread-local variables is *(TLS base + offset). Figure 7 shows the run-time layout of the TLS. TP is the thread pointer that points to the current thread's TLS block. The offset of x is known during static linking.
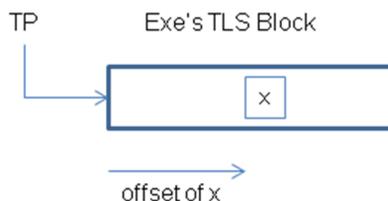
Figure 7. Static Executable TLS Run-Time Representation

7.4.2.1 Static Executable Addressing

The thread-local access code in the case of the static executable TLS model is the same as for the Linux Local Exec model (Section 7.4.1.4). In the case of a static executable, there is no Thread Control Block (TCB), so the TPR offset is the same as the TLS Block Base relative address.

Ideally we could generate TBR addressing for this case. However, the compiler options can be used to build using the bare-metal dynamic linking model, which requires a TCB. So, we generate TPR addressing for the static executable model as follows:

```
CALLP  __c6xabi_get_tp()  ; Returns TP in A4. Can be CSEed.
MVK    $TPR_byte(x), B4    ; reloc R_C6000_TPR_U15_B
LDB    *A4[B4], A4         ; A4 contains the value of thread-local char x

MVK    $TPR_hword(y), B4   ; reloc R_C6000_TPR_U15_H
LDH    *A4[B4], A4         ; A4 contains the value of thread-local short y

MVK    $TPR_word(z), B4    ; reloc R_C6000_TPR_U15_W
LDW    *A4[B4], A4         ; A4 contains the value of thread-local int z

MVK    $TPR_dword(l), B4   ; reloc R_C6000_TPR_U15_D
LDDW   *A4[B4], A7:A6      ; A4 contains the value of thread-local int l
```

The TPR relocations are resolved by the static linker with the offset of the variable in the executable's TLS block. The far TPR relocations can be used if the TLS block is expected to be bigger than 32 KB.

7.4.2.2 Static Executable TLS Runtime Architecture

In dynamic linking systems, the dynamic loader creates the main thread and the thread library creates additional threads. As part of the main thread creation, the dynamic loader allocates and initializes the main thread's TLS. Also, the dynamic loader can easily find the TLS initialization image using the segment type.

In the case of a static executable, there is no dynamic loader to perform these roles. The static linking model should support the following requirements:

- The allocation and initialization of the main thread's TLS before main() or any user code from init_array is called.
- During the main thread's execution, __c6xabi_get_tp() should return the pointer to main thread's TLS. This function must be supported even when there is no thread library.
- The thread library should have a way to access the TLS initialization image so that it can initialize the TLS blocks for the threads it creates.

Section 7.4.2.3.1 through Section 7.4.2.5 contain information that is toolchain-specific. Mentions of the .TI.tls_init and .TI.tls sections, the __TI_tls_init_table copy table, the __TI_TLS_MAIN_THREAD_BASE and __TI_TLS_BLOCK_SIZE symbols, and the __TI_tls_init() function are included as examples of how a toolchain can implement the TLS model.

7.4.2.3 Static Executable TLS Allocation

Three memory areas need to be allocated in order to support TLS in the static executable model: the initialization image, the main thread's TLS block, and the TLS area where the thread-library can allocate TLS blocks for the threads it creates.

7.4.2.3.1 TLS Initialization Image Allocation

The TLS initialization image is created in the output section, .TI.tls_init. This section is read-only. The user can specify the allocation for this output section as follows:

```
.TI.tls_init > ROM
```

If no allocation is specified, this output section is allocated using .cinit allocation. If no allocation specified for .cinit the default allocation is used. The user cannot specify the section specifier for this section.

The .TI.tls_init output segment is formed by combining the following linker created components:

- **.tdata.load** — Compressed TLS initialized section
- **.tbss.load** — Zero-init section to zero initialize uninitialized section
- **__TI_tls_init_table** — Copy table to initialize TLS blocks. This copy table has two copy records, one for each of these initialization sections.

7.4.2.3.2   Main Thread's TLS Allocation

Users can specify the allocation for the main thread's TLS block using:

```
.TI.tls > RAM
```

This uninitialized output section is initialized using the __TI_tls_init_table copy table at boot time. Users cannot specify the section specifier for this section.

If no allocation is specified for this section, it is allocated using the .fardata output section's allocation. If no allocation is specified for .fardata, the .far allocation is used. Otherwise, the default allocation is used.

The linker defines the symbol __TI_TLS_MAIN_THREAD_BASE to point to the start of the .TI.tls output section.

7.4.2.3.3   Thread Library's TLS Region Allocation

Allocating the TLS region to be used by the thread-library is specific to the library. The specification does not dictate a specific way to do this. One possible way to allocate the TLS region is as follows:

```
.tls_region { . += 0x2000; } START(TLS_REGION_START) > RAM
```

The thread library can use the symbol TLS_REGION_START to locate the TLS region. A user might want to allocate TLS blocks for N number of threads, and it is useful to know the size of TLS block. The user can do the following:

```
.tls_region { . += MAX_THREADS * __TI_TLS_BLOCK_SIZE; } > RAM
```

The static linker defines the symbol __TI_TLS_BLOCK_SIZE and sets it to the size of the TLS block.

7.4.2.4   Static Executable TLS Initialization

Two memory areas need to be initialized to support TLS in the static executable model: the main thread's TLS block, and the TLS area where the thread-library can allocate TLS blocks for the threads it creates.

7.4.2.4.1   Main Thread's TLS Initialization

During boot up, the startup code calls the run-time support (RTS) function __TI_tls_init(NULL) to initialize the main thread's TLS block. The RTS function initializes the main thread's TLS if a NULL argument is passed.

7.4.2.4.2   TLS Initialization by Thread Library

The thread library must initialize the TLS blocks once it creates them for a given thread. The static executable TLS model defines a new RTS function for this:

```
__TI_tls_init(void * dest_addr);
```

The thread library must pass the address of the TLS block to be initialized to this function.

This RTS function uses the copy table to perform the initialization. However, how this function initializes the TLS block is based on the interface between the static linker and this RTS function, which is subject to future changes. Therefore, the thread library must use only this RTS function as the interface to initialize TLS blocks.

7.4.2.5   Thread Pointer

In the Static Executable TLS model, the function __c6xabi_get_tp() is called to get the thread pointer value of the current thread. If a thread library is used, it is responsible for providing this function.

The thread library knows the address of the TLS block for the threads it creates. However, the main thread is not created by the thread library, so the thread library needs a standard way to find the address of the main thread's TLS block. As mentioned previously, the static linker defines the symbol __TI_TLS MAIN_THREAD_BASE for this purpose.

The TI RTS provides the following definition for the __c6xabi_get_tp() function:

```
extern __attribute__((weak)) far const void * __TI_TLS_MAIN_THREAD_Base;
__attribute__((weak)) void * __c6xabi_get_tp(void)
{
    return &__TI_TLS_MAIN_THREAD_Base;
}
```

This function is defined as "weak" so that a strong definition from the thread library will be used if one is present.

Let us consider the unlikely case in which a user declares thread-local variables but does not include the thread library. Obviously they cannot create any new threads. But the main thread should work and the main thread's thread-local variables should be accessible. In such cases, the previously mentioned RTS function is linked in and provides access to the main thread's TLS.

7.4.3   Bare-Metal Dynamic Linking TLS Model

Bare-metal dynamic linking involves only modules loaded initially. Modules that are dlopened are not currently supported with bare-metal dynamic linking. Objects compiled for a static executable can be used to create a bare-metal dynamic executable or library.

7.4.3.1   Default TLS Addressing for Bare-Metal Dynamic Linking

The default code generation for TLS should work for both static executables and bare-metal dynamic linking. For static executables, generate the following addressing using TPR addressing:

```
CALLP   __c6xabi_get_tp()   ; Returns TP in A4. Can be CSEed.
MVK     $TPR_byte(x), B4     ; reloc R_C6000_TPR_U15_B
LDB     *A4[B4], A4          ; A4 contains the value of thread-local char x
```

The code generated by default for bare-metal dynamic linking can assume that all modules are initially loaded. This means the offset of thread-local variables is a dynamic link time constant as shown in Figure 8. Hence the TPR addressing can be used. The only difference is that in the bare-metal dynamic linking case, a 64-bit TCB is needed to make the code compatible with any future support for dlopen(). In the case of static executables, the TCB is not present. Still the TPR addressing can be used for both models. The static linker will use a TCB size of zero for a static executable and a 64-bit TCB size for bare-metal dynamic linking.

Figure 8. Bare-Metal Default TLS Run-Time Representation

As mentioned earlier, the initially loaded modules are placed consecutively, and the executable's TLS block comes after the TCB. In this case, the variables in the executable can be accessed using static link time constant offsets from the TP. The variables defined in the dynamic libraries can be accessed using dynamic link time constant offsets from the TP.

When this addressing is generated the modules are marked DF_STATIC_TLS.

When building a dynamic executable, the static linker resolves the TPR relocations for symbols defined in the executable (own data) to the TP offset. If the symbol is imported, the relocation is copied to the dynamic relocation table to be resolved by the dynamic loader. When building a dynamic library, the TPR relocations are copied to the dynamic relocation table.

The thread-local access in bare-metal can result in dynamic relocations in a code segment. This means the resulting module is not truly PIC (position independent code). The TI compiler supports bare-metal PIC with the --gen_pic option. When this option is used, TPR offsets should be accessed from GOT entry to generate position independent code.

7.4.3.2   TLS Block Creation

In the case of bare-metal dynamic linking systems, the dynamic loader is responsible for creating the main thread's TLS block. The dynamic loader when loading an ELF File should load the PT_TLS segment and should provide a way for the thread library to access the PT_TLS Initialization Image so the thread library can use it to initialize the TLS blocks for the threads it creates. The static linker when building a dynamic executable/library generates the PT_TLS segment as per ELF requirements.

Each dynamic module (executable or shared object or dynamic library) gets its own TLS block. The PT_TLS segment contains the initial values for the TLS objects that are defined in a given module.

7.5   Thread-Local Symbol Resolution and Weak References

A thread-local reference can only be resolved by a thread-local definition. The linker should enforce this requirement. Also, the presence of a thread-local definition and a normal global definition with the same name is an error.

Thread-local variables can be defined or declared weak. A weak thread-local definition implies that it can be overridden by a strong definition if available. If a strong definition is not found, the weak definition is used. No special care is needed to support thread-local weak definitions.

A weak thread-local symbol reference is resolved to zero address if a definition is not found. This requires special handling in each of the TLS addressing models.

7.5.1   General and Local Dynamic TLS Weak Reference Addressing

In both the General and Local Dynamic TLS models, the function __tls_get_addr() is called to get the thread-local variable's address. The module-id in both the General and Local Dynamic TLS models is obtained from the GOT. The offset is obtained from the GOT in General Dynamic model and as a static link-time constant in Local Dynamic model. In the case of weak undefined reference, there is no thread-local definition to resolve the weak reference. Since there is no definition, the module-id and TBR offset resolve to zero.

For weak thread-local references, there is no change in the code generated to access the references. The R_C6000_TLSMOD relocation and all the R_C6000_TBR relocations resolve to zero if the thread-local reference is weak and there is no definition.

The __tls_get_addr() function returns zero when the module-id and offset are zero. This ensures that an undefined weak reference address is resolved to zero.

7.5.2   Initial and Local Executable TLS Weak Reference Addressing

Thread-pointer relative addressing cannot be used for weak references, since there is no way to generate a zero address if the symbol is undefined. Therefore, the Initial Executable access model must use General Dynamic addressing for weak references. Similarly, the Local Executable access model must use Local Dynamic addressing for weak references.

7.5.3   Static Exec and Bare Metal Dynamic TLS Model Weak References

Thread-pointer relative addressing cannot be used for weak references, since there is no way to generate a zero address if the symbol is undefined. Therefore, the Local Dynamic form must be used for weak references in the Static Executable and Bare-Metal Dynamic Linking access model.

In static and bare-metal dynamic linking the following addressing is generated for weak references:

```
MVK   $TPR_S16(x), A5              ; reloc R_C6000_TPR_S16
   ||   CALLP   __c6xabi_get_addr,B3 ; A4 has the address of x at return
```

The C6x eabi function __c6xabi_get_addr() has the following signature:

```
void * __c6xabi_get_addr(ptrdiff_t TPR_offst);
```

This function accepts a 32-bit TPR offset and returns the address of the thread-local variable. A special value of -1 for the TPR offset is used to indicate a weak undefined reference. A zero is returned in this case.

The static linker and dynamic linker resolve TPR_S16 relocations to -1 for a weak undefined reference.

8   Helper Function API

To enable object files built with one toolchain to be linked with a run-time 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 section. 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 sections.

8.1   *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 C6000 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, the ABI specifies them 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).

8.2   *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 _ _c6xabi_. Any identifier with this prefix is reserved for the ABI. In addition, the _ _tls_get_addr() helper function is needed to support dynamic linking access to thread-local storage.

The helper functions adhere to the standard calling conventions, except as indicated in Section 8.3.

The following tables specify the helper functions using C notation and syntax. The types in the table correspond to the generic data types specified in Section 2.1.

The functions in Table 8 convert floating-point values to integer values, in accordance with C's conversion rules and the floating-point behavior specified by Section 8.1.

Table 8. C6000 Floating Point to Integer Conversions

Signature	Description
int32 _ _**c6xabi_fixdi**(float64 x);	Convert float64 to int32
int40 _ _**c6xabi_fixdli**(float64 x);	Convert float64 to int40
int64 _ _**c6xabi_fixdlli**(float64 x);	Convert float64 to int64
uint32 _ _**c6xabi_fixdu**(float64 x);	Convert float64 to uint32
uint40 _ _**c6xabi_fixdul**(float64 x);	Convert float64 to uint40
uint64 _ _**c6xabi_fixdull**(float64 x);	Convert float64 to uint64
int32 _ _**c6xabi_fixfi**(float32 x);	Convert float32 to int32
int40 _ _**c6xabi_fixfli**(float32 x);	Convert float32 to int40
int64 _ _**c6xabi_fixflli**(float32 x);	Convert float32 to int64
uint32 _ _**c6xabi_fixfu**(float32 x);	Convert float32 to uint32
uint40 _ _**c6xabi_fixful**(float32 x);	Convert single-precision float to uint40
uint64 _ _**c6xabi_fixfull**(float32 x);	Convert single-precision float to uint64

The functions in Table 9 convert integer values to floating-point values, in accordance with C's conversion rules and the floating-point behavior specified by Section 8.1.

Table 9. C6000 Integer to Floating Point Conversions

Signature	Description
float64 _ _**c6xabi_fltid**(int32 x);	Convert int32 to double-precision float
float64 _ _**c6xabi_fltlid**(int40 x);	Convert int40 to double-precision float
float64 _ _**c6xabi_fltllid**(int64 x);	Convert int64 to double-precision float
float64 _ _**c6xabi_fltud**(uint32 x);	Convert uint32 to double-precision float
float64 _ _**c6xabi_fltuld**(uint40 x);	Convert uint40 to double-precision float
float64 _ _**c6xabi_fltulld**(uint64 x);	Convert uint64 to double-precision float
float32 _ _**c6xabi_fltif**(int32 x);	Convert int32 to single-precision float
float32 _ _**c6xabi_fltlif**(int40 x);	Convert int40 to single-precision float
float32 _ _**c6xabi_fltllif**(int64 x);	Convert int64 to single-precision float
float32 _ _**c6xabi_fltuf**(uint32 x);	Convert uint32 to single-precision float
float32 _ _**c6xabi_fltulf**(uint40 x);	Convert uint40 to single-precision float
float32 _ _**c6xabi_fltullf**(uint64 x);	Convert uint64 to single-precision float

The functions in Table 10 convert floating-point values from one format to another, in accordance with C's conversion rules and the floating-point behavior specified by Section 8.1.

Table 10. C6000 Floating-Point Format Conversions

Signature	Description
float32 _ _**c6xabi_cvtdf**(float64 x);	Convert double-precision float to single-precision
float64 _ _**c6xabi_cvtfd**(float32 x);	Convert single-precision float to double-precision

The functions in Table 11 perform floating-point arithmetic, in accordance with C semantics and the floating-point behavior specified by Section 8.1.

Table 11. C6000 Floating-Point Arithmetic

Signature	Description
float64 _ _**c6xabi_absd**(float64 x);	Return absolute value of double-precision float
float32 _ _**c6xabi_absf**(float32 x);	Return absolute value of single-precision float
float64 _ _**c6xabi_addd**(float64 x, float64 y);	Add two double-precision floats (x+y)
float32 _ _**c6xabi_addf**(float32 x, float32 y);	Add two single-precision floats (x+y)
float64 _ _**c6xabi_divd**(float64 x, float64 y);	Divide two double-precision floats (x/y)
float32 _ _**c6xabi_divf**(float32 x, float32 y);	Divide two single-precision floats (x/y)
float64 _ _**c6xabi_mpyd**(float64 x, float64 y);	Multiply two double-precision floats (x*y)
float32 _ _**c6xabi_mpyf**(float32 x, float32 y);	Multiply two single-precision floats (x*y)
float64 _ _**c6xabi_negd**(float64 x);	Return negated double-precision float (-x)
float32 _ _**c6xabi_negf**(float32 x);	Return negated single-precision float (-x)
float64 _ _**c6xabi_subd**(float64 x, float64 y);	Subtract two double-precision floats (x-y)
float32 _ _**c6xabi_subf**(float32 x, float32 y);	Subtract two single-precision floats (x-y)
int64 _ _**c6xabi_trunc**(float64 x);	Truncate double-precision float toward zero
int32 _ _**c6xabi_truncf**(float32 x);	Truncate single-precision float toward zero

The functions in Table 12 perform floating-point comparisons in accordance with C semantics and the floating-point behavior specified by Section 8.1.

The _ _c6xabi_cmp* functions return an integer less than 0 if x is less than y, 0 if the values are equal, or an integer greater than 0 of x is greater than y. If either operand is NaN, the result is undefined.

The explicit comparison functions operate correctly with unordered (NaN) operands. That is, they return non-zero if the comparison is true even if one of the operands is NaN, or 0 otherwise.

Table 12. Floating-Point Comparisons

Signature	Description
int32 _ _**c6xabi_cmpd**(float64 x, float64 y);	Double-precision comparison
int32 _ _**c6xabi_cmpf**(float32 x, float32 y);	Single-precision comparison
int32 _ _**c6xabi_unordd**(float64 x, float64 y);	Double-precision check for unordered operands
int32 _ _**c6xabi_unordf**(float32 x, float32 y);	Single-precision check for unordered operands
int32 _ _**c6xabi_eqd**(float64 x, float64 y);	Double-precision comparison: x == y
int32 _ _**c6xabi_eqf**(float32 x, float32 y);	Single-precision comparison: x == y
int32 _ _**c6xabi_neqd**(float64 x, float64 y);	Double-precision comparison: x != y
int32 _ _**c6xabi_neqf**(float32 x, float32 y);	Single-precision comparison: x != y
int32 _ _**c6xabi_ltd**(float64 x, float64 y);	Double-precision comparison: x < y
int32 _ _**c6xabi_ltf**(float32 x, float32 y);	Single-precision comparison: x < y
int32 _ _**c6xabi_gtd**(float64 x, float64 y);	Double-precision comparison: x > y
int32 _ _**c6xabi_gtf**(float32 x, float32 y);	Single-precision comparison: x > y
int32 _ _**c6xabi_led**(float64 x, float64 y);	Double-precision comparison: x <= y
int32 _ _**c6xabi_lef**(float32 x, float32 y);	Single-precision comparison: x <= y
int32 _ _**c6xabi_ged**(float64 x, float64 y);	Double-precision comparison: x >= y
int32 _ _**c6xabi_gef**(float32 x, float32 y);	Single-precision comparison: x >= y

The integer divide and remainder functions in Table 13 operate according to C semantics.

The _ _c6xabi_divremi and _ _c6xabi_divremu functions compute both a quotient (x/y) and remainder (x%y). The quotient is returned in A4 and the remainder in A5.

The _ _c6xabi_divremll and _ _c6xabi_divremull function computes the quotient (x/y) and remainder (x%y) of 64-bit integers. The quotient is returned in A5:A4 and the remainder in B5:B4.

Table 13. C6000 Integer Divide and Remainder

Signature	Description
int32 _ _**c6xabi_divi**(int32 x, int32 y);	32-bit signed integer division (x/y)
int40 _ _**c6xabi_divli**(int40 x, int40 y);	40-bit signed integer division (x/y)
int64 _ _**c6xabi_divlli**(int64 x, int64 y);	64-bit signed integer division (x/y)
uint32 _ _**c6xabi_divu**(uint32 x, uint32 y);	32-bit unsigned integer division (x/y)
uint40 _ _**c6xabi_divlu**(uint40 x, uint40 y);	40-bit unsigned integer division (x/y)
uint64 _ _**c6xabi_divllu**(uint64 x, uint64 y);	64-bit unsigned integer division (x/y)
int32 _ _**c6xabi_remi**(int32 x, int32 y);	32-bit signed integer modulo (x%y)
int40 _ _**c6xabi_remli**(int40 x, int40 y);	40-bit signed integer modulo (x%y)
int64 _ _**c6xabi_remlli**(int64x. int64 y);	64-bit signed integer modulo (x%y)
uint32 _ _**c6xabi_remu**(uint32 x, uint32 y);	32-bit unsigned integer modulo (x%y)
uint40 _ _**c6xabi_remul**(uint40, uint40);	40-bit unsigned integer modulo (x%y)
uint64 _ _**c6xabi_remull**(uint64, uint64);	64-bit unsigned integer modulo (x%y)
_ _**c6xabi_divremi**(int32 x, int32 y);	32-bit combined divide and modulo
_ _**c6xabi_divremu**(uint32 x, uint32 y);	32-bit unsigned combined divide and modulo
_ _**c6xabi_divremull**(uint64 x, uint64 y);	64-bit unsigned combined divide and modulo

The wide integer arithmetic functions in Table 14 operate according to C semantics.

Table 14. C6000 Wide Integer Arithmetic

Signature	Description
int64 _ _**c6xabi_negll**(int64 x);	64-bit integer negate
uint64 _ _**c6xabi_mpyll**(uint64 x, uint64 y);	64x64 bit multiply
int64 _ _**c6xabi_mpyiill**(int32 x, int32 y);	32x32 bit multiply
uint64 _ _**c6xabi_mpyuiill**(uint32 x, uint32 y);	32x32 bit unsigned multiply
int64 _ _**c6xabi_llshr**(int64 x, uint32 y);	64-bit signed right shift (x>>y)
uint64 _ _**c6xabi_llshru**(uint64 x, uint32 y);	64-bit unsigned right shift (x>>y)
uint64 _ _**c6xabi_llshl**(uint64 x, uint32 y);	64-bit left shift (x<<y)

The miscellaneous helper functions in Table 15 are described in the sections that follow.

Table 15. C6000 Miscellaneous Helper Functions

Signature	Description
void _ _**c6xabi_strasgi**(int32 *dst, const int32 *src, uint32 cnt);	Interrupt safe block copy; cnt >= 28
void _ _**c6xabi_strasgi_64plus**(int32*, const inst32*, uint32) ;	Interrupt safe block copy; cnt >= 28
void _ _**c6xabi_abort_msg**(const char *string);	Report failed assertion
void _ _**c6xabi_push_rts**(void);	Push all callee-saved registers
void _ _**c6xabi_pop_rts**(void);	Pop all callee-saved registers
void _ _**c6xabi_call_stub**(void);	Save caller-save registers; call B31
void _ _**c6xabi_weak_return**(void);	Resolution target for imported weak calls
void _ _**c6xabi_get_addr**(ptrdiff_t TPR_offst);	Get the address of the thread-pointer register (TPR) offset.
void _ _**c6xabi_get_tp**(void);	Get the thread pointer value of the current thread.
void * _ _**tls_get_addr**(struct TLS_descriptor);	Get the address of a thread-local variable.

_ _c6xabi_strasgi

The function _ _c6xabi_strasgi 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 source and destination objects do not overlap.

The 7-word minimum is the threshold that allows a software-pipelined loop to be used on C64x+. For smaller objects, the compiler typically generates an inline sequence of load/store instructions. _ _c6xabi_strasgi does not disable interrupts and can be safely interrupted.

The function _ _c6xabi_strasgi_64plus is a version of _ _c6xabi_strasgi optimized for C64x+ architectures.

_ _c6xabi_abort_msg

The function _ _c6xabi_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.

_ _c6xabi_push_rts and _ _c6xabi_pop_rts

The function _ _c6x_push_rts is used on C64x+ architectures when optimizing for code size. Many functions save and restore most or all of the callee-saved registers. To avoid duplicating the save code in the prolog and restore code in the epilog of each such function, the compiler can employ this library function instead. The function pushes all 13 callee-saved registers on the stack, decrementing SP by 56 bytes, according to the protocol in Section 4.4.4.

The function _ _c6x_push_rts is implemented as shown:

```
__c6xabi_push_rts:
        STW      B14, *B15--[2]
        STDW   A15:A14, *B15--
        STDW   B13:B12, *B15--
        STDW   A13:A12, *B15--
        STDW   B11:B10, *B15--
        STDW   A11:A10, *B15--
        STDW   B3:B2,   *B15--
        B     A3
```

(This is a serial, unscheduled representation. Refer to the source code in the TI run-time library for the actual implementation.)

The function _ _c6xabi_pop_rts restores the callee-saved registers as pushed by _ _c6xabi_push_rts and increments (pops) the stack by 56 bytes.

_ _c6xabi_call_stub

The function _ _c6xabi_call_stub is also used to help optimize c64x+ functions for code size. Many call sites have several caller-save registers that are live across the call. These registers are not preserved by the call and therefore must be saved and restored by the caller. The compiler can route the call through _ _c6xabi_call_stub, which performs the following sequence of operations:

- Save selected caller-save registers on the stack

- Call the function

- Restore the saved registers

- Return

In this way the selected registers are preserved across the call without the caller having to save and restore them. The registers preserved by _ _c6xabi_call_stub are: A0, A1, A2, A6, A7, B0, B1, B2, B4, B5, B6, B7.

The caller invokes _ _c6xabi_call_stub by placing the address of the function to be called in B31, then branching to _ _c6xabi_call_stub. (The return address is in B3 as usual.)

The function _ _c6xabi_call_stub is implemented as shown:

```
__c6xabi_call_stub:
        STW    A2, *B15--[2]
        STDW   A7:A6, *B15--
```

```
            STDW    A1:A0, *B15--
            STDW    B7:B6, *B15--
            STDW    B5:B4, *B15--
            STDW    B1:B0, *B15--
            STDW    B3:B2, *B15--
            ADDKPC __STUB_RET, B3, 0
            CALL    B31
      __STUB_RET:
            LDDW    *++B15, B3:B2
            LDDW    *++B15, B1:B0
            LDDW    *++B15, B5:B4
            LDDW    *++B15, B7:B6
            LDDW    *++B15, A1:A0
            LDDW    *++B15, A7:A6
            LDW     *++B15[2], A2
            B       B3
```

(This is a serial, unscheduled representation. Refer to the source code in the TI run-time library for the actual implementation.)

Since _ _c6xabi_call_stub uses non-standard conventions, it cannot be called via a PLT entry. Its definition in the library must be marked as STV_INTERNAL or STV_HIDDEN to prevent it from being importable from a shared library.

_ _c6xabi_weak_return

The function _ _c6xabi_weak_return is a function that simply returns. The linker shall include it in a dynamic executable or shared object that contains any unresolved calls to imported weak symbols. The dynamic linker can use it to resolve those calls if they remain unresolved at dynamic load time.

_ _c6xabi_get_addr

The function _ _c6xabi_get_addr accepts 32-bit TPR offset and returns the address of the thread-local. A special value of -1 is used to indicate a weak undefined reference and a zero is returned in this case. This function is used when compiling for the Static Executable and Bare Metal Dynamic TLS access models. See Section 7 for details about thread-local storage.

_ _c6xabi_get_tp

The function _ _c6xabi_get_tp returns the thread pointer value for the current thread. This function does not modify any register other than the return register A4. This function can be called via PLT and hence the caller should assume B30 and B31 are modified by the call to this function. See Section 7 and Section 14.1.4 for details about thread-local storage.

_ _tls_get_addr

The function _ _tls_get_addr returns the address of a thread-local variable. See Section 7.4.1.1 for details about this function and the TLS_descriptor structure passed to it to specify the offset of a thread-local variable. This function is used when compiling for all access models other than the Static Executable and Bare Metal Dynamic TLS access models. See Section 7 for details about thread-local storage.

8.3 Special Register Conventions for Helper Functions

The helper functions adhere to the standard calling conventions, except as specifically noted previously. 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.

Helper functions with special register conventions cannot be called via PLT entries (see Section 6.5). Consequently, their definitions must be marked STV_INTERNAL or STV_HIDDEN to prevent them from being importable from a shared library.

Table 16 lists those helper functions that have modified register save conventions. If a function is listed in the table, the given registers are the only registers modified by a call to that function. If a function is not listed, it follows the standard rules.

Table 16. C6000 Register Conventions for Helper Functions

Function	Registers Modified
_ _c6xabi_divi	A0,A1,A2,A4,A6,B0,B1,B2,B4,B5,B30,B31
_ _c6xabi_divu	A0,A1,A2,A4,A6,B0,B1,B2,B4,B30,B31
_ _c6xabi_remi	A1,A2,A4,A5,A6,B0,B1,B2,B4,B30,B31
_ _c6xabi_remu	A1,A4,A5,A7,B0,B1,B2,B4,B30,B31
_ _c6xabi_divremi	A1,A2,A4,A5,A6,B0,B1,B2,B4,B30,B31
_ _c6xabi_divremu	A0,A1,A2,A4,A6,B0,B1,B2,B4,B30,B31
_ _c6xabi_strasgi_64plus	A31,A30,B31,B30,ILC,RILC,B30,B31
_ _c6xabi_push_rts	A15,A3,B3,B30,B31
_ _c6xabi_pop_rts	B10,B11,B12,B13,B14,B30,B31
_ _c6xabi_call_stub	A3-A5,A8,A9,A16-A31,B8,B9,B16-B31,ILC,RILC

B30 and B31 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 for trampolines. See Section 3.7.

8.4   *Helper Functions for Complex Types*

These functions support multiplication and division on complex types. The behavior is as specified by annex G of the C99 standard.

Table 17. Helper Functions for Complex Types

Signature	Description
float64 complex _ _**c6xabi_mpycd**(float64 complex x, float64 complex y);	Double-precision complex multiply
float32 complex _ _**c6xabi_mpycf**(float32 complex x, float32 complex y);	Single-precision complex multiply
float64 complex _ _**c6xabi_divcd**(float64 complex x, float64 complex y);	Double-precision complex divide (x/y)
float32 complex _ _**c6xabi_divcf**(float32 complex x, float32 complex y);	Single-precision complex divide (x/y)

8.5   *Floating-Point Helper Functions for C99*

These functions are unimplemented, but the names are reserved for use by a C99 compiler. The TI library does not currently implement these functions. The API relating to C99 is subject to change.

Table 18. Reserved Floating-Point Classification Helper Functions

Signature	Description
int32 _ _**c6xabi_isfinite**(float64 x);	True iff x is a representable value
int32 _ _**c6xabi_isfinitef**(float32 x);	True iff x is a representable value
int32 _ _**c6xabi_isinf**(float64 x);	True iff x represents "infinity"
int32 _ _**c6xabi_isinff**(float32 x);	True iff x represents "infinity"
int32 _ _**c6xabi_isnan**(float64 x);	True iff x represents "not a number"
int32 _ _**c6xabi_isnanf**(float32 x);	True iff x represents "not a number"
int32 _ _**c6xabi_isnormal**(float64 x);	True iff x is not denormalized
int32 _ _**c6xabi_isnormalf**(float32 x);	True iff x is not denormalized
int32 _ _**c6xabi_fpclassify**(float64 x);	Classify floating-point value
int32 _ _**c6xabi_fpclassifyf**(float32 x);	Classify floating-point value

The function _ _**c6xabi_fpclassify** is for use in classifying floating-point numbers. The operation is:

```
int32 __c6xabi_fpclassify(float64 x)
{
   if      (isnormal(x)) return 3;
   else if (isinf(x))    return 1;
   else if (isnan(x))    return 2;
   else                  return 4;
}
```

The following C99 functions are perform rounding and truncation toward zero.

Table 19. Reserved Floating-Point Rounding Functions

Signature	Description
float64 _ _**c6xabi_nround**(float64 x);	Round to nearest integer
float32 _ _**c6xabi_roundf**(float32 x);	Round to nearest integer
float64 _ _**c6xabi_trunc**(float64 x);	Truncate towards zero
float32 _ _**c6xabi_truncf**(float32 x);	Truncate towards zero

9   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). This section discusses header file and library compatibility.

Implementations that adhere to this ABI must conform to the C standard, and must produce object files that are compatible with those produced by another implementation.

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.

The C6000 is designed based on the ARM EABI. You can read the *C Library ABI for the ARM Architecture* document on the ARM Infocenter website for background and comments about how the standard C library should be implemented for EABI. The details that apply to ARM do not necessarily apply for C6000. See the chapter on "The C Library Section by Section" in that document.

9.1   Reserved Symbols

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

- _ftable
- _ctypes_

In addition, any symbols listed in Section 13.4.4 or symbols with the prefixes listed in Section 13.1 are reserved.

9.2   *<assert.h> Implementation*

The library must implement assert as a macro. If its expression argument is false, it must eventually call a helper function __c6xabi_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. See Table 15.

```
void __c6xabi_abort_msg(const char *);
```

9.3   *<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 C6000 toolset supports the C99 complex numbers and provides this header file.

9.4 *<ctype.h> Implementation*

The ctypes.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)

9.5 *<errno.h> Implementation*

errno is a macro that expands to an expression involving a function call as follows:

```
#define errno (*__c6xabi_errno_addr())
extern int *__c6xabi_errno_addr(void);
```

Note that this definition is affected by C6000 thread-local support. See Section 7.

The following constants are defined for used with errno:

```
#define EDOM   33
#define ERANGE 34
#define ENOENT 2
#define EFPOS  152
#define EILSEQ 88
```

9.6 *<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.

9.7 *<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. See Section 2.1.

9.8 *<iso646.h> Implementation*

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

9.9 *<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. See Section 2.1.

MB_LEN_MAX is defined as follows:

```
#define MB_LEN_MAX 1
```

9.10 *<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
```

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;
```

9.11 *<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 not currently specified.
- MATH_ERREXCEPT is not currently specified.

The following FP_* macros are defined:

```
#define FP_INFINITE  1
#define FP_NAN       2
#define FP_NORMAL    (-1)
#define FP_SUBNORMAL (-2)
#define FP_ZERO      0
```

The other FP_* macros are not currently specified.

9.12 <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 13 "int"s (that is, 32 bits * 13).

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.

9.13 <signal.h> Implementation

TI's toolset does not implement the signal library function.

TI's toolset creates the following typedef for "int".

```
typedef int sig_atomic_t;
```

TI's toolset defines the following constants:

```
#define SIG_DFL ((void (*)(int)) 0)
#define SIG_ERR ((void (*)(int)) -1)
#define SIG_IGN ((void (*)(int)) 1)

#define SIGABRT 6
#define SIGFPE  8
#define SIGILL  4
#define SIGINT  2
#define SIGSEGV 11
#define SIGTERM 15
```

9.14 <stdarg.h> Implementation

Only the type va_list shows up in the interface. Macros are used to implement va_start, va_arg, and va_end. See Section 3 for the format of the arguments in va_list.

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 3.3 and accessed using the macros in <stdarg.h>. The macros use a persistent argument pointer initialized via an invocation of va_start and advanced via invocations of va_arg. The following conventions apply to 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.

9.15 <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.

9.16 <stddef.h> Implementation

The types size_t and ptrdiff_t are defined in stddef.h. See Section 2.1.

9.17 <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. See Section 2.1.

9.18 *<stdio.h> Implementation*

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

```
#define _IOFBF 1
#define _IOLBF 2
#define _IONBF 4

#define BUFSIZ 256

#define EOF (-1)

#define FOPEN_MAX
#define FILENAME_MAX
#define TMP_MAX
#define L_tmpnam

#define SEEK_SET 0
#define SEEK_CUR 1
#define SEEK_END 2

#define stdin &_ftable[0]
#define stdout &_ftable[1]
#define stderr &_ftable[2]
```

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 C6000 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 an int.

9.19 *<stdlib.h> Implementation*

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

```
typedef struct { int quot; int rem; } div_t;
typedef struct { long int quot; long int rem; } ldiv_t;
typedef struct { long long int quot; long long int 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. See Section 7.

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. The TI toolset does not provide a system function.

9.20 *<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.

9.21 *<tgmath.h> Implementation*

The C99 standard completely specifies this header file. The TI toolset does not provide this header file.

9.22 *<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.

9.23 *<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.

9.24 *<wctype.h> Implementation*

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

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

10 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 C6000 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 as "GC++ABI", can be found at http://refspecs.linux-foundation.org/cxxabi-1.83.html.

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

10.1 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. For the C6000, the constraint is reduced to 24 bits. This implies a practical limit of 2^{23} -1 (or 0x7fffff) bytes on the size of a base class.

10.2 Export Template (GC++ABI 1.4.2)

Export templates are not currently specified by the ABI.

10.3 Data Layout (GC++ABI Chapter 2)

The layout of POD (Plain Old Data), is specified in Section 2 of this 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 2.7.

10.4 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;
}
```

10.5 Constructor Return Value (GC++ABI 3.1.5)

The C6000 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 specifies 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 C6000 ABI they are declared as pointers to functions that return void *, corresponding to 'this'.

10.6   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 10.4.

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

The C6000 ABI does not specify a mechanism to control object construction.

10.8   Demangler API (GC++ABI 3.4)

The C6000 ABI suspends the requirement for an implementation to provide the function
_ _cxa_demangle, which provides a run-time interface to the demangler.

10.9   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 C6000 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.

10.10   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 C6000 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 is first declared inline after the class definition.

10.11   Unwind Table Location (GC++ABI 5.3)

Exception handling is covered in Section 11 of this document.

11 Exception Handling

The C6000 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 section 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.

11.1 Overview

The C6000'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 section focuses on the C6000-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.

11.2 PREL31 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, this is done using a special-purpose PC-relative relocation called R_C6000_PREL31, abbreviated here as PREL31. A PREL31 field is encoded as a scaled, signed 31-bit offset which occupies the least significant 31 bits of a 32-bit word. The remaining (most significant) bit is used for different purposes in different contexts. The relocated address to which the field refers is found by left-shifting the encoded offset by 1 bit and adding it to the address of the field.

11.3 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 PREL31 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 a either a PREL31 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 detailed in the subsections that follow.

11.3.1 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-31-encoded address of the EXTAB entry for this address range in the other bits.

31	30-0
0	PREL31 Representation of function address
0	PREL31 Representation of EXTAB entry

11.3.2 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 calls abort or std::terminate, depending on the language.

31	30-0
0	PREL31 Representation of function address
0x00000001 (EXIDX_CANTUNWIND)	

11.3.3 Inlined EXTAB Entry

If the entire EXTAB entry for this function is small enough, it is placed in the second EXIDX word and the upper bit is set to one. The second word uses the same encoding as the EXTAB compact model described in Section 11.4, 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			PREL31 Representation of function address
1	**000**	PR Index	Data for personality routine specified by 'index'

11.4 *The Exception Handling Instruction Table (EXTAB)*

Each EXTAB entry is 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 20.

11.4.1 EXTAB Generic Model

A generic EXTAB entry is indicated by setting bit 31 of the first word to 0. The first word has a PREL31 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	30-0
0	PREL31 Representation of personality routine address
	Optional data for the personality routine
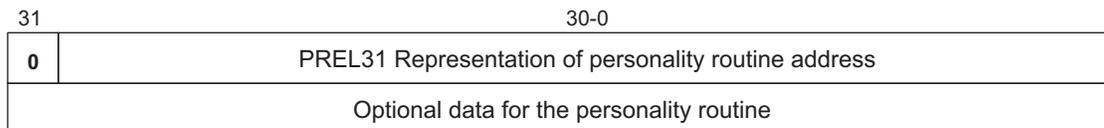
The format of the optional data is up to the discretion of the personality routine, but the length must be an integer multiple of whole 32-bit words. The unwinder calls the personality routine, passing it a pointer to the first word of optional data.

11.4.2 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)			
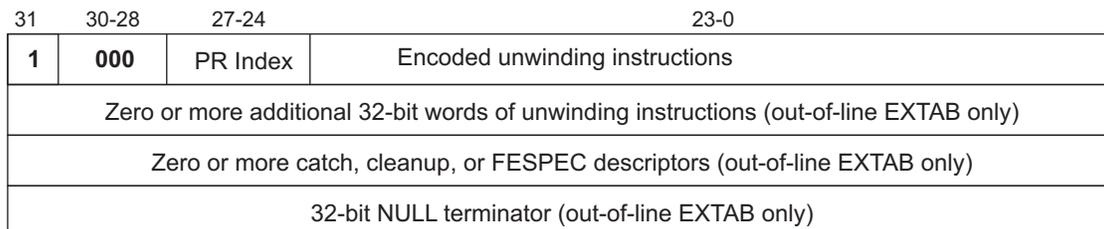
11.4.3 Personality Routines

The C6000 has the following ABI-specified personality routines. The first three have the same format as the ARM EABI. The following table specifies the personality routines and their PR indexes.

Table 20. C6000 TDEH Personality Routines

PR Index (bits 27-24)	Personality	Routine Name	Unwind Instructions	Width of Scope Fields	Notes
0000	PR0 (Su16)	_ _c6xabi_unwind_cpp_pr0	Up to 3 one-byte instructions	16	
0001	PR1 (Lu16)	_ _c6xabi_unwind_cpp_pr1	Unlimited onebyte instructions	16	
0010	PR2 (Lu32)	_ _c6xabi_unwind_cpp_pr2	Unlimited onebyte instructions	32	Must be used if 16-bit scope fields will not reach
0011	PR3	_ _c6xabi_unwind_cpp_pr3	24 bits	16	Optimized C6x-specific unwinding format
0100	PR4	_ _c6xabi_unwind_cpp_pr4	24 bits	16	Same as PR3, but the function epilog uses the alternate C64x+ *compact frame* layout.

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_C6000_NONE relocation.

11.5 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 decode the information and simulate the epilog.

The unwinding instructions make assumptions about the C6x stack layout; in particular, *callee-saved register safe debug order* is always assumed, except when the C64x+-specific _ _c6xabi_push_rts layout is used.

11.5.1 Common Sequence

Abstractly, all unwinding sequences take the following form:

1. Restore SP

    (a) If an FP was used, SP := FP

    (b) Otherwise, SP := SP + constant

2. (Optional) Restore B3 from a callee-saved register

3. (Optional) Restore callee-saved registers (reg1 := SP[0]; reg2 := SP[-1]; and so on)

4. Return through B3

Step 1: Restore SP

An actual epilog does not restore SP until after the callee-saved 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-saved registers.

SP will be restored by either copying from FP or incrementing by a constant. In the latter case, in addition to the explicit increment, the SP is implicitly incremented to account for the size of the callee-saved area. If SP is restored from FP, this additional increment is not implied.

Step 2: Restore B3

The return address must be in B3 before the return occurs. If it is stored in a callee-saved register (say "R"), then B3 needs to be restored from R before step 3 restores R itself.

Step 3: Restore Registers

Abstractly, the callee-saved registers are restored in *register safe debug* order (Section 4.4.2) starting with the location pointed to by (the old) SP and moving to lower addresses. TDEH forces the safe debug ordering except when using the c6xabi_push_rts layout (Section 4.4.4).

For stack frames created using the *compact frame* method (Section 4.4.4), there may be gaps between the saved registers due to the optimization favoring compressible instructions. The unwinder must be aware of the algorithm used to lay out the registers and adjust the register locations accordingly.

In big-endian mode, to facilitate the use of LDDW and STDW, if the two registers in a pair occupy the same aligned double word, the order of the pair is swapped. This is computed after the safe debug ordering is used to determine the layout, so some register pairs will not be swapped.

Generally the SP (B15) is not restored by the explicit register restores; it is explicitly restored for DATA_MEM_BANK layout (Section 4.4.3) when an FP is not available.

Step 4: Return

Every unwinding sequence ends with an implicit or explicit "RET B3", which indicates that unwinding is complete for the current frame.

11.5.2 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 B3" 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.

31	30-28	27-24	23-16	15-8	7-0
1	000	0000 (PR0)	First unwind instruction	Second unwind instruction	Third unwind instruction
Optional descriptors					
NULL					

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

31	30-28	27-24	23-16	15-8	7-0
1	000	PR Index	Number of additional unwinding words	First unwind instruction	Second unwind instruction
Third unwind instruction		Fourth unwind instruction		...	...
Optional descriptors					
NULL					

Table 21 summarizes the unwinding instruction set. Each instruction is described in more detail following the table.

Table 21. Stack Unwinding Instructions

Encoding	Instruction	Description
00kk kkkk	SP += (k << 3) + 8	Increment SP by a small constant
1101 0010 kkkk kkkk . . .	SP += (ULEB128 << 3) + 0x408	Increment SP by a ULEB128-encoded constant
1000 0000 0000 0000	CANTUNWIND	Function cannot be unwound, but might catch exceptions
100x xxxx xxxx xxxx	POP bitmask	POP one or more registers [x != 0]
101x xxxx xxxx xxxx	POP bitmask	POP one or more registers from a C64x+ compact frame [x != 0]
1100 nnnn xxxx xxxx . . .	POP register	n represents the number of registers to be popped, which are encoded in the following 4-bit nibbles
1101 0000	MV FP, SP	Restore SP from FP instead of incrementing SP
1101 0001	_ _c6xabi_pop_rts	Simulate a call to _ _c6xabi_pop_rts
1110 0111	RET B3	Unwinding complete for this frame
1110 xxxx	RETURN or restore B3	B3 := register x (x != B3)

All other bit patterns are reserved.

The following paragraphs detail the interpretation of the unwinding instructions.

Small Increment

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

The value of k is extracted from the lower 6 bits of the encoding. This instruction can increment the SP by a value in the range 0x8 to 0x200, inclusive. Increments in the range 0x208 to 0x400 should be done with two of these instructions.

Large Increment

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

The value ULEB128 is ULEB128-encoded in the bytes following the 8-bit opcode. This instruction can increment the SP by a value of 0x408 or greater. Increments less than 0x408 should be done with one or two Small Increment instructions.

CANTUNWIND

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

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**	**0**	A15	B15	B14	B13	B12
B11	B10	B3	A14	A13	A12	A11	A10

This two-byte instruction indicates that up to thirteen callee-saved 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 *safe debug* ordering.

When any registers are popped using the "POP bitmask" instruction, the SP is first **implicitly incremented** by the size of the callee-saved register area, rounded up to 8 bytes. This is in addition to any explicit SP increment instructions. However, if the "MV FP, SP" instruction has been used, "POP bitmask" does **not** implicitly increment SP.

POP Bitmask; C64x+ Compact Frame

7	6	5	4	3	2	1	0
1	**0**	**1**	A15	B15	B14	B13	B12
B11	B10	B3	A14	A13	A12	A11	A10

The same as POP Bitmask, but indicates the use of C64x+ *compact frame* layout, which may leave holes on the stack in order to favor the use of SP-autodecrementing stores. The unwinder must be aware of the algorithm used to place the holes and compensate accordingly.

POP Register

7	6	5	4	3	2	1	0
1	**1**	**0**	**0**			n	
		r0				r1	
		r2				...	

In cases where the compiler was unable to maintain safe debug order, or for compilers which choose different layouts, each callee-saved register can be popped individually. The first four bits after the 4-bit opcode indicate the number of registers to be popped. Each subsequent 4-bit nibble represents the encoding of a callee-saved register, or the special value 0xF, which represents a *hole*. If a hole is indicated, the virtual SP should be decremented but no register should be popped.

The 4-bit register encoding is as follows:

Table 22. Register Encoding in Unwinding Instructions

Encoding	Register	Encoding	Register
0000	A15	1000	A14
0001	B15	1001	A13
0010	B14	1010	A12
0011	B13	1011	A11
0100	B12	1100	A10
0101	B11	1101	Reserved
0110	B10	1110	Reserved
0111	B3	1111	"hole"

MV FP, SP

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

This instruction restores SP from FP (A15) instead of incrementing SP. When an FP is available, it is easier to just restore the SP value from the FP. For the DATA_MEM_BANK layout, this may be the only way to restore SP.

_ _c6xabi_pop_rts

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

This instruction indicates that all of the register restoring is done by a call to _ _c6xabi_pop_rts. The behavior of this function should be simulated by the unwinder. _ _c6xabi_pop_rts implicitly restores B3 and does a RET B3.

Restore B3

7	6	5	4	3	2	1	0
1	1	1	0	r	r	r	R

If r represents any register other than B3, this instruction encodes "MV reg, B3", which restores B3 from "reg". This must be performed before any POP instruction in case the POP overwrites the register.

RET B3

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

This instruction encodes a simulated *return*, indicating that unwinding is complete for this frame. Note that the encoding is the same as "Restore B3" but with the source register indicated as B3 itself.

Every sequence of unwinding instructions ends with an explicit or an implicit "RET B3". This instruction can be omitted from the explicit unwinding instructions, and the unwinder will implicitly add it.

11.5.3   24-Bit Unwinding Encoding

PR3 and PR4 use an optimized encoding. Most functions use PR3. If you are optimizing C64x+ code for size use PR4.

31	30-28	27-24	23-17	16-4	3-0
1	000	Index	Stack increment	Register bitmask	Return register

The stack increment is similar to the byte-encoded small constant increment, but it is not biased by 8. The special increment value 0x7F is used to encode "MV FP, SP". If the value of the stack increment is not 0x7F, 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 22. If this register is any register other than B3 itself, B3 should be restored from this register before executing the POP operation (next paragraph).

The bitmask is interpreted as in the byte-encoded POP bitmask instruction. If the personality routine is PR3, the non-compact POP instruction is performed; if the personality routine is PR4, the compact-frame POP instruction is performed. This includes the possibly implicit increment of SP.

11.6 Descriptors

If any local objects need to be destroyed, or if the exception is caught by this function, the EXTAB contains *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			

11.6.1 Encoding of Type Identifiers

Catch descriptors and FESPEC descriptors (Section 11.6.5) 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. The special relocation type R_C6000_EHTYPE is used to mark type_info references in the EXTAB.

The linker encodes a type_info field as a DP-relative offset to the type_info object, thereby preserving position independence of the tables. The offset is relative to the static base of the module that defines the function containing the referenced catch clause or exception specification.

11.6.2 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, PR3, or PR4. If either the length or offset exceed 15-bits, the scope uses the long form encoding and PR2 must be used.

31-17	16	15-1	0
Length	X	Offset	Y
Data for descriptor			

Figure 9. Short Form Scope

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

31-1	0
Length	X
Offset	Y
Data for descriptor	

Figure 10. Long Form Scope

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

11.6.3   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.

31-0
Scope (long or short form)
0 PREL31 program address of landing pad
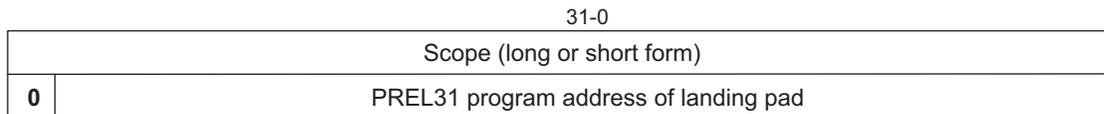
The cleanup descriptor simply contains a single pointer to a cleanup code block containing one or more calls to destructor functions.

11.6.4   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.

31-0	
Scope (long or short form)	0
R PREL31 program address of landing pad	
Type	
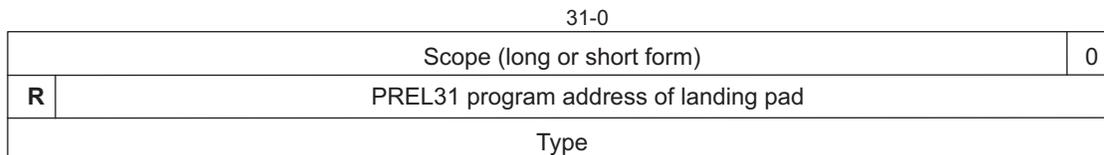
If bit R is 1, the type of the catch clause is a reference type represented by TYPE. If bit R is 0, the type is not a reference type.

The type field is either a reference to a type_info object (relocated via a R_C6000_EHTYPE 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.

11.6.5 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 (but there are exceptions).

	31-0
	Scope (long or short form)
D	Number of type info pointers
	Reference to type_info object
	Reference to type_info object
	. . .
0	(if D == 1) PREL31 program address of landing pad

The first word of the descriptor consists of 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 PREL31 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_C6000_NONE relocation to _ _cxa_call_unexpected.

11.7 Special Sections

All of the exception handling tables are stored in two sections. The EXIDX table is stored in a section called .c6xabi.exidx with type SHT_C6000_UNWIND. The linker must combine all the input .c6xabi.exidx sections into one contiguous .c6xabi.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 .c6xabi.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.

11.8 Interaction With Non-C++ Code

11.8.1 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.

11.8.2 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.

11.9 Interaction With System Features

11.9.1 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.

11.9.2 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.

11.9.3 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 which 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.

11.10 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 PREL31 representation.

$EXIDX_EXTAB

The argument is an EXTAB label to be encoded using the PREL31 representation.

$EXTAB_LP

The argument is a landing pad label to be encoded using the PREL31 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 R_C6000_EHTYPE relocation.

$EXTAB_SCOPE

The argument is an offset into a function. This expression will be used in a scope descriptor to indicate during which portions of the functions it should be applied.

12 DWARF

The C6000 uses the DWARF Debugging Information Format Version 3, also known as DWARF3, to represent information for a symbolic debugger in object files. DWARF3 is documented in http://www.dwarfstd.org/doc/Dwarf3.pdf. This section augments that standard by specifying parts of the representation that are specific to the C6000.

12.1 DWARF Register Names

DWARF3 refers to registers using register name operators, as described in section 2.6.1 of the DWARF3 standard. The operand of a register name operator is a register number representing an architecture register. Table 23 defines the mapping from DWARF3 register numbers/names to C6000 registers.

Table 23. DWARF3 Register Numbers for C6000

DWARF Name	C6000 ISA Register	Description
0-15	A0-A15	
16-31	B0-B15	
32	Reserved	
33	PCE1	E1 Phase Program Counter
34	IRP	Interrupt Return Pointer Register
35	IFR	Interrupt Flag Register
36	NRP	NMI Return Pointer Register
37-52	A16-A31	
53-68	B16-B31	
69	AMR	Address Mode Register
70	CST	Control Status Register
71	ISR	Interrupt Set Register
72	ICR	Interrupt Clear Register
73	IER	Interrupt Enable Register
74	ISTP	Interrupt Service Table Pointer Register
75	IN	Undocumented Control Register
76	OUT	Undocumented Control Register
77	ACR	Undocumented Control Register
78	ADR	Undocumented Control Register
79	FADCR	Floating-Point Adder Configuration Register
80	FAUCR	Floating-Point Auxiliary Configuration Register
81	FMCR	Floating-Point Multiplier Configuration Register
82	GFPGFR	Galois Field Polynomial Generator Function Register
83	DIER	Undocumented Control Register
84	REP	Restricted Entry Point Register
85	TSCL	Time Stamp Counter - Low Half
86	TSCH	Time Stamp Counter - High Half
87	ARP	Undocumented Control Register
88	ILC	SPLOOP Inner Loop Count Register
89	RILC	SPLOOP Reload Inner Loop Count Register
90	DNUM	DSP Core Number Register
91	SSR	Saturation Status Register
92	GPLYA	GMPY Polynomial - A Side Register
93	GPLYB	GMPY Polynomial - B Side Register
94	TSR	Task State Register
95	ITSR	Interrupt Task State Register
96	NTSR	NMI/Exception Task State Register

Table 23. DWARF3 Register Numbers for C6000 (continued)

DWARF Name	C6000 ISA Register	Description
97	EFR	Exception Flag Register
98	ECR	Exception Clear Register
99	IERR	Internal Exception Report Register
100	DMSG	Undocumented Control Register
101	CMSG	Undocumented Control Register
102	DT_DMA_ADDR	Undocumented Control Register
103	DT_DMA_DATA	Undocumented Control Register
104	DT_DMA_CNTL	Undocumented Control Register
105	TCU_CNTL	Undocumented Control Register
106	RTDX_REC_CNTL	Undocumented Control Register
107	RTDX_XMT_CNTL	Undocumented Control Register
108	RTDX_CFG	Undocumented Control Register
109	RTDX_RDATA	Undocumented Control Register
110	RTDX_WDATA	Undocumented Control Register
111	RTDX_RADDR	Undocumented Control Register
112	RTDX_WADDR	Undocumented Control Register
113	MFREG0	Undocumented Control Register
114	DBG_STAT	Undocumented Control Register
115	BRK_EN	Undocumented Control Register
116	HWBP0_CNT	Undocumented Control Register
117	HWBP0	Undocumented Control Register
118	HWBP1	Undocumented Control Register
119	HWBP2	Undocumented Control Register
120	HWBP3	Undocumented Control Register
121	OVERLAY	Undocumented Control Register
122	PC_PROF	Undocumented Control Register
123	ATSR	Undocumented Control Register
124	TRR	Undocumented Control Register
125	TCRR	Undocumented Control Register
126	DESR	Undocumented Control Register
127	DETR	Undocumented Control Register
128	STRM_HOLD	Undocumented Control Register
129	PDATA_O	Undocumented Control Register
130	TCR	Undocumented Control Register

12.2 Call Frame Information

Debuggers need to be able to view and modify the local variables of any function as its execution progresses.

DWARF3 does this by having the compiler keep track of where (in registers or on the stack) a function stores its data. The compiler encodes this information in a byte-coded language specified in Section 6.4 of the DWARF3 standard. This allows the debugger to progressively recreate a previous state by interpreting the byte-coded language. Each function 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. Given the point to which the activation's execution has progressed, the debugger can figure out where all of the function's data is, and can unwind the stack to a previous state, including a previous function activation.

The DWARF3 standard suggests a very large unwinding table, with one row for each code address and one column for each register, virtual or not, including the CFA. Each cell contains unwinding instructions for that register at that point in time (code address).

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

The set of registers includes all the registers listed in Table 23, indexed by their DWARF register numbers from the first column.

For the CFA, the C6000 ABI follows the convention suggested in the DWARF3 standard, defining it as the value of SP (B15) at the call site in the previous frame (that of the calling procedure).

There is no distinct column in the unwinding table for the virtual return address as suggested in Section 6.4.4 of the DWARF3 standard. In accordance with the calling conventions, the return address is represented by the B3 column of the unwinding table.

The unwinding table may include registers that are not present on all C6000 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.

12.3 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   C6000 Code Generation Tools from Texas Instruments
GNU   The GNU Compiler Collection (GCC)

12.4 Vendor Extensions

The DWARF standard allows toolchain vendors to define additional tags and 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 C6000 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 24 defines TI vendor-specific DIE tags that apply to the C6000. Table 24 defines TI vendor-specific attributes.

Table 24. TI Vendor-Specific Tags

Name	Value	Description
DW_TAG_TI_branch	0x4088	Identifies calls and returns

DW_TAG_TI_branch

This tag identifies branches that are used as calls and returns. It is generated as a child of a DW_TAG_subprogram DIE. It has a DW_AT_lowpc attribute corresponding to the location of the branch instruction.

If the branch is a function call, it has a DW_AT_TI_call attribute with non-zero value. It may also have a DW_AT_name attribute that indicates the name of the called function, or a DW_AT_TI_indirect attribute if the callee is not known (as with a call through a pointer).

If the branch is a return, it has a DW_AT_TI_return attribute with non-zero value.

Table 25. TI Vendor-Specific Attributes

Name	Value	Class	Description
DW_AT_TI_symbol_name	0x2001	string	Object file name (mangled)
DW_AT_TI_return	0x2009	flag	Branch is a return
DW_AT_TI_call	0x200A	flag	Branch is a call
DW_AT_TI_asm	0x200C	flag	Function is assembly language
DW_AT_TI_indirect	0x200D	flag	Branch is an indirect call
DW_AT_TI_plt_entry	0x2012	flag	Function is a PLT entry
DW_AT_TI_max_frame_size	0x2014	constant	Activation record size

DW_AT_TI_call

DW_AT_TI_return

DW_AT_TI_indirect

These attributes apply to DW_TAG_TI_branch DIEs, as described previously.

DW_AT_TI_symbol_name

This attribute can appear in any DIE that has a DW_symbol_name. It provides the object-file-level name associated with the variable or function; that is, with any mangling or other alteration applied by the toolchain to the source-level name.

DW_AT_TI_plt_entry

This attribute is added, with a non-zero-value, to DW_TAG_subprogram DIEs corresponding to Procedure Linkage Table entries. Its meaning is similar to that of DW_AT_trampoline.

DW_AT_TI_max_frame_size

This attribute may appear in a DW_TAG_subprogram DIE. It indicates the amount of stack space required for an activation of the function, in bytes. Its intended use is for downstream tools that perform static stack depth analysis.

13 Object Files (Processor Supplement)

The C6000 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 (http://www.sco.com/developers/gabi/2003-12-17/contents.html). This section contains the C6000 processor-specific supplement for Chapter 4 (Object Files). Section 13.5.3 of this document contains the processor-specific supplement for Chapter 5 (Program Loading and Dynamic Linking).

13.1 Registered Vendor Names

The compiler toolsets create and use vendor-specific symbols. To avoid potential conflicts TI encourages vendors to define and use vendor-specific namespaces. The list of currently registered vendors and their preferred shorthand name is given in Table 26.

Table 26. Registered Vendors

Name	Vendor
cxa, _ _cxa	C++ ABI namespace. Applies to all symbols specified by the C++ ABI.
c6xabi, _ _c6xabi	Common namespace for symbols specified by the C6000 EABI.
C6000	Common namespace for symbols specified by the C6000.
TI, _ _TI	Reserved for symbols specific to the TI toolchain. This also represents a composite namespace for all TI processor ABIs.
gnu, _ _gnu	Reserved for symbols specific to the GCC toolchain.

> **NOTE:** The TI or _ _TI specification defines names for processor-specific section types, special sections, and so on. Where there is commonality among different TI processors, such entities are named using TI rather than defining distinct names for each processor. For example, the Exception Table Index Table section type is SHT_TI_EXIDX for all TI processors, rather than SHT_C6000_EXIDX for C6000, SHT_C2000_EXIDX for C2000, and so on.

13.2 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 C6000.

e_indent

The 16-byte ELF identification field identifies the file as an object file and provides machine-independent data with which to decode and interpret the file's contents. Table 27 specifies the values to be used for C6000 object files.

Table 27. ELF Identification Fields

Index	Symbolic Value	Numeric Value	Comments
EI_MAG0		0x7f	Per System V ABI
EI_MAG1		E	Per System V ABI
EI_MAG2		L	Per System V ABI
EI_MAG3		F	Per System V ABI
EI_CLASS	ELFCLASS32	1	32-bit ELF
EI_DATA	ELFDATA2LSB	1	Little-endian
	ELFDATA2MSB	2	Big-endian
EI_VERSION	EV_CURRENT	1	
EI_OSABI	ELFOSABI_C6000_ELFABI	64	Bare-metal dynamic linking platform
EI_OSABI	ELFOSABI_C6000_LINUX	65	MMU-less 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 (Section 14.4) and Linux (Section 15) are two such platforms 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 C6000-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_C6000 (140, 0x8c).

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 C6000-specific flag.

Name	Value	Comment
EF_C6000_REL	0x1	File contains static relocation information

The EF_C6000_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 statically or dynamically.

13.3 Sections

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

13.3.1 Section Indexes

The ABI defines one special section index:

Name	Value	Comment
SHN_C6000_SCOMMON	0xFF00	Common block symbols with near-DP addressing

The SHN_C6000_SCOMMON index identifies common block symbols addressed with near-DP addressing; see Section 13.4.2.

13.3.2 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 28.

Not all these section types are used in the C6000 ABI. Some are specific to the TI toolchain but outside the ABI, and some are used by TI toolchains for architectures other than C6000. They are documented here for completeness, and to reserve the tag values.

Table 28. ELF and TI Section Types

Name	Value	Comment
SHT_C6000_UNWIND	0x70000001	Unwind function table for stack unwinding
SHT_C6000_PREEMPTMAP	0x70000002	DLL dynamic linking pre-emption map
SHT_C6000_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_C6000_UNWIND identifies a section containing unwind function table for stack unwinding. See Section 11 for details.

SHT_C6000_PREEMPTMAP identifies a section containing a C6000 DLL dynamic linking preemption map.

SHT_C6000_ATTRIBUTES identifies a section containing object compatibility attributes. See Section 17.

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 Section 18.

SHT_TI_PHATTRS identifies a section containing additional properties for program segments in an executable or shared object file. See Section 19.

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 C6000.

13.3.3   Extended Section Header Attributes

There are no processor-specific section attribute flags defined. All processor-specific values are reserved to future revisions of this specification. Program header attributes are described in Section 19.

13.3.4   Subsections

C6000 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 13.3.5. For example .text:func1 is an instance of a .text section.

13.3.5 Special Sections

The System V ABI, along with other base documents and other sections of this ABI, defines several sections with dedicated purposes. Table 29 consolidates dedicated sections used by the C6000 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.

All of the section names listed in the table that follows are prefixes. The type and attributes apply to all sections with names that begin with these strings.

Table 29. C6000 Special Sections

Prefix	Type	Attributes
Code Sections		
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.plt	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
Near Data Sections		
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.neardata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.rodata	SHT_PROGBITS	SHF_ALLOC
Far Data Sections		
.far	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.fardata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.const	SHT_PROGBITS	SHF_ALLOC
.fardata:const	SHT_PROGBITS	SHF_ALLOC
Dynamic Data Sections		
.got	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.dsbt	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
Exception Handling Data Sections		
.c6xabi.exidx	SHT_C6000_UNWIND	SHF_ALLOC + SHF_LINK_ORDER
.c6xabi.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

Table 29. C6000 Special Sections (continued)

Prefix	Type	Attributes
.symtab_shndx	SHT_SYMTAB_SHNDX	None
.strtab	SHT_STRTAB	SHF_STRINGS
.shstrtab	SHT_STRTAB	SHF_STRINGS
.note	SHT_NOTE	None
Dynamic Loading Structure		
.dynamic [1]	SHT_DYNAMIC	SHF_ALLOC
.dynsym [1]	SHT_DYNSYM	SHF_ALLOC
.dynstr [1]	SHT_STRTAB	SHF_ALLOC + SHF_STRINGS
.hash [1]	SHT_TAB	SHF_ALLOC
.interp	SHT_PROGBITS	None
Build Attributes		
.c6xabi.attributes	SHT_C6000_ATTRIBUTES	None
Symbolic Debug Sections		
.debug [2]	SHT_PROGBITS	None
Symbol Versioning Sections[3]		
.gnu.version	SHT_GNU_versym	SHF_ALLOC
.gnu.version_d	SHT_GNU_verdef	SHF_ALLOC
.gnu.version_r	SHT_GNU_verneed	SHF_ALLOC
Sections for Thread-Local Storage		
.tbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.tdata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.tdata1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.TI.tls_init	SHT_PROGBITS	SHF_ALLOC
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
.switch	SHT_PROGBITS	SHF_ALLOC
.cinit	SHT_TI_INITINFO	SHF_ALLOC
.const:handler_table	SHT_PROGBITS	SHF_ALLOC
.ppdata	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.ppinfo	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.TI.icode	SHT_TI_ICODE	None
.TI.phattrs	SHT_TI_PHATTRS	None
.TI.preempt.map	SHT_C6000_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 C6000 EABI		
.comment	SHT_PROGBITS	None
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.line	SHT_PROGBITS	None

[1] Whether the .dynamic section and related sections are allocated into memory is platform specific.

[2] Additional sections with names like .debug_info and .debug_line are also used. The .debug section name is a prefix, as are other section names. The type and attributes apply to all sections with names that begin with .debug.

[3] Whether the .dynamic section and related sections are allocated into memory is platform specific.

Table 29. C6000 Special Sections (continued)

Prefix	Type	Attributes
.rodata1	SHT_PROGBITS	SHF_ALLOC

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.

Some sections under the "Sections Unused by the C6000 EABI" heading are sections that are specified by the System V ABI, but are not used or defined under the C6000 ABI. Other sections are used by TI for other devices; these names are reserved.

See Section 7 for details about thread-local storage.

In addition, .common and .scommon are section names used by the linker. These are abstract sections, not actual sections in the object files. The names are a convention in the linker command file for placing variables. These sections should not be used for other purposes.

13.3.6   Section Alignment

Sections containing C6000 code must be at least 32-byte aligned, and padded to 32-byte boundaries. The latter requirement is to avoid misinterpreting adjacent data as a fetch packet header on C64+ and later architectures.

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

13.4   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 C6000 ABI follows the ELF specification with respect to global and weak symbol definitions, and the meaning of symbol values.

13.4.1   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.

In addition, thread-local symbols have a symbol type of STT_TLS.

13.4.2   Common Block Symbols

As described in the ELF specification, symbols with type STT_COMMON are allocated by the linker. The C6000 ABI extends the common block mechanism to accommodate both near and far data addressing. If a common block symbol is addressed using near DP-relative addressing, it must have the processor-specific value SHN_C6000_SCOMMON as its section index. The linker allocates such symbols into a near data section, typically .bss.

Common block symbols addressed with other addressing forms should have section index SHN_COMMON, as described in the base ELF specification. Such symbols may be allocated into a far data section, typically .far.

13.4.3 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.

The C6000 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

13.4.4 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 26.
*   Global symbols (STB_GLOBAL, STB_WEAK) ending with any of $$Base or $$Limit
*   Symbols matching the pattern ${Tramp}${I|L|S}[$PI]$$symbol
*   Compiler generated temporary symbols beginning with P , O, C (as described in Section 4.5)

13.4.5 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.

13.5 Relocation

The ELF relocations for C6000 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. This results in slightly more relocation types than the older C6000 COFF ABI. Relocation types are not compatible between COFF and ELF.

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 Elf32_Rela or Elf32_Rel. The Rela entries contain an explicit addend which is used in the relocation calculation. Entries of type Rel use the relocatable field itself as the addend. Certain relocations are identified as Rela only. For the most part these correspond to the upper 16 bits of a 32-bit address, where the resultant value depends on carry propagation from lower bits that are not available in the field. Where Rela is specified, an implementation must honor this requirement. An implementation may choose to use Rel or Rela type relocations for other relocations.

13.5.1 Relocation Types

Relocation types are described using two tables. Table 30 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 31 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 30.

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 Elf32_rel relocations, A is encoded into the relocatable field according to Table 31. For Elf32_Rela relocations, A is given by the r_addend field of the relocation entry.
PC	The address of the container containing the field.
FP(x)	The address of the fetch packet containing the instruction at address x; that is: FP(x) := x & 0xFFFFFFE0.
P	The fetch packet address of the instruction being relocated; that is: P := FP(PC).
B	The base address of the data segment for the current load module. This location is marked by the symbol _ _c6xabi_DSBT_BASE, and is the value of the DP register when the program is executing.
GOT(S)	The address of the Global Offset Table (GOT) entry of the symbol (S) associated with the relocation.
TBR(x)	The offset of x from the Thread-Local Storage (TLS) Block Base. See Section 7 for details about thread-local storage.
TPR(x)	The offset of x from the thread-pointer (TP).
TLS(x)	The TLS Descriptor for x, which contains the module-id and TBR offset of x.
TLSMOD(x)	The TLS module identifier of the module that defines x.

Table 30. C6000 Relocation Types

Name	Value	Operation	Constraints
R_C6000_NONE	0		
R_C6000_ABS32	1	S + A	
R_C6000_ABS16	2	S + A	
R_C6000_ABS8	3	S + A	
R_C6000_PCR_S21	4	S + A – P	
R_C6000_PCR_S12	5	S + A – P	
R_C6000_PCR_S10	6	S + A – P	
R_C6000_PCR_S7	7	S + A – P	
R_C6000_ABS_S16	8	S + A	
R_C6000_ABS_L16	9	S + A	
R_C6000_ABS_H16	10	S + A	Rela only
R_C6000_SBR_U15_B	11	S + A – B	
R_C6000_SBR_U15_H	12	S + A – B	
R_C6000_SBR_U15_W	13	S + A – B	
R_C6000_SBR_S16	14	S + A – B	
R_C6000_SBR_L16_B	15	S + A – B	
R_C6000_SBR_L16_H	16	S + A – B	
R_C6000_SBR_L16_W	17	S + A – B	
R_C6000_SBR_H16_B	18	S + A – B	Rela only
R_C6000_SBR_H16_H	19	S + A – B	Rela only
R_C6000_SBR_H16_W	20	S + A – B	Rela only
R_C6000_SBR_GOT_U15_W	21	GOT(S) + A – B	
R_C6000_SBR_GOT_L16_W	22	GOT(S) + A – B	
R_C6000_SBR_GOT_H16_W	23	GOT(S) + A – B	Rela only
R_C6000_DSBT_INDEX	24	DSBT Index of this static link unit	
R_C6000_PREL31	25	S + A – PC	

Table 30. C6000 Relocation Types (continued)

Name	Value	Operation	Constraints
R_C6000_COPY	26	Load-time copy of preempted symbol	ET_EXEC only
R_C6000_JUMP_SLOT	27	S + A	ET_EXEC / ET_DYN
R_C6000_EHTYPE	28	S + A - B	
R_C6000_PCR_H16	29	S - FP(P - A)	Rela only
R_C6000_PCR_L16	30	S - FP(P - A)	Rela only
Reserved	31		
Reserved	32		
R_C6000_TBR_U15_B	33	TBR(S)	Static only
R_C6000_TBR_U15_H	34	TBR(S)	Static only
R_C6000_TBR_U15_W	35	TBR(S)	Static only
R_C6000_TBR_U15_D	36	TBR(S)	Static only
R_C6000_TPR_S16	37	TBR(S)	
R_C6000_TPR_U15_B	38	TPR(S)	
R_C6000_TPR_U15_H	39	TPR(S)	
R_C6000_TPR_U15_W	40	TPR(S)	
R_C6000_TPR_U15_D	41	TPR(S)	
R_C6000_TPR_U32_B	42	TPR(S)	Dynamic only
R_C6000_TPR_U32_H	43	TPR(S)	Dynamic only
R_C6000_TPR_U32_W	44	TPR(S)	Dynamic only
R_C6000_TPR_U32_D	45	TPR(S)	Dynamic only
R_C6000_SBR_GOT_U15_W_TLSMOD	46	GOT(TLSMOD(S)) + A – B	Static only
R_C6000_SBR_GOT_U15_W_TBR	47	GOT(TBR(S)) + A - B	Static only
R_C6000_SBR_GOT_U15_W_TPR_B	48	GOT(TPR(S))+A-B	Static only
R_C6000_SBR_GOT_U15_W_TPR_H	49	GOT(TPR(S))+A-B	Static only
R_C6000_SBR_GOT_U15_W_TPR_W	50	GOT(TPR(S))+A-B	Static only
R_C6000_SBR_GOT_U15_W_TPR_D	51	GOT(TPR(S))+A-B	Static only
R_C6000_SBR_GOT_L16_W_TLSMOD	52	GOT(TLSMOD(S)) + A - B	Static only
R_C6000_SBR_GOT_L16_W_TBR	53	GOT(TBR(S)) + A - B	Static only
R_C6000_SBR_GOT_L16_W_TPR_B	54	GOT(TPR(S))+A-B	Static only
R_C6000_SBR_GOT_L16_W_TPR_H	55	GOT(TPR(S))+A-B	Static only
R_C6000_SBR_GOT_L16_W_TPR_W	56	GOT(TPR(S))+A-B	Static only
R_C6000_SBR_GOT_L16_W_TPR_D	57	GOT(TPR(S))+A-B	Static only
R_C6000_SBR_GOT_H16_W_TLSMOD	58	GOT(TLSMOD(S)) + A - B	Static only
R_C6000_SBR_GOT_H16_W_TBR	59	GOT(TBR(S)) + A - B	Static only
R_C6000_SBR_GOT_H16_W_TPR_B	60	GOT(TPR(S))+A-B	Static only
R_C6000_SBR_GOT_H16_W_TPR_H	61	GOT(TPR(S))+A-B	Static only
R_C6000_SBR_GOT_H16_W_TPR_W	62	GOT(TPR(S))+A-B	Static only
R_C6000_SBR_GOT_H16_W_TPR_D	63	GOT(TPR(S))+A-B	Static only
R_C6000_TLSMOD	64	TLSMOD(S)	Dynamic only
R_C6000_TBR_U32	65	TBR(S)	Dynamic only
R_C6000_ALIGN	253	None	ET_REL only
R_C6000_FPHEAD	254	None	ET_REL only
R_C6000_NOCMP	255	None	ET_REL only

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

The R_C6000_ABS8/16/32 relocations directly encode the relocated address of a symbol into 8-, 16-, or 32-bit fields. They are commonly used for initialized data, not for instructions. The signedness of the field is unspecified; that is, they are used for both signed and unsigned values.

```
.field X,32          ; R_C6000_ABS32
.field X,16          ; R_C6000_ABS16
.field X,8           ; R_C6000_ABS8
```

The PCR relocations encode signed PC-relative branch displacements. They are scaled to 32-bit (word) units. Displacements are computed relative to the fetch packet of the source instruction.

```
B      func          ; R_C6000_PCR_S21
CALLP  func,B3       ; R_C6000_PCR_S21
BNOP   func          ; R_C6000_PCR_S12
BPOS   func,A10      ; R_C6000_PCR_S10
BDEC   func,A1       ; R_C6000_PCR_S10

ADDKPC func,B3,4     ; R_C6000_PCR_S7
```

Relocations with L16 in their names encode the lower 16 bits of a 32-bit address or offset. Those containing H16 encode the upper 16 bits, and are always Rela. Relocations with S16 encode a signed 16-bit value (generally not part of an address). Those with U15 encode an unsigned 15-bit DP-relative displacement.

```
MVHL   sym,A0        ; R_C6000_ABS_L16
MVKH   sym,A0        ; R_C6000_ABS_H16

MVK    const16,A0    ; R_C6000_ABS_S16   sign extend const16 into A0

MVKLH  const16,A0    ; R_C6000_ABS_L16   move const16 into A0[16:31]
```

The PCR_L16 and PCR_H16 relocations encode the lower and upper bits, respectively, of a PC-relative offset between a target address and the fetch packet address of a reference PC (the "base PC"). The offset from the fetch packet of the current instruction to the base PC is encoded in the addend field; that is A := (P-base). The relocation then computes S-FP(P-A), resulting in the offset between S and FP(base). These relocations are used to address objects in different sections using PC-relative addressing, as described in Section 5.1.

```
MVK    $PCR_OFFSET(sym,base),A0     ; R_C6000_PCR_L16
MVKH   $PCR_OFFSET(sym,base),A0     ; R_C6000_PCR_H16
```

The SBR_U15 relocations encode 15-bit unsigned DP-relative offsets for near-DP data addressing. They are scaled according to the access width: 32-bit word (_W), 16-bit halfword (_H), or byte (_B).

```
LDB    *+DP(sym),A1 ; R_C6000_SBR_U15_B
ADDAB  DP,sym,A2    ; R_C6000_SBR_U15_B

LDH    *+DP(sym),A1 ; R_C6000_SBR_U15_H
ADDAH  DP,sym,A2    ; R_C6000_SBR_U15_H

LDW    *+DP(sym),A1 ; R_C6000_SBR_U15_W
ADDAW  DP,sym,A2    ; R_C6000_SBR_U15_W
```

The other SBR relocations are used to encode the high and low parts of 32-bit DP-relative offsets, for far DP-relative addressing. In the examples that follow:

- $bss represents the data segment base address, corresponding to _ _c6xabi_DSBT_BASE (the value in DP)
- $DPR_byte(sym) represents the DP-relative offset in bytes
- $DPR_hword(sym) represents the DP-relative offset divided by 2
- $DPR_word(sym) represents the DP-relative offset divided by 4

```
MVK    (sym - $bss),A0       ; R_C6000_SBR_S16
```

```
        MVKL    $DPR_byte(sym),A0    ; R_C6000_SBR_L16_B
        MVKH    $DPR_byte(sym),A0    ; R_C6000_SBR_H16_B


        MVKL    $DPR_hword(sym),A0   ; R_C6000_SBR_L16_H
        MVKH    $DPR_hword(sym),A0   ; R_C6000_SBR_H16_H


        MVKL    $DPR_word(sym),A0    ; R_C6000_SBR_L16_W
        MVKH    $DPR_word(sym),A0    ; R_C6000_SBR_H16_W
```

The SBR_GOT relocations correspond to the same instructions and encodings as the SBR relocations, but refer to the DP-relative GOT address of the referenced symbol instead of the symbol itself. Typically the GOT is accessed with near DP-relative addressing, so R_C6000_DBR_GOT_U15_W is used. When the GOT is far the offset is generated with MVKL/MVKH with the other two relocations (see Section 6.6). In the examples that follow,

- GOT(sym) is the DP-relative offset of the GOT entry for sym, in bytes
- $DPR_GOT(sym) is the DP-relative offset of the GOT entry for sym, in words

```
        LDW     *+DP[GOT(sym)],A0    ; R_C6000_SBR_GOT_U15_W


        MVKL    $DPR_GOT(sym), A0    ; R_C6000_SBR_GOT_L16_W
        MVKH    $DPR_GOT(sym), A0    ; R_C6000_SBR_GOT_H16_W
```

The R_C6000_DSBT_INDEX encodes the index into the Data Segment Base Table of the current load module. It is present only in files that use the DSBT model for position independence. See Section 6.7.

```
        LDW     *+DP($DSBT_INDEX(__c6xabi_DSBT_BASE)),DP  ; R_C6000_DSBT_INDEX
```

R_C6000_COPY is used to mark a duplicate symbol defined in an executable that preempts a library definition, under the import-as-own convention described in Section 15.9. When the executable is loaded, the dynamic loader must copy any initial value from the library's definition to that of the executable. This relocation type is present only in the dynamic relocation table of an executable file (ET_EXEC).

R_6000_JUMP_SLOT is used to mark GOT entries that refer to imported functions and are referred to only from PLT entries, and are therefore subject to lazy binding as described in Section 15.6. R_C6000_JUMP_SLOT relocations occur only in executables and shared objects, and only in the DT_JMPREL section of the dynamic relocation table.

R_C6000_PREL31 is used to encode code addresses in exception handling tables. R_C6000_EHTYPE is used to encode typeinfo addresses in exception handling tables. See Section 11.2.

Relocations with values from 33 to 65 are for use with Thread-Local Storage (TLS). These relocations include the R_C6000_TBR_*, R_C6000_TPR_*, R_C6000_SBR_GOT_*_W_T*, R_C6000_TLSMOD, and R_C6000_TBR_U32 relocations. See Section 7 for details about thread-local storage. Examples that use these TLS relocations are provided in Section 7.4.

R_C6000_ALIGN and R_C6000_FPHEAD are used as markers for the C64+ compressor. They have no effect under the ABI. A downstream tool that combines relocatable files (ET_REL) into other relocatable files, such as partial link, should either preserve them or mark the sections in which they occur with R_C6000_NOCMP.

R_C6000_NOCMP marks a section as being uncompressable.

13.5.2 Relocation Operations

Table 31 provides detailed information on how each relocation is encoded and performed. The table uses the following 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. Sign-extension is conceptually performed to the width of the address space.

ZE(x)   Zero-extended value of x. Zero-extension is conceptually performed to the width of the address space.

For relocation types 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 $[0 \ldots 2^{FS})$.
- A relocation whose signedness is indicated as *either* overflows if the encoded value falls outside the half-open interval $[-2^{FS-1} \ldots 2^{FS})$.
- The R_C6000_DSBT_INDEX relocation overflows if the encoded value is equal to or larger than the size of the module's DSBT table.

Table 31. C6000 Relocation Operations

Relocation Name	Signedness	Field [CS, O, FS] (F)	Addend (A)	Result (R)	Overflow Check	Encoded Value (EV)
R_C6000_NONE	none	[32, 0, 32]	none	none	no	none
R_C6000_ABS32	either	[32, 0, 32]	F	S + A	no	R
R_C6000_ABS16	either	[16, 0, 16]	SE(F)	S + A	yes	R
R_C6000_ABS8	either	[8, 0, 8]	SE(F)	S + A	yes	R
R_C6000_PCR_S21	signed	[32, 7, 21]	SE(F << 2)	S + A – P	yes	R >> 2
R_C6000_PCR_S12	signed	[32, 16, 12]	SE(F << 2)	S + A – P	yes	R >> 2
R_C6000_PCR_S10	signed	[32, 13, 10]	SE(F << 2)	S + A – P	yes	R >> 2
R_C6000_PCR_S7	signed	[32, 16, 7]	SE(F << 2)	S + A – P	yes	R >> 2
R_C6000_ABS_S16	signed	[32, 7, 16]	SE(F)	S + A	yes	R
R_C6000_ABS_L16	none	[32, 7, 16]	F	S + A	no	R
R_C6000_ABS_H16	none	[32, 7, 16]	r_addend	S + A	no	R >> 16
R_C6000_SBR_U15_B	unsigned	[32, 8, 15]	ZE(F)	S + A – B	yes	R
R_C6000_SBR_U15_H	unsigned	[32, 8, 15]	ZE(F << 1)	S + A – B	yes	R >> 1
R_C6000_SBR_U15_W	unsigned	[32, 8, 15]	ZE(F << 2)	S + A – B	yes	R >> 2
R_C6000_SBR_S16	signed	[32, 7, 16]	SE(F)	S + A – B	yes	R
R_C6000_SBR_L16_B	unsigned	[32, 7, 16]	ZE(F)	S + A – B	no	R
R_C6000_SBR_L16_H	unsigned	[32, 7, 16]	ZE(F << 1)	S + A – B	no	R >> 1
R_C6000_SBR_L16_W	unsigned	[32, 7, 16]	ZE(F << 2)	S + A – B	no	R >> 2
R_C6000_SBR_H16_B	unsigned	[32, 7, 16]	r_addend	S + A – B	no	R >> 16
R_C6000_SBR_H16_H	unsigned	[32, 7, 16]	r_addend	S + A – B	no	R >> 17
R_C6000_SBR_H16_W	unsigned	[32, 7, 16]	r_addend	S + A – B	no	R >> 18
R_C6000_SBR_GOT_U15_W	unsigned	[32, 8, 15]	ZE(F << 2)	GOT(s) + A – B	yes	R >> 2
R_C6000_SBR_GOT_L16_W	unsigned	[32, 7, 16]	ZE(F << 2)	GOT(s) + A – B	no	R >> 2
R_C6000_SBR_GOT_H16_W	unsigned	[32, 7, 16]	r_addend	GOT(s) + A – B	no	R >> 18
R_C6000_DSBT_INDEX	unsigned	[32, 8, 15]	none	DSBT Index	yes	R

Table 31. C6000 Relocation Operations (continued)

Relocation Name	Signedness	Field [CS, O, FS] (F)	Addend (A)	Result (R)	Overflow Check	Encoded Value (EV)
R_C6000_PREL31	none	[32, 0, 31]	SE(F << 1)	S + A - PC	no	R >> 1
R_C6000_COPY	none	[32, 0, 32]	none	F	no	F
R_C6000_JUMP_SHOT	either	[32, 0, 32]	F	S + A	no	R
R_C6000_EHTYPE	either	[32, 0, 32]	F	S + A - B	no	R
R_C6000_PCR_H16	signed	[32, 7, 16]	r_addend	S - FP(P - A)	no	R >> 16
R_C6000_PCR_L16	none	[32, 7, 16]	r_addend	S - FP(P - A)	no	R
R_C6000_TBR_U15_B	unsigned	[32,8,15]	ZE(F)	TBR(S)	Yes	R
R_C6000_TBR_U15_H	unsigned	[32,8,15]	ZE(F<<1)	TBR(S)	Yes	R >> 1
R_C6000_TBR_U15_W	unsigned	[32,8,15]	ZE(F<<2)	TBR(S)	Yes	R >> 2
R_C6000_TBR_U15_D	unsigned	[32,8,15]	ZE(F<<3)	TBR(S)	Yes	R >> 3
R_C6000_TPR_S16	Signed	[32,7,16]	SE(F)	TBR(S)	Yes	R
R_C6000_TPR_U15_B	Unsigned	[32,8,15]	ZE(F)	TPR(S)	Yes	R
R_C6000_TPR_U15_H	Unsigned	[32,8,15]	ZE(F<<1)	TPR(S)	Yes	R >> 1
R_C6000_TPR_U15_W	Unsigned	[32,8,15]	ZE(F<<2)	TPR(S)	Yes	R >> 2
R_C6000_TPR_U15_D	Unsigned	[32,8,15]	ZE(F<<3)	TPR(S)	Yes	R >> 3
R_C6000_TPR_U32_B	Unsigned	[32,0,326]	ZE(F)	TPR(S)	No	R
R_C6000_TPR_U32_H	Unsigned	[32,0,326]	ZE(F<<1)	TPR(S)	No	R >> 1
R_C6000_TPR_U32_W	Unsigned	[32,0,326]	ZE(F<<2)	TPR(S)	No	R >> 2
R_C6000_TPR_U32_D	Unsigned	[32,0,326]	ZE(F<<3)	TPR(S)	No	R >> 3
R_C6000_SBR_GOT_U15_W_TLSMOD	Unsigned	[32,8,15]	ZE(F<<2)	GOT(TLSMOD(S)) + A - B	Yes	R >> 2
R_C6000_SBR_GOT_U15_W_TBR	Unsigned	[32,8,15]	ZE(F<<2)	GOT(TBR(S)) + A - B	Yes	R >> 2
R_C6000_SBR_GOT_U15_W_TPR_B	Unsigned	[32,8,15]	ZE(F<<2)	GOT(TBR(S)) + A - B	Yes	R >> 2
R_C6000_SBR_GOT_U15_W_TPR_H	Unsigned	[32,8,15]	ZE(F<<2)	GOT(TBR(S)) + A - B	Yes	R >> 2
R_C6000_SBR_GOT_U15_W_TPR_W	Unsigned	[32,8,15]	ZE(F<<2)	GOT(TBR(S)) + A - B	Yes	R >> 2
R_C6000_SBR_GOT_U15_W_TPR_D	Unsigned	[32,8,15]	ZE(F<<2)	GOT(TBR(S)) + A - B	Yes	R >> 2
R_C6000_SBR_GOT_L16_W_TLSMOD	Unsigned	[32,7,16]	ZE(F<<2)	GOT(TLSMOD(S)) + A - B	No	R >> 2
R_C6000_SBR_GOT_L16_W_TBR	Unsigned	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	No	R >> 2
R_C6000_SBR_GOT_L16_W_TPR_B	Unsigned	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	No	R >> 2
R_C6000_SBR_GOT_L16_W_TPR_H	Unsigned	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	No	R >> 2
R_C6000_SBR_GOT_L16_W_TPR_W	Unsigned	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	No	R >> 2
R_C6000_SBR_GOT_L16_W_TPR_D	Unsigned	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	No	R >> 2
R_C6000_SBR_GOT_H16_W_TLSMOD	Unsigned	[32,7,16]	ZE(F<<2)	GOT(TLSMOD(S)) + A - B	No	R >> 18
R_C6000_SBR_GOT_H16_W_TBR	Unsigned	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	No	R >> 18
R_C6000_SBR_GOT_H16_W_TPR_B	Unsigned	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	No	R >> 18
R_C6000_SBR_GOT_H16_W_TPR_H	Unsigned	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	No	R >> 18
R_C6000_SBR_GOT_H16_W_TPR_W	Unsigned	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	No	R >> 18
R_C6000_SBR_GOT_H16_W_TPR_D	Unsigned	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	No	R >> 18
R_C6000_TLSMOD	Unsigned	[32,0,32]	F	TLSMOD(S)	No	R
R_C6000_TBR_U32	Unsigned	[32,0,32]	F	TBR(S)	No	R
R_C6000_FPHEAD	none	none	none	none	no	none
R_C6000_NOCMP	none	none	none	none	no	none

13.5.3 Relocation of Unresolved Weak References

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

• When used in an absolute relocation type (R_C6000_ABS*) the reference resolves to zero.

• When used in a base-relative relocation type (R_C6000_SBR*) the reference resolves to the static base address (B).

When used in a R_C6000_PCR_S21 relocation and the instruction to be relocated has the following form:

```
B.S2  sym     ; R_C6000_PCRS21
```

Then the instruction is replaced with:

```
B.B2  B3
```

All other cases are non-conformant with the ABI.

NOTE: As required elsewhere in this specification, if the weak symbol is resolved and the 21-bit PC-relative address cannot reach the target destination, the linker must generate a trampoline to implement the relocation.

---

14   Program Loading and Dynamic Linking (Processor Supplement)

In general, *program loading* describes the steps involved in taking a program represented as an ELF file—or in the case of dynamic linking, more than one ELF file—and beginning its execution. By its nature, this process is platform and system specific.

Dynamic linking is a set of related mechanisms that enables programs to consist of separately built components that are linked and relocated at load time, and to share those components among multiple executables.

A system may use a subset of the mechanisms depending on its specific requirements. For example, a bare-metal platform running only one process may require dynamic linking and loading, but not require position independence or shared objects.

This part of the ABI is based on Chapter 5 of the System V ABI standard (http://www.sco.com/developers/gabi/2003-12-17/contents.html), which describes object file information and system actions that create running programs. This section contains a processor-specific supplement to that standard for those elements that are common to most C6000-based systems. This section also defines one specific profile, called the Bare-Metal Dynamic Linking Model.

The other specific profile defined by this ABI is the Linux model. The processor specific supplement to the System V ABI standard for Linux is in Section 15.

14.1   Program Header

The program header contains the following fields.

p_type

The C6000 defines one processor-specific segment type for the p_type field in the program header.

Name	Value	Comment
PT_C6000_PHATTR	0x70000000	Extended Segment Attributes

The PT_C6000_PHATTR segment type identifies the segment as containing additional descriptive information about the other PT_LOAD segments in the program. The segment contains a single section of type SHT_TI_PHATTRS. Program header attributes are described in more detail in Section 19.

p_vaddr, p_paddr

The C6000 does not currently have virtual addressing. Both the p_vaddr and p_paddr fields indicate the execution address of the segment. Segments that are loaded at one address and copied to another to execute are represented in the object file by two distinct segments: a load-image segment containing the segment's code or data whose address fields refer to the load address; and an uninitialized run-image segment whose address fields refer to the run address. The application is responsible for copying the contents of the load image to the run address at the appropriate time.

p_flags

There is one processor-specific segment flag defined for C6000.

Name	Value	Comment
PF_C6000_DPREL	0x10000000	Accessed using DP-relative addressing

The PF_C6000_DPREL flag identifies segments that are accessed using DP-relative addressing, and therefore subject to post-link placement constraints. A position independent module will not typically contain dynamic relocations for DP-relative addressing. If there are multiple DP-relative segments, their position relative to the DP (and therefore to each other) must be maintained. This flag serves to identify such segments to a dynamic loader or other post-link agent so that it can coordinate their allocation.

There are some secondary segment attributes that are used by the TI toolchain. Due to the limited number of available flags, we have defined an alternate mechanism for additional segment attributes: the program header attributes table described in Section 19.

p_align

> As described in the System V ABI, loadable segments are aligned in the file such that their p_vaddr (address in memory) and p_offset (offset in the file) are congruent, modulo p_align. In systems with virtual memory, p_align generally specifies the page size. Unless specified for a specific platform, for the C6000 the meaning and setting of p_align is unspecified.

14.1.1 Base Address

Position independent code can be loaded and run at any address—not necessarily the address specified in the p_vaddr field of the program header—without requiring load-time relocation. However, since segments may refer to each other using relative offsets, their relative positions must be maintained even if they are loaded somewhere other than the location given by their p_vaddrs. The System V ABI refers to the displacement between segments. specified and actual addresses as the *base address*.

The degree to which position-independent segments can be loaded at a different address is platform-specific. However, there are a few universal rules:

- Segments that are not position independent must either be loaded at their specified address or relocated at load time.
- Segments that have the PHA_BOUND attribute must be loaded at their specified address.

14.1.2 Segment Contents

The base ABI (this section) does not define any requirements for what segments must be present or what their contents are. For example, a C6000 program may contain any number of code and data segments, including multiple code segments, multiple DP-relative segments, and multiple absolute data segments, as described in Section 4 and Section 5. Specific platforms may have their own requirements: for example some high-level operating systems may constrain programs to have only one code and one data segment, or perhaps just one segment for both.

14.1.3 Bound and Read-Only Segments

As described in Section 19.2, there is a mechanism to annotate segments with additional properties. The mechanism is used to represent properties that apply to ROM-based segments.

A segment marked with the attribute PHA_BOUND is bound to its specified 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.

A segment marked with the attribute PHA_READONLY indicates that its contents are locked and not subject to any relocations or other downstream changes. This property applies to sections that are located in ROM. A dynamic loader can use this as a hint to avoid relocation processing for such segments.

The difference between a PHA_READONLY segment and one with a segment permission of PF_R (read only) in its program header is that a PF_R segment is usually modifiable by the loader but not by the program itself, whereas a PHA_READONLY segment is not modifiable by either.

14.1.4 Thread-Local Storage

Thread-Local Storage (TLS) is a storage class that allows a program to define thread-specific variables with static storage durations. A TLS variable or thread-local variable is a global/static variable that is instanced once per thread. See Section 7 for details about thread-local storage.

The C6000 EABI supports TLS, but this is dependent on whether the runtime operating system's thread library implements the __c6xabi_get_tp() function and other aspects of TLS support.

Thread-local variables are represented in ELF object files and modules similarly to static data. The difference is that ELF requires that thread-local variables be allocated in sections with the SHF_TLS flag set in relocatable files. The ELF specification requires that the section names .tdata and .tbss be used for initialized and uninitialized thread-local storage, respectively. These sections have read-write permission.

In modules, ELF requires that the TLS segment be indicated by the PT_TLS segment type. This segment is read-only. The PT_TLS segment is the TLS Image.

Thread-local symbols have a symbol type of STT_TLS.

14.2 Program Loading

There are many system-specific aspects of loading a program and starting its execution. This section describes in general terms aspects of the process that are common to most systems, with an emphasis on items that are specific to C6000.

These steps may be performed by a combination of an offline agent such as a host-based loader, run-time components of the target system such an operating system, or library components that are linked into the program itself such as self-boot code.

In general, loading a program consists of four series of actions: creating the process image, initializing the execution environment, executing the program, and performing termination actions.

Creating the process image involves copying the program and its subcomponents into memory and performing relocation if needed. These steps must necessarily be performed by some external agent such as a host-based loader or operating system.

Initializing the execution environment involves steps that must occur before the program starts running (i.e. before main is called). These steps can be performed either by an external agent or by the program itself. Likewise, termination actions occur when main returns (or calls exit), and can be performed either externally or by the program.

Table 32, Table 33 and Table 34 list the steps to create, initialize, and terminate a program. While the order of the steps is not absolute, there are dependencies that must be honored. The column labeled "DL Only" indicates steps that apply only to systems using dynamic linking or loading.

Table 32. Steps to Create a Process Image from an ELF Executable

Step		DL Only
1.	Determine the address for each loadable segment. In bare-metal or non-dynamic systems, this is usually the address in the p_vaddr field of the segment's program header. Other considerations are discussed in Section 14.1.	
2.	Initialize the memory system and allocate memory.	
3.	Copy the contents of each segment into memory. If a segment has unfilled space (that is, its file size is less than its memory size), initialize the unfilled space to 0.	
4.	Create the process image for dependent libraries. Dependent libraries are identified by DT_NEEDED entries in the dynamic section. Libraries should be checked for compatibility with respect to target processor, ABI, OS, and DSBT indexing.	√
5.	Assign DSBT indexes for this module and all dependent libraries. Indexes must be unique among an executable and all its libraries. A given instance of a library must have only one index even if shared among multiple programs. See Section 6.7.	√
6.	Resolve symbolic references between imported and exported symbols. Symbols with dynamic linkage are represented in the dynamic symbol table, identified by the DT_SYMTAB tag in the dynamic section. Exported symbols with visibility STV_DEFAULT may be preempted by definitions from parent files. For symbols that have version information, identified by a DT_SYMVER tag in the dynamic section, the loader should insure that references are matched up with the proper definitions.	√
7.	Perform relocation if needed. Load-time relocations are indicated by DT_REL and/or DT_RELA tags in the dynamic section. Relocations are processed as specified in Section 13.5.	√
8.	Initialize DSBT entries for the executable and dependent libraries. This step has two parts. First, the DSBT for the current executable must be initialized with the static base address of all loaded modules (including itself, at index 0). Second, the DSBTs for all the other loaded modules must be updated with this module's base address, at the index assigned to this module in step 5.	√
9.	Marshall command line arguments and environment variables. This step is platform specific.	

Table 33. Steps to Initialize the Execution Environment

Step		DL Only
10.	Set SP. SP (B15) should be set to the value of the symbol _ _TI_STACK_END, properly aligned on an 8-byte boundary.	
11.	Set DP. DP (B14) should be set to the value of the symbol _ _c6xabi_DSBT_BASE, corresponding to the lowest address of any DP-relative segment.	

Table 33. Steps to Initialize the Execution Environment (continued)

Step		DL Only
12.	Initialize variables. For self-booting ROM-based systems, some mechanism is required to initialize RAM-based (read-write) variables with their initial values. The mechanism is toolchain and platform specific. One such mechanism, implemented in the TI tools, is described in Section 18.	
13.	Perform preinit calls. These are calls to initialization functions defined to occur before those of dependent libraries. Preinit calls are identified by the DT_PREINIT_ARRAY tag in the dynamic section, as specified in the System V ABI.	√
14.	Recursively perform the initialization calls (step 15) for dependent libraries, according to the ordering defined in the section on Initialization and Termination Functions of the System V ABI.	√
15.	Perform initialization calls. Generally these are calls to constructors for global objects defined in the module. They occur *after* those of dependent libraries. Pointers to initialization functions are stored in a table. In files with dynamic information, the table is identified by the DT_INIT_ARRAY and/or DT_INIT tags. In other files, the table is delimited by a pair of global symbols: _ _TI_INITARRAY_Base and _ _TI_INITARRAY_Limit.	
16.	Branch to the entry point. The entry point is specified in the e_entry field of the ELF header. On systems with some underlying software fabric such an OS, the entry point is typically the main function. On bare-metal systems, most of the initialization steps listed in this table may be performed by the program itself, via library code that executes before main. In that case the ELF entry point is the address of that code. For example the TI tools provide an entry routine called _c_int00 that begins the sequence in Step 10 (set SP) once the process image is created.	

Table 34. Termination Steps

Step		DL Only
17.	Perform atexit calls. Functions registered by atexit are called, in reverse order of registration.	
18.	Recursively perform the termination calls (step 19) for dependent libraries, according to the ordering defined in the System V ABI.	√
19.	Call the termination functions for the current module, identified by the DT_FINI and/or DT_FINI_ARRAY tags.)	√

14.3 Dynamic Linking

Dynamic linking is set of related mechanisms that enable programs to consist of separately built components. The mechanisms consist of:

- **Linkage mechanisms**—to support references between separately linked objects. These consist primarily of the dynamic section and related subcomponents such as the dynamic symbol table and dynamic relocations.
- **Sharing mechanisms**—so each application sharing the code can have private copies of its data at different locations. Systems with MMUs typically rely on virtual to physical address translation. The C6000, lacking an MMU, relies on a mechanism called the Data Segment Base Table, as specified in Section 6.
- **Addressing mechanisms**–to support linkage and sharing. These are also specified in general in Section 6.

A system may use a subset of the mechanisms depending on its specific requirements. For example, a bare-metal platform running only one process may require dynamic linking and loading, but not require position independence or shared objects.

The ABI currently defines two specific profiles with different levels of capability. The first is the Bare-Metal Dynamic Linking Model, described in Section 14.4. The other is the Linux model, described in Section 15.

14.3.1 Program Interpreter

As described in Section 14.2, program loading is performed by some external agent. On Linux and probably other OS-based systems, the agent responsible for performing this function is stored in the executable itself as the PT_INTERP tag of the program header. Usually this is the dynamic loader, for example ld.so.

Bare-metal executables do not rely on an interpreter; the system is responsible for knowing how to load the program. A bare-metal dynamic executable may contain dynamic information in a PT_DYNAMIC segment but not have at PT_INTERP entry.

14.3.2 Dynamic Section

As specified in the System V ABI, a dynamic linked program has an entry of type PT_DYNAMIC in its program header. This entry points to a special section called .dynamic, having section type SHT_DYNAMIC, that contains information relating to dynamic linking and loading. The dynamic section refers to other sections such as dynamic symbol table sections and dynamic relocation sections, collectively called *dynamic information*.

The dynamic information may or may not be contained within the loadable image of the program (that is, within one or more PT_LOAD segments), depending on platform-specific conventions. If the dynamic information is not loadable, then dynamic tags that refer to object components are represented as file offsets rather than virtual addresses.

The dynamic section is specified in the System V ABI. There are a handful of C6000-specific dynamic tags, listed in Table 35.

Table 35. C6000 Dynamic Tags

Name	Value	d_un	Executable	Shared Object
DT_C6000_GSYM_OFFSET	0x6000000D	d_val	Optional	Optional
DT_C6000_GSTR_OFFSET	0x6000000F	d_val	Optional	Optional
DT_C6000_PRELINKED	0x60000011	d_val	Optional	Optional
DT_C6000_DSBT_BASE	0x70000000	d_ptr	Mandatory (if DSBT model)	Mandatory (if DSBT model)
DT_C6000_DSBT_SIZE	0x70000001	d_val	Mandatory (if DSBT model)	Mandatory (if DSBT model)
DT_C6000_PREEMPTMAP	0x70000002	d_ptr	Optional	Optional
DT_C6000_DSBT_INDEX	0x70000003	d_val	Optional	Optional

Global Symbol Marker Tags

Symbols in the dynamic symbol table are designated as local or global. Local symbols are needed only for relocation of their containing module; they are not involved in dynamic symbol resolution, so the dynamic loader can throw them away after relocating the module. Grouping the local symbols before the global symbols in the dynamic symbol table helps the dynamic loader exploit this opportunity on bare-metal platforms. The DT_C6000_GSYM_OFFSET tag contains the offset of the first global symbol in the dynamic symbol table (.dynsym). The DT_C6000_GSTR_OFFSET tag contains the offset of the first global symbol name in the dynamic string table (.dynstr).

Local symbols may still be present after the locations marked by the tags, but there are guaranteed to be no global symbols before the marked locations.

DT_C6000_PRELINKED

This tag is used only in bare-metal load modules. It indicates that the file has had its virtual address assigned, perhaps by a prelinker or similar tool. The value represents a timestamp.

DT_C6000_PRELINKED is similar to the DT_GNU_PRELINKED tag used by the Linux prelinker, but since bare-metal prelinking is not precisely the same, a different tag is defined.

DSBT Tags

These tags are used in load modules that use the DSBT model for position independence (see Section 6.7). The DT_C6000_DSBT_BASE tag marks the statically linked location of the data segment; it corresponds to the _ _c6xabi_DSBT_BASE symbol. Since load modules are not required to contain symbol tables, the value is replicated in this tag.

The DT_C6000_DSBT_SIZE tag specifies the size reserved for the DSBT table. All load modules must have table sizes that are as least as large as the highest-numbered DSBT index among them. If a module is loaded with a too-small table or a too-large index, the loader must fail to load that module.

As described in Section 6.7, a module's DSBT index can be assigned statically by the linker or dynamically by the loader. If the load module has a statically assigned index, the DT_C6000_DSBT_INDEX tag specifies its value. No other dynamically linked module in the same process can use the same index. Modules with dynamically assignable indexes omit this tag.

DT_PREEMPTMAP

This tag contains the file offset of the preemption map for platforms that rely on static binding to pre-compute symbol preemptions.

DT_PLTGOT

This tag contains the virtual address of the Global Offset Table (GOT).

Dynamic Relocation Tags

The System V ABI defines seven dynamic tags that identify the location and type of dynamic relocations in the object file:

- DT_RELA, DT_RELASZ—These tags identify the start and size of the dynamic relocations.
- DT_PLTREL—This tag identifies the type of the relocations in the DT_JMPREL section of the table. For C6000, its value is always DT_RELA.
- DT_JMPREL, DT_PLTRELSZ—These tags identify a subrange of the DT_RELA table that contains relocations for symbols that are referred to only by PLT entries.
- DT_REL, DT_RELSZ—These tags are not used by the C6000.

The base specification is unclear on whether the dynamic relocations delineated by DT_RELA and DT_RELASZ include the PLT-specific relocations delineated by DT_JMPREL and DT_PLTRELSZ. The C6000 ABI adopts the convention that the DT_RELA table includes the DT_JMPREL table.

14.3.3 Shared Object Dependencies

Executables may depend on libraries, which may in turn depend on other libraries. These dependencies are encoded into DT_NEEDED entries in the dynamic section. When an executable or library depends on another library, the dependent library is named by a DT_NEEDED entry in the referrer's dynamic section. The dynamic linker must find the dependent library and load it as described in Section 14.2.

Some platforms, such as Linux, have a standardized search mechanism for finding dependent libraries, for example the LD_LIBRARY_PATH environment variable, as described in the System V ABI. Bare-metal platforms have no standardized guidelines. In any case, symbol resolution proceeds in the breadth-first fashion described in the System V ABI.

14.3.4 Global Offset Table

Some contexts, including libraries shared among multiple executables, require position independent addressing. To avoid encoding position-dependent addresses into the code segment, such addresses are instead generated into a table called the Global Offset Table (GOT) which is part of each static link unit's data segment. Instead of accessing the object directly, a program reads the variable's address from the GOT and addresses it the variable indirectly. The GOT is part of the data segment and is always addressed DP-relative using offsets that are fixed at static link time. It is generated by the linker in response to special GOT-generating relocations emitted by the compiler. The addresses in the GOT are patched at dynamic link time when the addresses are known.

The compiler references the GOT using special relocation entries. The static linker generates the table itself in response to the special relocations. The table entries themselves have (dynamic) relocations that the dynamic loader uses to patch in the final resolved address of the referenced object. GOT-based addressing is covered in Section 6.6. Relocations that apply to GOT entries are described in Section 13.5.1.

Executables and libraries using the bare-metal model may or may not require GOT-based addressing.

14.3.5 Procedure Linkage Table

As described in Section 6.5, the procedure linkage table (PLT) is a collection of stubs that connect calls from one load module to an imported function in another module. The address of an imported function is not known at static link time, so the static linker instead generates a position-independent stub to call the function, and patches the original call to go through the stub. The stub is relocated at load time according to the dynamically linked address of the callee.

The PLT is part of the code segment. A PLT entry may use absolute or GOT-based addressing to address the callee, depending on whether position independence is required.

14.3.6 Preemption

Preemption occurs when a symbol defined in a library is masked by a definition in an *earlier* executable or library. Earlier in this sense is according to the breadth-first ordering established by the dependence tree formed by the executable and its dependent libraries.

A symbol can be preempted only if all references to it, even from the module that defines it, use GOT-based addressing. The dynamic linker carries out the preemption by simply patching the address of the overriding symbol into the appropriate slot of the GOT.

14.3.7 Initialization and Termination

Load modules may require execution of initialization code prior to being referenced or invoked, such as C++ constructors for static objects in the module. Similarly, termination code such as destructors may be required when the module terminates.

A module specifies any required initialization and termination using the DT_INIT, DT_INIT_ARRAY, DT_PREINIT_ARRAY, DT_FINI, and DT_FINI_ARRAY entries in the dynamic section, as specified by the System V ABI.

As with initialization, the loader and/or execution environment are responsible for executing the termination functions, according to the ordering constraints imposed by module dependencies.

14.4 Bare-Metal Dynamic Linking Model

The bare-metal dynamic linking model is a platform-neutral model intended for applications that require separately linked components, but are not bound by the specific conventions of a particular operating system. Both the DSBT model and GOT-based addressing can be optionally excluded, reducing the run-time performance penalty of dynamic linking to nearly zero, at the expense of more constrained placement and addressing schemes.

In its minimal form, without DSBT and without position-independence, the model supports *dynamic linking and loading* of libraries, but does not support *sharing* of libraries between different executables. In other words, without GOT and without DSBT, the bare-metal dynamic linking model uses exactly the addressing schemes of a single statically linked bare-metal executable, resulting in significant performance advantages at the expense of flexibility.

When more flexibility is required, DSBT can be optionally enabled, allowing separately built libraries to have their own data segment. Similarly, position independence can be optionally enabled, allowing libraries to be shared among executables.

14.4.1 File Types

A program may be separately linked as an executable (file type ET_EXEC) and dependent libraries (file type ET_DYN). Under this model the files are called a *bare-metal dynamic executable* and *bare-metal dynamic libraries*, respectively. These files contain the usual dynamic information referenced through a dynamic section via the PT_DYNAMIC program header. The program and its libraries can optionally be dynamically relocated at load time.

14.4.2 ELF Identification

Executables and shared objects that conform to this model shall be identified with ELFOSABI_C6000_ELFABI in the EI_OSABI field of the ELF header. Relocatable files are identified as ELFOSABI_NONE.

14.4.3 Visibility and Binding

The default visibility for global symbols is STV_INTERNAL. That is, symbols that are imported or exported must be explicitly declared as such. Symbol preemption is not supported. The one definition rule is not honored for symbols with vague linkage (vtbls, rtti type info etc) across shared objects. The bare-metal model uses forced static binding. That is, the linker forces that imported references are bound to their definitions during static linking.

In the dynamic symbol table all symbols with STV_DEFAULT visibility are marked STB_GLOBAL. That is, weak symbols are converted to global symbols if they have default visibility. This is to simplify the loader implementation.

14.4.4 Data Addressing

Use of the DSBT model is optional under the bare-metal dynamic linking model. Without DSBT, a program has a single DP which points to the data segment base address (first DP-relative segment) of the executable. The executable itself can use near DP-relative addressing to refer to its own data. Data in libraries must be addressed using *far* addressing modes (either far DP-relative or absolute). This applies both to an executable addressing imported data, and to a library addressing its own data (since the DP belongs to the executable). Without a DSBT, a library cannot have .bss, .neardata, or .rodata sections.

With DSBT enabled, each separately built component can have its own DP-relative segment(s).

Position-independent data via GOT-based addressing is also optional in the bare-metal dynamic linking model. Without GOT-based addressing, references to imported addresses are encoded into the code segment, either as absolute addresses, or, optionally for non-DSBT executables, as offsets from the executable's DP. Such code cannot access separate preprocess copies of libraries' data segments, so although separately linked libraries are supported, shared libraries are not. Code compiled without position independence is likely to require load-time fixups.

The linker shall enforce consistent use of the DSBT and GOT models.

14.4.5 Code Addressing

Calls to imported functions go through a PLT entry that can be generated by either the compiler or the static linker. Lazy binding is not supported. The PLT can use absolute, PC-relative, or GOT-based addressing to address the function, depending on the degree of position independence required.

14.4.6 Dynamic Information

Dynamic tags use file offsets (rather than virtual addresses as specified by the System V ABI) to reference dynamic information. Dynamic segments are not part of the load image of the program—that is, the PT_DYNAMIC and related sections are not contained within any PT_LOAD segment.

Table 36 summarizes the characteristics of the bare-metal dynamic linking model and compares the two bare-metal file types.

Table 36. Bare-Metal Dynamic Linking Files

Characteristic	Bare-Metal Dynamic Executable	Bare-Metal Dynamic Library
ELF file type (e_type)	ET_EXEC	ET_DYN
ELF identification (e_ident)	ELFOSABI_C6X_ELFABI	
Dynamic sections loadable	No	
Addressing own data	Can have .bss, .neardata, and .rodata, and access them using near DP-relative addressing	With DSBT: same as executable Without DSBT: Far (DP-relative, absolute or GOT)
Addressing imported data	Far (DP-relative, absolute, or GOT)	
Has PT_DYNAMIC segment	Yes	
Has PT_INTERP	No	
Can import/export symbols	Yes, with explicit directives	
Relocatable at load time	Optionally	Yes
Entry Point	Mandatory	Option

15 Linux ABI

This section specifies conventions for addressing, dynamic linking, and program loading for C6000 Linux-based systems. Our intention is to follow the conventions used by other embedded MMU-less Linux systems as much as is practical.

This part of the ABI is based on Chapter 5 of the System V ABI standard (http://www.sco.com/developers/gabi/2003-12-17/contents.html), which describes object file information and system actions that create running programs. This section, along with rest of this ABI, forms the processor-specific supplement to that standard specifically for programs running under Linux on the C6000.

These conventions apply to user-space application programs. The kernel is independent and may follow implementation-specific guidelines.

15.1 File Types

A program may be separately linked such that it is comprised of an executable (file type ET_EXEC) and shared libraries (file type ET_DYN). These files contain the usual dynamic information referenced through a dynamic section via the PT_DYNAMIC program header. The program and its libraries can optionally be dynamically relocated at load time.

Shared libraries are required to be position-independent. Executables may or may not be position independent. Position dependent executables require relocations to the code segment(s) at load time and subject to inefficient use of system resources.

15.2 ELF Identification

Executables and shared objects that conform to the Linux ABI shall be identified with ELFOSABI_C6000_LINUX in the EI_OSABI field of the ELF header. Relocatable files are identified as ELFOSABI_NONE.

The rationale for specifying the vendor-specific ELFOSABI_C6000_LINUX value instead of ELFOSABI_LINUX is to differentiate this ABI variant, corresponding to MMU-less Linux (uClinux), from a potential future MMU-enabled variant.

15.3 Program Headers and Segments

The following are the program headers and segments.

p_align

As described in the System V ABI, loadable segments are aligned in the file such that their p_vaddr (address in memory) and p_offset (offset in the file) are congruent, modulo p_align. For the Linux ABI, p_align is specified to be 0x1000.

PT_INTERP Segment

The PT_INTERP segment contains the name of the object file containing the dynamic loader. For ELF executable files as described in this document, the interpreter is typically ld.so.

Read-Only Segments

Shared objects and executables must have a PT_LOAD segment with Read+Execute permission that contains the module's program code and shareable constants. A shareable constant is any object that is not writable and whose value does not consist of an address. This segment also includes the ELF structures needed to load and execute the program, including the file header, PT_INTERP, PT_PHDR, PT_DYNAMIC, PT_NOTE (if present) and PT_PHATTR (if present) segments.

Position-dependent executables may have additional read-only or Read+Execute segments with unspecified contents. If there are multiple such segments, it is not permitted to have PC-relative references between them. If the loader relocates them, it is not required to preserve their position relative to each other.

Data Segments

Shared objects and executables must have a single PT_LOAD segment with Read+Write permission that contains the module's DSBT, GOT, and read-write data. This segment is addressed using DP-relative addressing and is therefore marked with the PF_C6000_DPREL flag.

ELF requires uninitialized data in a segment to follow all the initialized data. However, if the DP-relative segment contains both uninitialized near data (e.g. .bss) and initialized far data (e.g. .fardata), the uninitialized data may need to precede the initialized data to be within range of the DP. In this case the linker is required to fill the uninitialized portion of the segment with 0.

Shared objects and executables may have additional Read+Write segments. For position independence, these sections must be addressed exclusively using GOT-based addressing. Position-dependent executables may use absolute addressing.

Stack Segment

The C6000 Linux ABI follows common convention by defining an additional segment type that enables toolchains to specify the minimum stack allocation for an executable.

Name	Value	Comment
PT_GNU_STACK	0x6474E551	Stack size and permission

The p_flags member specifies the permissions on the segment containing the stack and is used to indicate whether the stack should be executable.

In the absence of this header, the size and permission of the stack remains unspecified.

Bound and Read-Only Segments

Linux executables and shared objects shall not contain segments marked as bound or read-only as described in Section 14.1.3.

15.4 Data Addressing

Shared libraries are required to be fully position independent. That is, there are no load-time relocations to the read-only segment. Any object with visibility STV_DEFAULT must be addressed through the GOT. All other static data must be addressed DP-relative.

Executables can be optionally built to be position independent. Executables may use the import-as-own preemption mechanism described in Section 15.9 to avoid using GOT-based addressing for STV_DEFAULT variables.

Position-dependent executables may use a combination of position-independent addressing and position-dependent (absolute) addressing. Executables that use absolute addressing are subject to load-time relocation.

15.4.1 Data Segment Base Table (DSBT)

Linux executables and shared objects must conform to the DSBT model as described in Section 6.7. The near-DP segment must contain a DSBT table that has at least as many entries as the largest DSBT index among all the modules comprising the program. For consistency among library vendors, the ABI standardizes the default DSBT table size to be 64 entries. The presence and size of the DSBT table is indicated by the C6000-specific dynamic tags specified in Section 14.3.2.

DSBT index 0 is reserved for the executable. DSBT index 1 is reserved for the program interpreter. Libraries are statically assigned unique DSBT indexes starting with 2. To satisfy the convention of position independence, dynamic re-assignment of DSBT indexes is not supported.

15.4.2 Global Offset Table (GOT)

As described in Section 6.6, position independence for code and data is achieved through the Global Offset Table (GOT). The GOT is part of the data segment and is always addressed DP-relative using offsets that are fixed at static link time. The GOT consists of 4-byte *slots* that contain dynamically-assigned addresses. A Linux executable or shared object must have a GOT with at least two slots (8 bytes). GOT entries are marked with dynamic relocations that reference dynamic symbols. GOT entries are initialized by the static linker as follows:

- A GOT entry marked with an R_C6000_JUMP_SLOT relocation is initialized with the address of the lazy binding resolver stub, as described in Section 15.6.
- All other GOT entries are initialized to zero.

The static linker must reserve the first two slots in the in the GOT for use by the lazy binder. See Section 15.6.

15.5 Code Addressing

For calls to imported or potentially imported functions, the compiler or linker generates a stub called a Procedure Linkage Table Entry as described in Section 6.5.

Calls that require patching through a PLT are marked by relocation types that meet all of following conditions:

- The relocation type is R_C6000_PCR21.
- The visibility of the referenced symbols is STV_DEFAULT.
- The type of the referenced symbol is STT_FUNC or STT_NONE.

In an executable, an additional condition applies:

- The symbol is undefined in the static link unit containing the call.

A PLT entry in a shared object or position-independent executable must use position-independent (GOT-based) addressing to address the callee. In this case, PLT entries must follow the lazy binding convention as described in Section 15.6. That is, the first instruction of the PLT must load the byte offset of the R_C6000_JUMP_SLOT relocation entry that marks the callee's GOT entry into B0.

A PLT entry in a position-dependent executable may use absolute addressing. The C6000 does not adopt the convention common to other architectures in which a reference to a function's address can be statically resolved to the PLT entry. See Section 6.7.3.

15.6 Lazy Binding

For large programs, load-time symbol resolution can significantly degrade program startup time. Lazy binding is a mechanism that delays resolution of function symbols until they are actually called by the program, thus reducing startup time and improving overall performance since only functions that are actually called need to be resolved.

The general approach is that the first call through a PLT vectors control through a resolver function in the dynamic linker, which performs the resolution and re-routes future calls directly to the function itself.

The resolver requires two arguments. The first is a module id that identifies the current module (the one containing the reference). The representation of the module id is unspecified by the ABI, to be determined by the loader. The second argument specifies the relocation entry corresponding to the target function. The relocation entry in turn provides both the name of the target symbol, and the location of the reference in the GOT. The relocation entry is specified by its byte offset in the object file from the address in the DT_RELPLT tag in the file's .dynamic section.

Since all this happens behind the caller's back, the mechanism must preserve any state that affects the standard function-call interface. In particular, it must not disturb any registers used for argument passing or the return address register, and it must preserve any callee-saved registers it modifies. To avoid disturbing the normal argument registers, the resolver's two arguments are passed in B0 and B1.

Two slots in the Global Offset Table are reserved for use by the dynamic loader to implement lazy binding. GOT[0] is used by the loader to hold the address of the resolver function. GOT[1] is used to hold the module id.

The following sequence describes the mechanism:

1. The static linker identifies candidates for lazy binding. A candidate is a GOT entry that is only referred to by a PLT entry; that is, only used for calling an imported function.

2. The static linker generates, or includes from a library, a special *resolver stub*. In this description the stub is called PLT0, although the ABI does not specify its name or location.

3. The static linker initializes candidate GOT entries with the address of PLT0, and marks them with R_C6000_JUMP_SLOT relocations. The linker locates any such relocation in the section of the dynamic relocation table marked with the DT_JMPREL tag.

4. The PLT entry is generated with an additional instruction for use in lazy resolution. The instruction loads register B0 with the first of the resolver's two arguments: the byte offset of the R_C6000_JUMP_SLOT relocation entry relative to the dynamic relocation table, indicated by the DT_REL[A] tag. It then loads the target address from the GOT in the usual way and jumps to it. As a result of the initialization in step 3, the first time this jump happens, control transfers to PLT0.

5. PLT0 loads the second of the resolver's two arguments from GOT[1] into B1: a loader-defined value that identifies the current module. It then loads the address of the loader's resolver function from GOT[0] and tail-calls it.

6. The resolver function uses its two arguments to find the specified dynamic relocation in the object file specified by the module id. It looks up the symbol in the dynamic symbol table to get the actual address of the function, and replaces the GOT entry with that address. Finally, it jumps to that address, effectively tail-calling the target function.

7. When the PLT entry is entered for subsequent calls, the GOT has been updated with the actual address, so control passes directly to the function.

Lazy Binding PLT Entry

```
$sym$plt:
      MVK    reloc_offset(sym),B0    ;byte offset of GOT reloc entry from DT_RELPLT
      MVKH   reloc_offset(sym),B0
      LDW    *+DP($GOT(sym)),tmp    ;&PLT0 first time, &sym after that
      B      tmp
```

Resolver Stub—PLT0

```
PLT0:
      LDW    *+DP($GOT(0)),tmp    ; address of resolver
      LDW    *+DP($GOT(4)),B1     ; module id
      B tmp                       ; tail-call resolver
```

Global Offset Table

```
               ; $GOT(0) reserved, initialized to module id
               ; $GOT(4) reserved, initialized to &resolver function
               ; ...
               ; $GOT(sym) R_C6000_JUMP_SLOT initialized to &PLT0
               ; updated to &sym by resolver
```

15.7 Visibility

Under Linux, the default visibility for global symbols is STV_DEFAULT. In a shared object, defined symbols with STV_DEFAULT visibility are subject to preemption and must be addressed as if they were imported. In an executable, the import-as-own convention (see Section 15.9) allows defined and undefined variables with STV_DEFAULT visibility to be addressed as it they were STV_INTERNAL; that is, using DP-relative addressing.

Toolchains may implement vendor-specific options or extensions that alter the default visibility rules. These must be reflected using standard values for the visibility flags in the affected symbol(s).

15.8 Preemption

Linux adopts the convention that with respect to symbol resolution, dynamic linking preserves the behavior of static linking. Preemption occurs when there are multiple definitions of the same symbol: specifically, a symbol defined in a library is masked by a definition in an *earlier* executable or library. Earlier in this sense is according to the breadth-first ordering established by the dependence tree formed by the executable and its dependent libraries.

A symbol can be preempted only if all references to it, even those in the module that defines it, use GOT-based addressing. The dynamic linker carries out the preemption by simply patching the address of the overriding symbol into the appropriate slot of the GOT.

15.9 Import-as-Own Preemption

In Linux, external symbols generally have STV_DEFAULT visibility, and are therefore subject to preemption unless declared otherwise. This would normally result in GOT-based addressing for almost all references to variables, including those defined in the same module. In other words, Linux modules are required to treat all references to extern variables as if they were imported, even if they are not. To avoid the resultant performance penalty, executables employ a special convention that allows them to evade it.

An executable may choose to treat any reference to a variable as if as it was its own—that is, defined in the executable—allowing the compiler to generate efficient DP-relative addressing. At static link time, any variable that turns out to be imported is given a duplicate definition in the executable. At dynamic load time, the duplicate definition preempts the original definition in the library, and any initializer is copied from the preempted definition to the new definition.

The size of the duplicate definition is specified by the st_size field from the source definition. The minimum alignment of the duplicate definition is given as follows:

- Let *max* be the maximum possible alignment required for an object of the given size defined in the source module. This value can be determined as a function of the object's size, the alignment requirements of Section 2, and the alignment specified by the TAG_ABI_array_object_alignment build attribute.
- Let *vaddr* be the virtual address of the object in the source module.
- The alignment of the duplicate object is the greatest common divisor of *vaddr* and *max*.

(Intuitively, this defines the duplicate object to be at least as well aligned as the original object, up to its maximum possible required alignment.)

Any initial value stored in the original symbol when the process image was created must be propagated to the duplicate. The R_C6000_COPY relocation serves this purpose. The linker marks the duplicate definition in the executable with R_C6000_COPY. At load time, the dynamic loader finds the referenced symbol in the library and copies the data at that location to the duplicate definition in the executable.

In this way the executable is not penalized for dynamic linking. Instead, the penalty is borne by the library, which must assume that all its extern variables are imported—which, because of preemption, it would have to do anyway.

15.10 Program Loading

The Linux kernel begins the process of loading a program by copying or mapping both its load segments and those of the interpreter program specified by its PT_INTERP header into memory. The kernel then jumps to an entry point in the interpreter, which completes the loading process. For ELF executables the interpreter is usually the dynamic loader, ld.so.

The first time the interpreter is invoked, it must bootstrap itself by processing its own dynamic relocations. It must then load dependent libraries, perform any dynamic symbol resolution, and process the dynamic relocations of the program itself.

The kernel communicates startup information to the interpreter via an initialized data structure called the load map, declared as follows:

Example 1. Program Load Map Data Structure

```
struct elf32_dsbt_loadmap
{
    /* Protocol version number, must be zero. */
    Elf32_Word version;

    /* Pointer to DSBT */
    unsigned   *dsbt_table;
    unsigned    dsbt_size;
    unsigned    dsbt_index;

    /* Number of segments */
    Elf32_Word  nsegs;

    /* The actual memory map. */
    struct elf32_dsbt_loadseg segs[nsegs];
};

struct elf32_dsbt_loadseg
{
    /* Core address to which the segment is mapped. */
    Elf32_Addr addr;

    /* Virtual address recorded in the program header. */
    Elf32_Addr p_vaddr;

    /* Size of this segment in memory. */
    Elf32_Word p_memsz;
};
```

The kernel invokes the kernel with the 4 arguments in registers and the rest on the stack. The register arguments are:

B4	address of the executable's load map
A6	address of the interpreter's load map
B6	address of the interpreter's dynamic section
B14 (DP)	_ _c6xabi_DSBT_BASE for the interpreter

The kernel allocates a stack for the process and initializes SP. The initial contents of the stack provide the program's command-line arguments and environment variables:

SP ▶	
SP+4	alignment padding
SP+8	**argc**
SP+12	**argv[0]**
	…
	argv[argc-1]
SP+12+(4*argc)	**NULL**
SP+16+(4*argc)	**envp[0]**
	…
	NULL
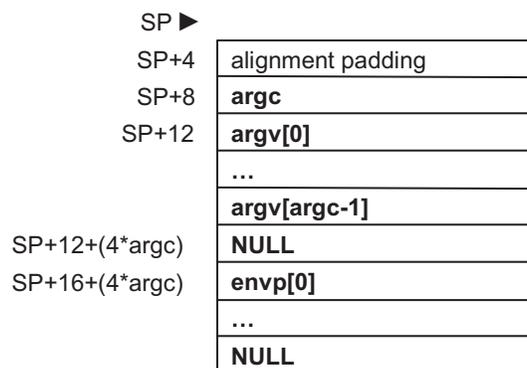
Figure 11. Initial Contents of Stack for Example 1

The kernel then jumps to the entry point of the interpreter, labeled with the symbol _start.

15.11 Dynamic Information

The dynamic segment contains information related to program loading and dynamic linking. It is specified by the System V ABI. The value and meaning of C6000-specific dynamic tags is specified in Section 14.3.2. Linux modules do not contain the global symbol marker tags DT_C6000_GSYM_OFFSET and DT_C6000_GSTR_OFFSET.

Under the Linux ABI, all dynamic linking metadata is part of the load image of the program—that is, the PT_DYNAMIC segment and related sections are contained within a read-only PT_LOAD segment. Consequently, dynamic tags with address values (d_ptr) are represented as virtual addresses rather than file offsets as in the bare-metal ABI.

15.12 Initialization and Termination Functions

The System V ABI specifies an initialization sequence for executables and shared objects through which functions such as constructors for global objects can be called prior to calling main. Similarly, there is a mechanism for defining functions to be called after main returns. These mechanisms use tables of function pointers marked by DT_INIT* and DT_FINI* dynamic tags.

Section 3.3.5 of the GC++ ABI augments the termination mechanism to enable C++ programs to properly register destructors to be called when a shared object is unloaded before the program that uses it terminates. The mechanism uses an API function in the C++ compiler support library called _ _cxa_atexit, which is called as follows:

```
__cxa_atexit(dtor, obj, &__dso_handle);
```

(Here dtor is a pointer to the destructor function and obj is a pointer to the object.)

The third argument, _ _dso_handle, is a unique address that identifies the shared object. The C6000 ABI defines its value to be the address of the module's near-DP segment.

Another function, _ _cxa_finalize, implements calls to the registered functions when the shared object is unloaded. This function is called as follows:

```
__cxa_finalize(&__dso_handle);
```

The linker must arrange for this call to occur as the first termination action, typically via the DT_FINI* table. Since _ _cxa_finalize has an argument, and DT_FINI functions are called without arguments, the linker must generate a nullary wrapper function for the call.

To summarize the requirements for this convention, the static linker is responsible for:
- Generating the hidden symbol _ _dso_handle with the address of the near-DP segment.
- Generating a wrapper function with no arguments that calls _ _cxa_finalize as shown previously.
- Registering the wrapper function as the first call in the termination function list marked with the DT_FINI or DT_FINI_ARRAY dynamic tag.

These requirements apply when generating any executable or shared object containing a call to _ _cxa_atexit.

15.13 Summary of the Linux Model

Table 37. Linux Program Files

Characteristic	Position-Dependent Executable	Position-Independent Executable	Shared Object
ELF file type (e_type)	ET_EXEC		ET_DYN
ELF identification (e_ident)	ELFOSABI_C6X_LINUX		
Read-only segments	Multiple Allowed	One	
DP-relative data segment	One		
Other read-write segments	Absolute	GOT only	
Code addressing	PC-relative or absolute	PC-relative or GOT	
Addressing of own hidden data	DP-relative or absolute	DP-relative	

Table 37. Linux Program Files (continued)

Characteristic	Position-Dependent Executable	Position-Independent Executable	Shared Object
Addressing of imported STV_DEFAULT data	Far (DP-relative, absolute, or GOT)	DP-relative or GOT	GOT
DSBT model	Required		
Requires load-time relocations to read-only segments	Yes	No	
Default visibility of extern symbols	STV_DEFAULT		
Load segments include metadata (PT_INTERP, PT_PHDR, PT_DYNAMIC)	Yes		

16   Symbol Versioning

Symbol versioning provides a mechanism to support multiple versions of symbols in shared libraries and to insure compatibility among dynamically-linked components. The C6000 implementation is based on the one used in the GNU toolchain, which was in turn adapted from Sun Microsystems. The reference document for GNU's symbol versioning support is the paper by Ulrich Drepper at http://people.redhat.com/drepper/symbol-versioning. As far as we know there are no C6000-specific additions or deviations. The description in this document summarizes the mechanism for reference.

An executable file using symbol versioning shall set EI_OSABI field in the ELF header to appropriate operating-system specific value.

16.1   ELF Symbol Versioning Overview

GNU symbol versioning allows a user to specify a version name for a symbol exported from a DSO. This allows more than one version of the same symbol definition in a DSO. Exactly one of them is marked the default. When linked against this symbol definition, the default version is always used to bind the symbol references.

For example, assume a library implementer defines an API function api_do_encode in codec_1_0.dso. Initially there is only one version, say VER1. When an application links against this DSO, all the references to api_do_encode are resolved by VER1 of api_do_encode. Later the implementer enhances the API by adding an updated, but incompatible, version of api_do_encode, but still wants to support previously built applications using the older API. The implementer can create a new codec_2_0.dso with both the original VER1 api_do_encode and a new VER2 definition of the same symbol, which now becomes the designated default version. When a new application links against codec_2_0.dso, references to api_do_encode are resolved by VER2 api_do_encode. The original VER1 api_do_encode is still available to satisfy references from older applications built against codec_1_0.dso.

Please refer to Drepper's paper for details on the mechanics to specify the symbol versions.

GNU symbol versioning information is recorded in the three ELF sections:

*   **Version Definition Section**

    This section defines version names associated with symbols exported from this executable file. The version of the file is also defined in this section.

    This section can be located via the DT_VERDEF tag entry in the dynamic section. The tag DT_VERDEFNUM contains the number of version definitions this section contains. The version definition section has the section type SHT_TI_verdef. Note that this section type value 0x6FFFFFFD is the same as SHT_GNU_verdef. This specification recommends the name .gnu.version_d for this section. However, only the section type should be used to identify this section; the name should not be used.

*   **Version Needed Section**

    This section records the versions needed by undefined symbols references in this executable file. Each entry names a DSO and points to a list of versions needed from it. When the dynamic linker loads an executable, it will find and load all the DSOs needed. Before making such DSOs public, the dynamic linker will first check if the version needed by the executable is satisfied by this DSO's version definitions. This version needed information is recorded by the static linker when it binds references to definitions from DSOs.

*   **Version Section**

    This section extends the dynamic symbol table by adding the version number to the dynamic symbol entries. This section contains the same number of entries as the dynamic symbol table. The symbol id is used to index this table of version numbers. If the symbol is undefined, the version number matches a version needed entry in the version needed section. If the symbol is defined, the version number matches a version definition entry in the version definition section. The version definition is *default* when bit 15 is clear.

ELF provides a mechanism to locate and identify these symbol version sections in an ELF executable. These sections are located by the dynamic tags from the dynamic section and are identified using special section types.

For example, the version definition section is located by the dynamic tag DT_VERDEF. The DT_VERDEFNUM tag contains the number of version definitions in the version definition section. This section shall have the section type SHT_GNU_verdef (0x6FFFFFFD). The name of this section is nominally .gnu.version_d, but implementations should rely on the section type rather than the name.

16.2 Version Section Identification

Table 38 lists the tags, section type and section names the three symbol version elf section types associated with symbol versioning.

Table 38. Version Section Identification

ELF Sections	Dynamic Tags	Section Type	Section Name
Version Definition	DT_VERDEF (0x6FFFFFFC) DT_VERDEFNUM (0x6FFFFFFD)	SHT_GNU_verdef (0x6FFFFFFD)	.gnu.version_d
Version Needed	DT_VERNEED (0x6FFFFFFE) DT_VERNEEDNUM (0x6FFFFFFF)	SHT_GNU_verneed (0x6FFFFFFE)	.gnu.version_r
Version	DT_VERSYM (0x6FFFFFF0)	SHT_GNU_versym (0x6FFFFFFF)	.gnu.versym

17   Build Attributes

The ABI specification for the ARM ABIv2 specification defines the build attributes mechanism to capture the build time options so that a linker can enforce compatibility of relocatable files. The C6x ELF specification uses the same structure to encode the build attributes as documented in the ARM ABIv2 build attributes specifications in "ARM Addenda" to, and "Errata" in, the *ABI for the ARM Architecture*, document number ARM IHI0045A released on 13th November 2007.

Build attributes are classified as vendor-specific or ABI-specific. The 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_C6000_ATTRIBUTES (0x70000003), conventionally named c6xabi.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' (0x41), followed by a sequence of vendor subsections.

'A'	vendor subsection	vendor subsection	...
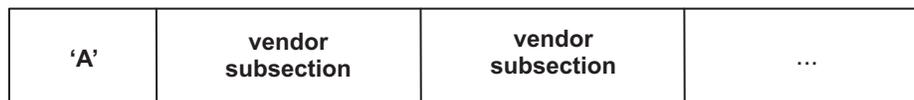
Each subsection has the following format:

length	vendor name	0	vendor data
uint32	char[ ]	uint8	
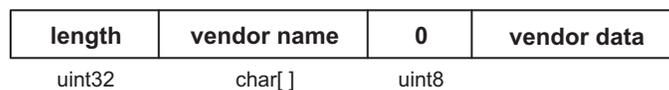
The length field specifies the length in bytes of the entire subsection. The vendor name "c6xabi" is reserved for ABI-specified attributes. The format and interpretation of vendor data in other subsections is vendor-specific.

17.1   C6000 ABI Build Attribute Subsection

Attributes that are specified by this ABI are recorded in the subsection with the vendor string c6xabi. 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 c6xabi 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	section numbers	0	attributes	Apply to specified sections
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 "", 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 do not 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.

## 17.2 C6000 Build Attribute Tags

Tag_ISA (=4), ULEB128

This tag specifies the C6000 ISA(s) that can execute the instructions encoded in the file. The following values are defined:

0	No ISA specified
1	C62x
2	Reserved
3	C67x
4	C67x+
5	Reserved
6	C64x
7	C64x+
8	C6740
9	Tesla
10	C6600

This tag determines object compatibility as follows. Here, the transitive relation A < B means that B is compatible with A; that is, B can execute code generated for either A or B. When combining attributes, the *greatest* ISA that can execute both (B in this case) should be used.

- Tesla is not compatible with any other ISA revisions.
- C62x < all ISAs except Tesla
- C67x < C67x+
- C67x+ < C6740
- C64x < C64x+
- C64x+ < C6740
- C6740 < C6600

C6000 ISA compatibility is illustrated by the following directed graph in which an edge A→B represents the compatibility relation A < B.
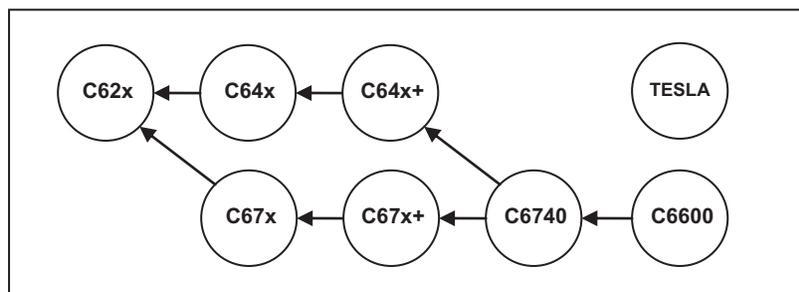
Figure 12. C6000 ISA Compatibility Graph

Tag_ABI_wchar_t, (=6), ULEB128

    0   wchar_t is not used.

    1   The size of wchar_t is 2 bytes.

    2   The size of wchar_t is 4 bytes.

Section 2.1 specifies wchar_t as unsigned int. However, in some circumstances the TI toolchain defines wchar_t as unsigned short. This tag enables detection of any incompatibility resulting from this violation.

Tag_ABI_stack_align_needed, (=8), ULEB128

    0   Code requires 8-byte stack alignment at function boundaries.

    1   Code requires 16-byte stack alignment at function boundaries.

Tag_ABI_stack_align_preserved, (=10), ULEB128

    0   Code requires 8-byte stack alignment at function boundaries.

    1   Code requires 16-byte stack alignment at function boundaries.

All currently supported ISAs use 8-byte stack alignment. 16-byte alignment is anticipated for future ISAs.

Code that requires 16-byte stack alignment is not compatible with code that only preserves 8-bute alignment. When merging tags, the result should reflect the smallest alignment given by TAG_ABI_stack_align_preserved, and the largest alignment given by TAG_ABI_stack_align_needed.

Tag_ABI_DSBT, (=12), ULEB128

    0   DSBT addressing is not used.

    1   DSBT addressing is used.

Tag_ABI_PID, (=14), ULEB128

    0   Data addressing is position dependent.

    1   Data addressing is position independent; GOT is accessed using near DP addressing.

    2   Data addressing is position independent; GOT is accessed using far DP addressing.

An object file with a non-zero Tag_ABI_PID uses no absolute addressing for data. All data is addressed using either DP-relative, GOT, or in the case of read-only constants, PC-relative addressing. Such an object can have the location of its DP-relative data segment assigned dynamically, without requiring relocation, as required by a shared object.

A value of 2 indicates that the object relies on far GOT-based addressing (see Section 6.6). That is, the GOT itself is far.

Tag_ABI_PIC, (=16), ULEB128

    0   Addressing conventions are unsuitable for a shared object.

    1   Addressing conventions are 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.

Tag_ABI_array_object_alignment, (=18), ULEB128

      0   Array variables are aligned on 8 byte boundaries.

      1   Array variables are aligned on 4-byte boundaries.

      2   Array variables are aligned on 16-byte boundaries.

Tag_ABI_array_object_align_expected,(=20), ULEB128

      0   Code assumes 8-byte alignment for array variables.

      1   Code assumes 4-byte alignment for array variables.

      2   Code assumes 16-byte alignment for array variables.

The preceding two tags apply to array variables with external visibility, as discussed in Section 2.6. For compatibility, the alignment value indicated by the TAB_ABI_array_align_expected tag must be less than or equal to the alignment value indicated by the TAG_ABI_array_object_alignment tag. When merging tags, the result should reflect the smallest alignment given by TAG_ABI_array_object_alignment, and the largest alignment given by TAG_ABI_array_object_align_expected.

Tag_ABI_compatibility, (=32), ULEB128, char[]

This tag enables vendors to arrange specific compatibility conventions beyond the scope of the ABI. It has two operands, a ULEB128 flag and a NULL-terminated string. The string specifies the name of the extra-ABI convention, as defined by the arranging vendor. The flag characterizes the object with respect to the convention. In the following description, the term *ABI-compatible* means compliant with this ABI, and compatible according to the conditions set forth in this document, such as other build attribute tags. The flag values are:

      0   The object has no toolchain-specific compatibility requirements, and is therefore compatible with any other ABI-compatible object.

      1   The object is compatible with other ABI-compatible objects provided that it is processed by a toolchain that complies with the named convention (for example, if the convention names a vendor, that vendor's toolchain).

     N>1   The object is not compatible with the ABI, but may be compatible with other objects under the named convention. In this case the interpretation of the flag is defined by the convention.

Note that the string identifies the extra-ABI convention, not necessarily the toolchain that produced the file.

If the ABI compatibility tag is omitted, it as the same meaning as a tag with flag value 0 (no additional compatibility requirements).

Tag_ABI_conformance, (=67), char[]

This tag specifies the version of the ABI to which the object conforms. The tag value is a NULL-terminated string containing the ABI version. The version specified in this standard is "1.0". Digits following the decimal point are informational only and do not affect compatibility checking.

To simplify recognition by consumers for the common case in which the while file conforms to the ABI, this tag should be the first attribute in the first attribute vector in the c6xabi subsection.

Table 39 summarizes the build attribute tags defined by the ABI.

Table 39. C6000 ABI Build Attribute Tags

Tag	Tag Value	Parameter Type	Compatibility Rules
Tag_File	1	uint32	
Tag_Section	2	uint32	
Tag_Symbol	3	uint32	
Tag_ISA	4	ULEB128	See previous description
Tag_ABI_wchar_t	6	ULEB128	If not zero, must match exactly

Table 39. C6000 ABI Build Attribute Tags (continued)

Tag	Tag Value	Parameter Type	Compatibility Rules
Tag_ABI_stack_align_needed	8	ULEB128	Must be compatible with Tag_ABI_stack_align_preserved. Combine using max value.
Tag_ABI_stack_align_preserved	10	ULEB128	Must be compatible with Tag_ABI_stack_align_needed. Combine using min value.
Tag_ABI_DSBT	12	ULEB128	Exact
Tag_ABI_PID	14	ULEB128	Warn if different; combine using min value.
Tag_ABI_PIC	16	ULEB128	Warn if absent when building shared library; combine using min value.
TAG_ABI_array_object_alignment	18	ULEB128	Must be at least alignment from TAG_ABI_array_object_align_expected. Combine using max alignment.
TAG_ABI_array_object_align_expected	20	ULEB128	Must be <= alignment from TAG_ABI_array_object_alignment. Combine using min alignment.
Tag_ABI_compatibility	32	ULEB128 char[]	See description in text.
Tag_ABI_conformance	67	char[]	Unspecified

18   Copy Tables and Variable Initialization

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 run-time 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 section 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 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 *removes* the data from .mysect, making it an uninitialized section, and assigns its address as its run location. It creates a new initialized section called .mydata.load1 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 run-time library to decode and copy the data at run time, 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 run time, 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 .ovly [1] section.

---

[1]   Section names for copy table sections and compressed source data are arbitrarily chosen by the linker.
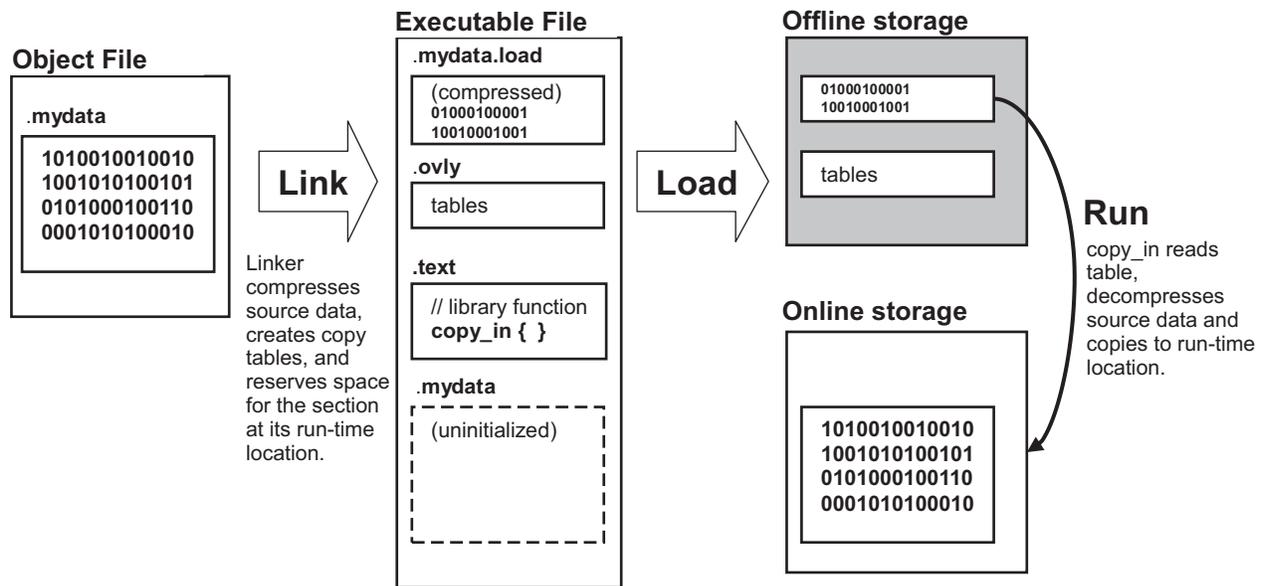
Figure 13. Copy Table Overview

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 18.3.

- **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.

18.1 Copy Table Format

A copy table has the following format:

```
typedef struct
{
    uint16     rec_size;
    uint16     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
{
    uint32  load_addr;
    uint32  run_addr;
    uint32  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 the size is non-zero, the source data is the exact image of the data to copy; in other words, it is not compressed. The copy-in operation is to simply copy *size* bytes from the load address to the run address.

If the 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:
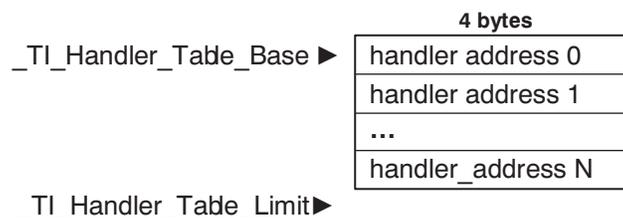
Figure 14. Handler Table Format

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.

Example 2 provides a reference implementation of the copy_in function:

Example 2. Reference Implementation of 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);
      }
   }
}
```

18.2 Compressed Data Formats

Abstractly, compressed source data has the following format:

handler index	compressed data
1 byte	length is format-specific
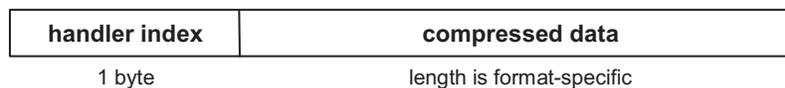
Figure 15. Compressed Source Data Format

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).

18.2.1 RLE

The data following the 8-bit index is compressed using run length encoded (RLE) format. The C6000 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. Go to step 2.

6. If L = 4 read the next byte (B'). Copy B' to the output buffer L times. 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.

18.2.2 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.

18.3 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 in Section 18. The initialization mechanism is conceptually similar to copy tables, but differs slightly in the details.

Figure 16 depicts the conceptual operation of variable initialization under the ROM model. In this model, the linker *removes* the data from sections that contain initialized variables. The sections become uninitialized sections, allocated into RAM at their run-time 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 run-time library decodes and copies it to its run address.
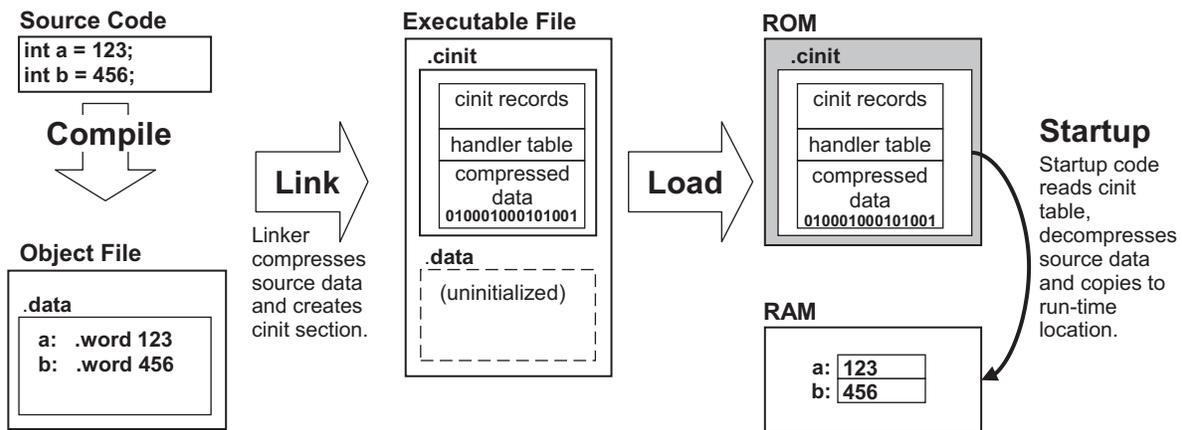
Figure 16. ROM-Based Variable Initialization Via cinit

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**, consisting of cinit records, which are similar to copy records.
- The **handler table**, consisting of pointers to decompression routines, as described in Section 18.1. The handler table and handlers are shared by initialization and copy tables.
- The **source data**, consisting of compressed or uncompressed data used to initialize variables.

These items may be in any order.

Figure 17 is a schematic depiction of the .cinit section.
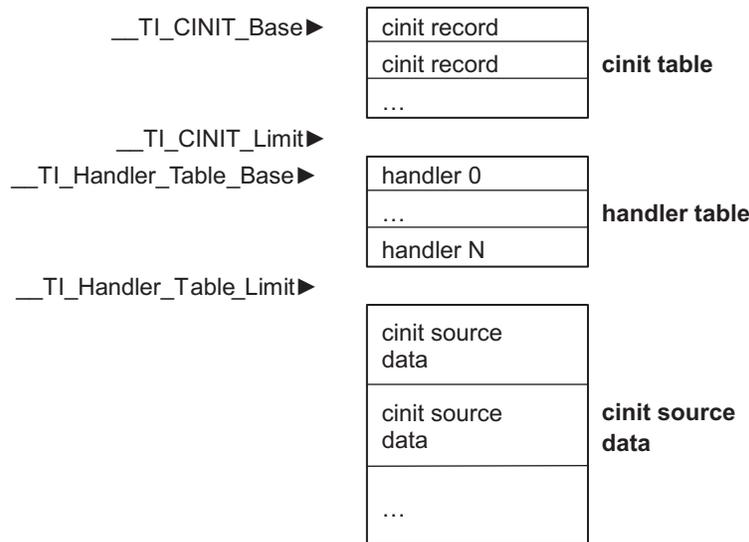
Figure 17. The .cinit Section

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
{
    uint32  source_data;
    uint32  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 (see Section 18.1), 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 includes 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 run-time 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 run-time 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.

19 Extended Program Header Attributes

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: up to eight OS-specific flags and four processor-specific flags are permitted.

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 C6000 EABI conforming tool can choose to implement support for extended program header attributes as a quality-of-implentation issue. Support for extended program header attributes is not required to be C6000 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_TI_PHATTRS (0x70000000).

19.1 Encoding

The program header attributes are encoded in a triplet, which can be represented as shown here:

```
typedef struct
{
    Elf32_Half pha_seg_id;       /* Segment id */
    Elf32_Half pha_tag_id;       /* Attribute kind id */

    Union
    {
        Elf32_Off pha_offset;    /* byte offset within the .TI.phattrs section */
        Elf32_Word pha_value;    /* Constant tag value */
    } pha_un;

} Elf32_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 fields in the union pha_un 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) or a null terminated byte string (NTBS). The fixed-length binary data size should be 32-bits 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_un is interpreted. Each attribute has pre-defined behavior. The pha_un field can be interpreted as pha_value or pha_offset, or may be unused. If pha_offset is used, the value points to either NTBS or FLBD. If pha_offset is interpreted as FLBD, the length of the field shall be pre-defined.

19.2 Attribute Tag Definitions

Texas Instruments has introduced two attributes in support of native ROMing support.

Table 40. ROMing Support Attributes

Name	Tag ID	d_un	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 er can use this as a hint to avoid relocation processing for such segments.

19.3 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)

Figure 18. Format of the Extended Program Header Attributes Section

The first part is a vector of Elf32_TI_PHAttrs, terminated by PHA_NULL. This is followed by the FLBD part and the NTBS part. If used, pha_un.pha_offset 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 access the pha_offset field.

20 Revision History

Table 41 lists changes made since the previous version of this document was published.

Table 41. Revision History

Location	Additions / Modifications / Deletions
Section 7	New section describing thread-local storage (TLS).
Section 13.5.1	Thread-local storage relocation types added.
Section 13.5.2	Thread-local storage relocation operations added.
Section 14.1.4	Points to the new section on thread-local storage.
Table 15	Discussion following table points to the new section on thread-local storage.
Section 13.3.5	Information about sections for thread-local storage.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI's terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers' products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers' products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have *not* been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products		Applications	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2014, Texas Instruments Incorporated