MSP430 Embedded Application Binary Interface

Application Report

Literature Number: SLAA534A
June 2013–Revised June 2020

Contents

List of Figures

List of Tables

Application Report
SLAA534A–June 2013–Revised June 2020

MSP430 Embedded Application Binary Interface

ABSTRACT

This document is a specification for the ELF-based Embedded Application Binary Interface (EABI) for the MSP430 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, and object file formats. The purpose of the specification is to enable tool providers, software providers, and users of the MSP430 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 MSP430 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 MSP430

Prior to release 4.0 of TI's MSP430 Compiler Tools, the one and only ABI for MSP430 was the original COFF-based ABI. It was strictly a bare-metal ABI; there was no execution-level component.

Release 4.0 of the TI Compiler Tools introduced a new ABI called the MSP430 EABI. It is based on the ELF object file format. 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, program loading, and initialization. Currently bare metal is the only supported platform. 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.

1.2 Scope

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

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

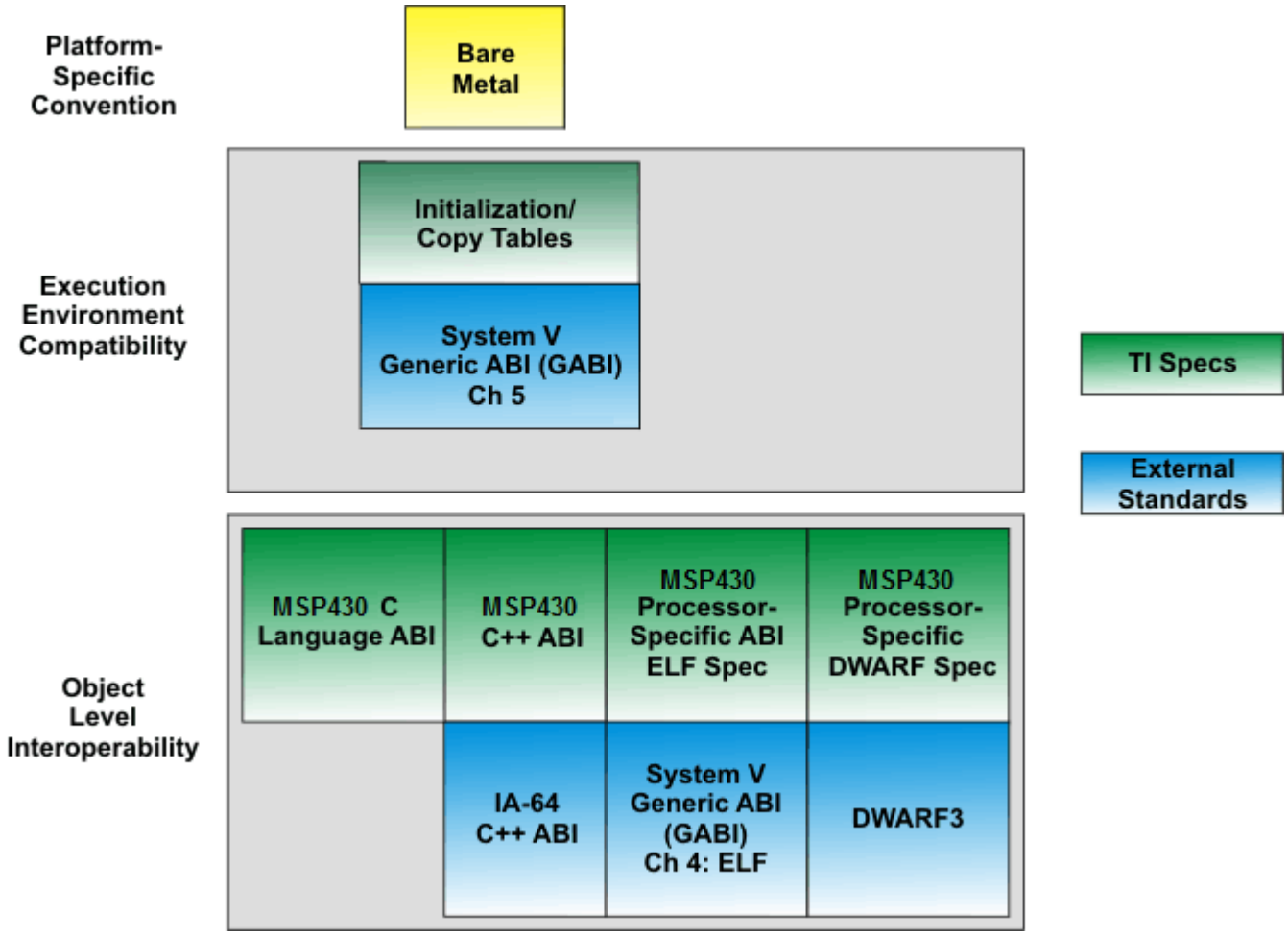


Figure 1. Parts of the ABI Specification

The **C Language ABI** (Section 2, Section 3, Section 4, Section 5, Section 6 and Section 7) specifies function calling conventions, data type representations, addressing conventions, and the interface to the C run-time library.

The **C++ ABI** (Section 8) 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 9). The MSP430 C++ ABI is based on the prevalent IA-64 (Itanium) C++ ABI.

The **DWARF** component (Section 10) 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 11) specifies the representation of object files. This specification extends the System V ABI specification with processor specific information.

Build Attributes (Section 13) 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 components in the top part of Figure 1 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.

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 14) 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.

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. There are model variants within the ABI that may be used or not used. The ABI standardizes the implementation in cases where such variants are used. Some of the variants are incompatible with each other. If any object uses a particular model, all objects must. In such cases, toolchains are expected to use build attributes to prevent incompatible objects from being combined.

- **Bare Metal—Standalone**. This model refers to a single self-contained statically-linked executable. It is the simplest form in terms of interoperability. The relevant parts of the ABI are the object-level components in the lower part of Figure 1. Since the executable is statically linked and bound (relocated), there is no need for position-independence.

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 MSP430 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 MSP430 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 file.
- An **executable** file holds a program suitable for execution.

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

1.7 Segments

An ELF load module (an executable file) 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 happens statically at link time.

1.8 MSP430 Architecture Overview

The MSP family has the following architectures:

- MSP430 has 16-bit CPU registers and a 16-bit address space.
- MSP430X has 20-bit CPU registers and a 20-bit address space. It is object-code compatible with MSP430.

Memory is little-endian only on MSP devices.

1.9 MSP430 Memory Models

Some MSP family devices support multiple data and code memory models. These models differ by the allowed size of objects and pointers. The compiler uses different instructions and relocations to implement these models.

The MSP430 has only one memory model, the small memory model. In this memory model, both code and data are restricted to the lower 64 KB of memory (16-bit address space), which is all that exists on MSP430. This means that the size of both code and data pointers are 16 bits.

The MSP430X supports the MSP430 small memory model for backward compatibility and introduces several more models to take advantage of the larger memory. Code and data memory models may be set independently, except that using the small code model requires use of the small data model. The various memory models are mutually incompatible; that is, object files with differing memory models may not be linked together.

- For code, there are small and large code models. In the small code model, function pointers are 16 bits. In the large code model, function pointers are 20 bits.
- For data, there are small, restricted, and large data models. In the small data model, object pointers are 16 bits. In the restricted and large data models, object pointers are 20 bits.

For more about memory models and placement conventions, see Section 4.3.2.1. For information about how pointers differ depending on the memory model, see Section 2.4.

1.10 Reference Documents

Document Title	Link or URL
MSP430 Optimizing C/C++ Compiler User's Guide	SLAU132
MSP430 Assembly Language Tools User's Guide	SLAU131
MSP430x2xx Family User's Guide	SLAU144
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

1.11 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; the types used by those languages will define their own mapping to these representations.

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

Table 1. Data Sizes for Standard Types

Type	Generic Name	Size	Alignment
signed char	schar	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	int16	16	16
unsigned int	uint16	16	16
long, signed long	int32	32	16
unsigned long	uint32	32	16
long long, signed long long	int64	64	16
unsigned long long	uint64	64	16
enum	--	varies (see Section 2.9)	16
float	float32	32	16
double	float64	64	16
long double	float64	64	16
pointer	--	varies (see Section 2.4)	16

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

The *char* type is unsigned by default. This is in contrast to the "signed char" and "unsigned char" types, which specify their sign behavior. In the TI toolchain, the default for the "char" type can be changed using the --plain_char=signed compiler option.

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.

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

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

See Section 2.4 for the sizes of the size_t and ptrdiff_t types, which are dependent on the selected code and data model.

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 16 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 16 bits are sign extended into the upper bits of the register. Unsigned values smaller than 16 bits are zero extended.

Objects whose size is between 16 and 32 bits use register pairs. Register pairs are any two general-purpose registers, one holding the least significant part of the value, and other holding the most significant part. In this document, register pairs are denoted as RL:RH where RL contains the LSB and RH contains the MSB (for example, R12:R13). The MSP430 instruction set does not use this notation. Signed integral values are sign extended into the upper bits of RH. Unsigned values are zero extended.

Objects whose size is greater than 32 bits and up to 64 bits use register quads. For example, R8::R11 is a register quad consisting of registers R8, R9, R10, and R11. See Section 3.3.4.

2.3 Data in Memory

The MSP430 uses little-endian mode only. Endianness refers to the memory layout of multi-byte values. 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: MOV.B for bytes, MOV.W for words, and MOVA for 20-bit addresses. These instructions correctly account for endianness when moving to and from memory.

Twenty-bit addresses have 3 bytes, designated 0 (LSB) through 2 (MSB). In memory, 20-bit values are padded to 32 bits (4 bytes). If the address of the value in memory is N, then Figure 2 gives the storage layout:

location	little-endian
N	byte 0 (lsb)
N+1	byte 1
N+2	byte 2 (msb)
N+3	padding

Figure 2. Representation of 20-Bit Values in Memory

2.4 Pointer Types

Some MSP family devices support multiple data and code memory models. These models differ by the allowed size of objects and pointers. The compiler uses different instructions and relocations to implement these models.

Section 1.9 provides an overview of the code and data models supported by the MSP family. For more about memory models and placement conventions, see Section 4.3.2.1.

The sizes of pointers vary depending on the code and data model you are using. The code and data model affects the size, alignment, and storage space used for function pointers, data pointers, the size_t type, and the ptrdiff_t type. Pointers with sizes that are not a power of 2 are always stored in a container with a size of a power of 2 bits. That is, 20-bit types are stored in 32 bits.

Table 2. Data Sizes for Pointers

Code or Data Model	Type	Size	Storage	Alignment
small code model	function pointer	16	16	16
large code model	function pointer	20	32	16
small data model	data pointer	16	16	16
small data model	size_t	16	16	16
small data model	ptrdiff_t	16	16	16
restricted data model	data pointer	20	32	16
restricted data model	size_t	16	16	16
restricted data model	ptrdiff_t	16	16	16
large data model	data pointer	20	32	16
large data model	size_t	20	32	16
large data model	ptrdiff_t	20	32	16

2.5 Complex Types

The _Complex types defined in the C99 standard are supported. The internal representation is as follows:

```
struct _Complex
    { float_type real;
      float_type imag; };
```

2.6 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.

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). The MSP430 uses little-endian mode only.

2.7 Arrays

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

2.8 Bit Fields

The MSP430 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 MSP430 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. The MSP430 uses little-endian mode only.

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 MSP430 EABI.

2.8.1 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.

2.9 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 that case, if long or unsigned long can represent all the enumerators, that type is used. Otherwise, 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

A function call is made by calling the dedicated CALL instruction, which pushes the return address to the function call stack and branches to the called function. The called function returns by executing a dedicated RET instruction, which pops the return address from the stack and branches to it.

The MSP430X uses the CALL instruction in the small code model, and the CALLA instruction in the large code model.

3.1.1 Call Instructions

The instructions used to perform calls differ depending on whether the function to be called is known or unknown as call time. For known functions, the instruction used also depends upon the architecture used.

3.1.1.1 Indirect Calls

When the function to be called is not known at compile time, for all architectures, the address of the function will be stored in a CPU register ("Rn"). This instruction reaches the entire address space. For example:

```
CALL R5 ; small code model
CALLA R5 ; large code model
```

3.1.1.2 Direct Calls

When the called function is known at compile time, all architectures use a direct call instruction. This instruction may use an immediate, absolute, or symbolic addressing mode. The examples here show only the immediate addressing modes.

The MSP430 uses the CALL instruction, which has a 16-bit addressing mode. This addressing mode reaches all valid code memory.

```
CALL #func    ; immediate mode, call func
```

The MSP430X small code model is identical to the MSP430. CALL instructions will not reach the entire address space. However, with the small code model, no code can be located beyond the lower 64 KB.

The MSP430X large code model uses the CALLA instruction, which has a 20-bit addressing mode. This addressing mode reaches all valid code memory.

```
CALLA #func   ; immediate mode, call func
```

3.1.2 Return Instruction

A called function returns by executing a dedicated RET instruction, which pops the return address from the stack and branches to it. This instruction is used on MSP430 and MSP430X with the small code model.

If the large code model is used on MSP430X, the RETA instruction is executed instead.

If the function is an interrupt handler function, the RETI instruction is used instead.

3.1.3 Pipeline Conventions

The MSP430 pipeline is protected. onsideration of pipeline latencies or instruction completion is not required (though it may be helpful in improving code performance).

3.1.4 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 ABI supports calls to imported weak functions; that is, functions potentially defined in a different static link unit. If a reference to a weak function remains unresolved at link time, the linker replaces its address with zeros. The user is responsible for adding a check that the address is not zero or NULL before attempting to call a weak function.

3.2 Register Conventions

All architectures have 16 CPU registers. These are named R0 through R15. Some of these registers are special-purpose registers. MSP430 has 16-bit registers. MSP430X has 20-bit CPU registers, with the exception of SR (R2), which is 16 bits. See the *MSP430x2xx Family User's Guide* (SLAU144) for details about registers.

R0 is the program counter (PC). The program counter must always remain aligned on a word (2-byte) boundary.

R1 is the call stack pointer (SP). The stack pointer must always remain properly aligned, even during hand-coded assembly functions (see Section 4.5.1). Both MSP430 and MSP430X require alignment to 2 bytes. Stack management and the local frame structure is presented in Section 4.5.

R2 is the status register (SR). In some addressing modes, if the register is SR, this is actually interpreted as a constant. Which constant SR represents depends on the addressing mode. See the section on "Constant Generator Registers" in the *MSP430x2xx Family User's Guide* for details.

R3 is the constant generator register . When this register is used, it is interpreted as a constant. Which constant R3 represents depends on the addressing mode.

Implementations must not use the special-purpose registers for any purpose other than the dedicated special purpose. The remaining registers are general-purpose registers.

For MSP430 and MSP430X, the ABI designates R4-R10 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.

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.

Table 3. MSP430 and MSP430X Register Conventions

Register	Alias	Preserved by Callee	Role in Calling Convention
R0	PC	N/A	Program Counter
R1	SP	yes	Call Stack Pointer
R2	SR	N/A	Status Register
R3	CG	N/A	Constant Generator Register
R4		yes	
R5		yes	
R6		yes	
R7		yes	
R8		yes	function argument (special case)
R9		yes	function argument (special case)
R10		yes	function argument (special case)
R11		no	function argument (special case)
R12		no	function argument, return value
R13		no	function argument, return value
R14		no	function argument, return value
R15		no	function argument, return value

3.2.1 Argument Registers

Four of the general-purpose registers are used to pass arguments to functions. For MSP430 and MSP430X, the argument registers are R12 through R15. For certain special cases, R8 through R11 are also used to pass arguments. See Section 3.3.5.

3.2.2 Callee-Saved Registers

A called function is required to preserve the callee-saved registers so that they have the same value on return from a function as they had at the point of the call.

For MSP430 and MSP430X, R4 through R10 are callee-saved. However, when R4 or R5 is used as a global register, neither the caller nor the callee should save, restore, or otherwise use the register. Global registers are intended as a global state for interrupt functions. Global registers are deprecated for EABI.

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

3.3 Argument Passing

For MSP430/430X, up to four arguments to a function are passed in registers.

The number of arguments passed in registers depends on the size and type of each argument. Arguments are assigned, in declared order, to the first available register single, pair, or quad from the following list into which it fits (with the following special exceptions).

For MSP430 and MSP430X, the argument registers are: **R12, R13, R14, R15**

The size of the CPU registers is different on each architecture. The ways the argument registers are used differs accordingly.

A single register cannot contain multiple arguments. A compiler may promote arguments with a type smaller than int (16 bits) to the size of a register when they are passed in a register. The TI compiler does promote such arguments. Note that the TI compiler does not promote such arguments if they are passed on the stack, unless the arguments are passed to a variadic function or a prototype is not in scope (default argument promotions). When a narrow value is to be passed in a register, the caller is responsible for correctly sign- or zero-extending it to fill the register width.

3.3.1 Register Singles

Arguments with a type that fits in a single CPU register are passed in a single CPU register.

For MSP430 and MSP430X, types up to 16 bits are passed in a single register. Pointer types are also passed in a single register, regardless of size.

For MSP430X, pointer types can be 20 bits when using large code or large data memory models, but CPU registers are also 20 bits, so pointer values always fit in a single register. For non-pointer values, MSP430X CPU registers are treated as if they had only 16 bits. A consequence of this is that the registers used to implement argument passing are the same for MSP430 and MSP430X, regardless of the memory model used.

MSP430 and MSP430X example:

C source code:

```
void func1(int a0, int a1, int a2, int a3);

int a0, a1, a2, a3;

void func2(void)
{
    func1(a0, a1, a2, a3);
}
```

Compiled assembly code:

```
            MOV.W     &a0,R12
            MOV.W     &a1,R13
            MOV.W     &a2,R14
            MOV.W     &a3,R15
        ; call instruction here
```

MSP430X example:

C source code:

```
void func1(int *a0, int *a1, int *a2, int *a3);

int a0, a1, a2, a3;

void func2(void)
{
    func1(&a0, &a1, &a2, &a3);
}
```

Compiled in large code model:

```
            MOVX.A    #a0,R12
            MOVX.A    #a1,R13
            MOVX.A    #a2,R14
            MOVX.A    #a3,R15
        ; call instruction here
```

3.3.2 Register Pairs

Arguments with a type that is larger than a single register, but no larger than twice the size of a single register, are passed in a register pair. The lowest-numbered register holds the LSW (least significant word).

For MSP430 and MSP430X, register pairs do not need to be aligned, so R12:R13, R13:R14, and R14:R15 are the valid register pairs. Types up to 32 bits are passed in a register pair. This includes "long int", "float" and structs of up to 32 bits size passed by value.

MSP430 and MSP430X example:

C source code:

```
void func1(int a0, long a1, int a2);

int a0, a2;
long a1;

func2(void)
{
    func1(a0, a1, a2);
}
```

Compiled assembly code:

```
            MOV.W     &a0,  R12
            MOV.W     &a1+0,R13
            MOV.W     &a1+2,R14
            MOV.W     &a2,  R15
```

3.3.3 Split Pairs

For MSP430 and MSP430X, a 32-bit argument may be split between the stack and memory. If an argument would be passed in a register pair, but only one register is available (always R15), the compiler will split the argument between R15 and one register-sized spot on the stack.

MSP430 and MSP430X example:

C source code:

```
void func1(int a0, long a1, long a2);

   int a0;
long a1, a2;

func2(void)
{
    func1(a0, a1, a2);
}
```

Compiled assembly code:

```
        SUB.W     #2,SP
        MOV.W     &a0,  R12
        MOV.W     &a1+0,R13
        MOV.W     &a1+2,R14
        MOV.W     &a2+0,R15
        MOV.W     &a2+2,0(SP)
```

3.3.4 Quads (Four-Register Arguments)

For MSP430 and MSP430X, arguments whose size is greater than 32 bits and up to 64 bits use register quads. For example, R8::R11 is the notation used in this manual for a register quad consisting of the registers R8, R9, R10, and R11 in sequence. The lowest-numbered register holds the LSW. Register quads must be aligned. Therefore, only R8::R11 and R12::R15 are valid register quads. Register quads require special handling.

If there are enough argument registers remaining (all four) to pass the 64-bit value, all four will be used. Otherwise, the 64-bit value will be passed entirely on the stack. This may leave unused argument registers (in other words, a "hole"). The calling convention will try to back-fill the hole with subsequent arguments, but only if they fit entirely in registers (see example 2 that follows). That is, after any argument has been placed on the stack, no argument will be placed in a "split pair", as in Section 3.3.3.

MSP430 and MSP430X example 1:

C source code

```
void func1(long long a0, long long a1);

long long a0, a1;

func2(void)
{
    func1(a0, a1);
}
```

Compiled in any model:

```
        SUB.W     #8,SP
        MOV.W     &a0+0,R12
        MOV.W     &a0+2,R13
        MOV.W     &a0+4,R14
        MOV.W     &a0+6,R15
        MOV.W     &a1+0,0(SP)
        MOV.W     &a1+2,2(SP)
        MOV.W     &a1+4,4(SP)
        MOV.W     &a1+6,6(SP)
```

MSP430 and MSP430X example 2:

This example shows a 64-bit argument that doesn't fit entirely in registers, so it is passed entirely on the stack, leaving unused registers which are then back-filled with arguments a2, a3, and a4.

C source code

```
void func1(int a0, long long a1, int a2, int a3, int a4);

int a0, a2, a3, a4;
long long a1;

func2(void)
{
    func1(a0, a1, a2, a3, a4);
}
```

Compiled in any model:

```
        SUB.W     #8,SP
        MOV.W     &a0,R12
        MOV.W     &a1+0,0(SP)
        MOV.W     &a1+2,2(SP)
        MOV.W     &a1+4,4(SP)
        MOV.W     &a1+6,6(SP)
        MOV.W     &a2,R13
        MOV.W     &a3,R14
        MOV.W     &a4,R15
```

MSP430 and MSP430X example 3:

This example shows a 64-bit argument that doesn't fit entirely in registers, so it is passed entirely on the stack, leaving unused registers which are then back-filled. However, the last argument (a 32-bit type) does not fit entirely in registers, so it is passed entirely on the stack. This type would have been split and passed partially in R15 and partially on the stack if the 64-bit argument weren't already on the stack.

C source code

```
void func1(int a0, long long a1, long a2, long a3);

int a0;
long long a1;
long a2, a3;

func2(void)
{
    func1(a0, a1, a2, a3);
}
```

Compiled in any model (note that R15 is unused):

```
        SUB.W     #12,SP
        MOV.W     &a0,R12
        MOV.W     &a1+0,0(SP)
        MOV.W     &a1+2,2(SP)
        MOV.W     &a1+4,4(SP)
        MOV.W     &a1+6,6(SP)
        MOV.W     &a2+0,R13
        MOV.W     &a2+2,R14
        MOV.W     &a3+0,8(SP)
        MOV.W     &a3+2,10(SP)
```

On MSP430 and MSP430X, holes and back-fill can only occur when register quads are used. If only singles and pairs are used, argument registers are used in order, there is no back-filling, and there are no holes.

3.3.5 Special Convention for Compiler Helper Functions

For MSP430 and MSP430X, for efficiency, the compiler uses a special calling convention for certain compiler helper functions that take two 64-bit arguments ("long long int" and "double" arithmetic).

In this special case, the compiler allows two register quads for argument passing: R8::R11 and R12::R15. This is the only case in which R8 through R11 are used as argument registers. The first argument is passed in R8::R11 and the second argument is passed in R12::R15. The return value is in R12::R15, as usual.

See Section 6.3 for helper functions that use modified conventions.

MSP430 and MSP430X example:

C source code

```
long long a1, a2;

long long func(void)
{
    return a1 / a2;
}
```

Compiled in small code, small data model:

```
 func:
        PUSH.W    R10    ; R10 is caller-saved!
        PUSH.W    R9    ; R9  is caller-saved!
        PUSH.W    R8    ; R8  is caller-saved!
        MOV.W     &a1+0,R8
        MOV.W     &a1+2,R9
        MOV.W     &a1+4,R10
        MOV.W     &a1+6,R11
        MOV.W     &a2+0,R12
        MOV.W     &a2+2,R13
        MOV.W     &a2+4,R14
        MOV.W     &a2+6,R15
        CALL      #__mspabi_divlli
        BR        #__mspabi_func_epilog_3
```

3.3.6 C++ Argument Passing

In C++, the "this" pointer is passed to non-static member functions in R12 as an implicit first argument. (If a non-static member function returns a struct by reference, the order is "&struct", "this".)

3.3.7 Passing Structs and Unions

Structures and unions are passed by reference, as described in Section 3.5.

3.3.8 Stack Layout of Arguments Not Passed in Registers

Any arguments not passed in registers are placed on the stack at increasing addresses, starting at 0(SP). 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 is the smallest power of two greater than or equal to its size. This is to allow loading arguments with aligned loads, even if the type is not naturally aligned strictly enough, which might be the case with struct of size 32 containing an array of char.
- Each argument reserves an amount of stack space equal to its size rounded up to the next multiple of its stack alignment.

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

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

Alignment "holes" can occur between arguments passed on the stack, but "back-fill" does not occur.

3.3.9 Frame Pointer

MSP430 does not use a frame pointer. This effectively limits a single call frame to 0x7fff bytes, which is the minimum SP offset supported by any instruction.

3.4 Return Values

The function return value is placed in the same register as the usual first argument register, based on its type and size.

For MSP430 and MSP430X, the return value is placed in R12, R12:13, or R12::R15. Return values with a type that fits in a single CPU register are placed in R12; this includes pointer types. 32-bit return values are placed in R12:R13. 64-bit return values are placed R12::R15. The LSW is always in R12.

Structures and unions are returned by reference.

3.5 Structures and Unions Passed and Returned by Reference

Structures (including classes) and unions 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 need to make its own copy of the pointed-to object. In some cases, the callee need not make a copy, such as if the callee is a leaf and it does not modify the pointed-to object.

The caller must pass an additional argument containing a destination address for the returned value, or NULL if the returned value is not used.

This additional argument is passed in the first argument register as an implicit first argument. The callee returns the object by copying it to the given address. 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 the first argument register.

Examples

C source code:

```
struct S { char big[100]; } g;

struct S accepts_and_returns_struct(struct S s)
{
    s.big[0] = 1;
    return s;
}

void caller(void)
{
    struct S w;
    w.big[0] = 0;
    g = accepts_and_returns_struct(w);
}
```

"Lowered" C code: (higher-level C code converted to lower-level C code)

```
struct S { char big[100]; } g;

void accepts_and_returns_struct(struct S *dst, struct S *sptr)
{
    struct S s;
    s = *sptr;
    s.big[0] = 1;
    if (dst) *dst = s;
}

void caller(void)
{
    struct S w;
    w.big[0] = 0;
    accepts_and_returns_struct(&g, &w);
}
```

3.6 Conventions for Compiler Helper Functions

The ABI specifies *helper functions* that the compiler uses to implement language features. Generally, these functions adhere to the standard calling conventions, but an exception is made for a handful of them to enable better performance. See Section 6.3 for helper functions that use modified conventions.

3.7 Scratch Registers for Functions Already Seen

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 have special conventions as described in Section 6.3.

3.8 _ _mspabi_func_epilog Helper Functions

On MSP430 only, the _ _mspabi_func_epilog set of helper functions reduce code size. Each function performs a typical POP-and-RET function epilog sequence. Code size is reduced by replacing the typical POP-and-RET epilog sequence with a branch to one of these functions. Each function is named after the number of consecutive registers (ending with R10) that it restores. The expected implementation is:

```
__mspabi_func_epilog:
      __mspabi_func_epilog_7:   POP    R4
      __mspabi_func_epilog_6:   POP    R5
      __mspabi_func_epilog_5:   POP    R6
      __mspabi_func_epilog_4:   POP    R7
      __mspabi_func_epilog_3:   POP    R8
      __mspabi_func_epilog_2:   POP    R9
      __mspabi_func_epilog_1:   POP    R10
                                RET
```

These functions are not needed on MSP430X, because multiple-register POP instructions are used.

3.9 Interrupt Functions

Interrupt functions must save all the registers that are used, even those that are normally considered callee-saved, except for the special purpose registers PC, SP, and SR.

Interrupts push the SR and PC registers onto the stack and branch to an interrupt handler. To return from an interrupt function, the function must execute the special instruction RETI, which restores the SR register and branches to the PC where the interrupt occurred.

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 3. Conventions for placement of static variables into sections and for how they are addressed are covered in Section 4.3.2.

The linker combines sections from object files to form segments in an ELF load module (executable). 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:

- 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.

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

Figure 3 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.

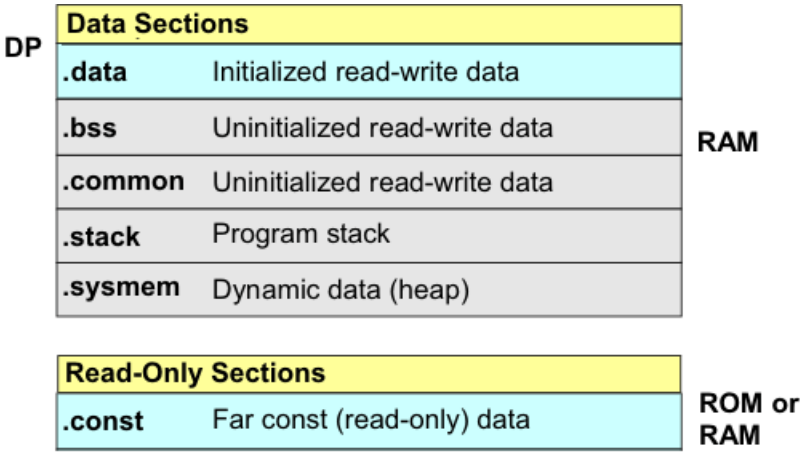


Figure 3. Data Sections and Segments (Typical)

The .const section contains read-only constants. The .const section may be located in read-only memory, and may be addressed using absolute addressing.

The .data section contains initialized read-write data.

The .bss section contains uninitialized read-write data.

The .common section contains common-block symbols allocated by the linker. This is not an actual section in the object files. Instead, the section name is a convention in the linker command file for placing variables. This section should not be used for other purposes.

Additional special sections that can be placed by the linker command file are listed in Section 11.3.5.

4.2 Addressing Modes

The MSP family uses a variety of assembly code addressing modes to implement data and code memory models. These modes are briefly listed here and described in detail in the "Addressing Modes" section of the *MSP430x2xx Family User's Guide* (SLAU144).

Addressing modes determine the type of relocation used. Relocations are discussed in Section 11.5.

Table 4. MSP430 Addressing Modes

Mode Name	Assembly Example	Relocation Type	Notes
Register mode	R5	no relocation	
Indexed mode	X(R5)	absolute relocation	
Symbolic mode	ADDR	PC-relative relocation	This is actually Indexed mode, but with PC as the base register.
Absolute mode(1)	&ADDR	absolute relocation	This is actually Indexed mode, but with SR as the base register. When used in this way, SR is treated as 0.
Indirect register mode	@R5	no relocation	
Indirect autoincrement mode	@R5+	no relocation	
Immediate mode	#X	absolute relocation	This is actually Indirect autoincrement mode, but relative to the PC register.

(1) "Absolute mode" is the term used in this user's guide for the addressing mode in which the exact address is encoded in the instruction. When a relocation is used for Absolute mode, it is an absolute relocation, but absolute relocations may be used for other addressing modes.

4.3 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) link time. These are allocated into various sections according to their properties and then combined into one or more static data segments.

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

4.3.1 Addressing Methods for Static Data

The ABI supports the following fundamental schemes for addressing static data: Absolute 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, 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.

4.3.1.1 Absolute Addressing

The following instruction loads the contents of address "sym" into R5.

```
MOV.W    &sym, R5
```

Because this addressing mode encodes an address, it is position-dependent.

4.3.1.2 Symbolic Addressing

Symbolic addressing is performed using indexed addressing relative to the PC register. (See Section 4.2.) The data is assumed to be located at a (link-time) constant offset from the code that accesses it. The addressing mechanism is the same whether addressing code or data. Symbolic addressing is sometimes called PC-relative addressing.

Examples include label tables for switch statements, and read-only constant variables that can be placed into the code segment (.const).

The following instruction loads the contents of "sym" into R5.

```
MOV.W    sym, R5
```

4.3.1.3 Immediate Addressing

The following instructions use immediate addressing.

```
MOV.W #sym, R5  ; put the address of sym into R5
MOV.W #123, R5  ; put the value 123 into R5
```

4.3.2 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 addressing.

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.

4.3.2.1 Abstract Conventions for Placement

Some MSP family devices support multiple data and code memory models. These models differ by the allowed size of objects and pointers. The compiler uses different instructions and relocations to implement these models. Section 1.9 provides an overview of the code and data models supported by the MSP family. For information about how pointers differ depending on the memory model, see Section 2.4.

The MSP family does not have a distinction between "near" and "far" addresses.

In addition to the models affecting the size of pointers, the data model affects the types of size_t and ptrdiff_t, and thus dictates the maximum allowable object size.

For the MSP430 and the MSP430X with the small data model, all memory locations are accessible with 16-bit addresses.

For the MSP430X with the restricted and large data models, most addresses are still accessible with a 16-bit address and all bits are zero above the 16th bit. Some instructions (such as CALL) handle only 16-bit pointers, so these instructions can only be used in the lower 64 KB of memory space; they ignore the upper 4 bits of the 20-bit pointers. Other instructions (such as CALLA) make use of the full 20-bit address; these instructions can reach the entire memory space.

MSP430X devices do not have any writeable memory above the 64 KB boundary. For these devices, even when the restricted or large data models are used, only constant data will be placed above 64 KB. The compiler can take advantage of this knowledge to produce more efficient code. This is controlled by the --near_data compiler option. If --near_data=globals is specified, it tells the compiler that all global read/write data must be located in the first 64 KB of memory. This is the default behavior. If --near_data=none is specified, the compiler cannot rely on this assumption to generate more efficient code.

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.

The ABI establishes conventional assignments of variables to sections. A variable's assignment is a function of 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

	Initialization category		
	Uninitialized	**Initialized**	**Const**
	.bss	.data	.const

The conventional assignments may be overridden in toolchain-specific ways. For example, variables may be assigned to user-defined sections.

4.3.2.2 Abstract Conventions for Addressing

All MSP430 variables are located within reach of absolute addressing (position dependent). Position-independent addressing is not supported.

4.3.3 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 14. Other toolchains may adopt a compatible mechanism, a different mechanism, or none at all.

4.4 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 via the stack pointer (R1).

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 address in the current frame. That is, the function may read/write 0(SP).

Section 4.5 provides more detail on the stack conventions and local frame structure.

4.5 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 lowest-addressed location within this function's frame. That is, 0(SP) is allocated, but -1(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 does not use a frame pointer, so a single call frame is limited to 0xffff bytes.

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, the exception handling stack unwinding instructions assume that no frame pointer is available.

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.

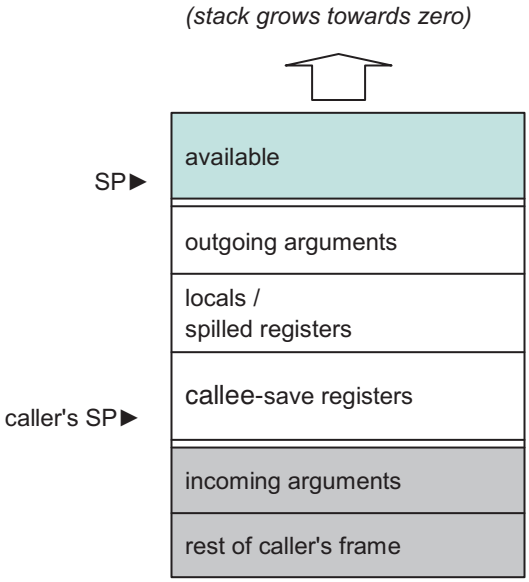


Figure 4. Local Frame Layout

Before the frame is allocated, SP points to the return address (SR for interrupt functions).

4.5.1 Stack Alignment

For MSP430 and MSP430X, the SP is 2-byte aligned.

The stack must remain properly aligned at all times, including during frame allocation and deallocation. This means that every atomic adjustment to SP must be a multiple of the required alignment.

Alignment must be maintained at all times even in hand-coded assembly functions, including interrupt functions. Hand-coded assembly functions may call or be called by C-compatible functions, and interrupts with C-compatible handlers could occur at any time.

4.5.2 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:

- MSP430X has PUSHM and POPM instructions which push or pop a certain number of consecutive registers to the stack. These instructions are more efficient than using individual PUSH and POP instructions for each register. These instructions push the registers in the following order, starting with the highest numbered register being pushed at the bottom (highest address) of the frame: **R10 R9 R8 R7 R6 R5 R4**. POPM of course pops them in the reverse order.
- **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* (PUSHM order) for encoding the bit vectors, so the orderings are the same.

When compiling for MSP430X or supporting exception handling, the compiler always saves registers in the PUSHM relative order, starting at the bottom (lowest 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.

4.6 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. Instructions have a variable length from 16 bits to 64 bits, in exact multiples of 16.

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
- Fill switch tables

There are three basic ways to form an address: Absolute, Symbolic, and Immediate. Absolute and Immediate addressing are position-dependent. Symbolic (PC-relative) addressing is position-independent. These modes are briefly listed in Section 4.2 and described in detail in the "Addressing Modes" section of the *MSP430x2xx Family User's Guide* (SLAU144).

5.1.1 Absolute Addressing for Code

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

For MSP430/MSP430X with the small code model, the following CALL instruction uses absolute addressing and calls the function whose address is stored in location func.

```
CALL    &func
```

For MSP430X with the large code model, the following CALLA instruction is equivalent to the previous CALL instruction.

```
CALLA   &func
```

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 Symbolic Addressing

Symbolic addressing is performed using indexed (absolute) addressing relative to the PC register. The desired address is the sum of the effective PC and the indexed addressing mode offset. This offset is not visible in the assembly code. The mechanism is the same whether addressing code or data. Symbolic addressing is sometimes called *PC-relative addressing*.

For MSP430/MSP430X with the small code model, the following CALL instruction uses symbolic addressing and calls the function whose address is stored in location "sym".

```
CALL    sym
```

For MSP430X with the large code model, the following CALLA instruction is equivalent to the previous CALL instruction.

```
CALLA   sym
```

5.1.3 Immediate Addressing

This is the addressing mode typically used to make a function call. You can use immediate addressing to load the address of function "func" into R5.

```
MOV.W #func, R5   ; immediate
```

For MSP430/MSP430X with the small code model, the following CALL instruction uses immediate addressing and calls the function beginning at address "func".

```
CALL     #func
```

For MSP430X with the large code model, the following CALLA instruction is equivalent to the previous CALL instruction.

```
CALLA    #func
```

5.2 Branching

Branches are always assumed to be within the same function, and therefore can always use Symbolic (PC-relative) addressing and be resolved no later than static link time.

For MSP430 and MSP430X small code model, branches and calls use the BR and CALL instructions, which have 16-bit offsets.

For MSP430X large code model, branches and calls use the BRA and CALLA instructions, which have 20-bit offsets.

5.3 Calls

A function call is made by calling a dedicated CALL instruction, which pushes the return address to the function call stack and branches to the called function. The called function returns by executing a dedicated RET instruction, which pops the return address from the stack and branches to it.

5.3.1 Direct 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.

5.3.2 Far Call Trampoline

The entire address space of the MSP430 and MSP430X can be reached by direct calls. Therefore, trampolines are not used.

5.3.3 Indirect Calls

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

```
CALL  R5                    ; indirect call
```

6 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.

6.1 Floating-Point Behavior

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

The MSP430 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).

6.2 C Helper Function API

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

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

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

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 6 convert between various floating-point and integer formats, in accordance with C's conversion rules and the floating-point behavior specified by Section 6.1.

Table 6. MSP430 Floating-Point and Integer Conversions

Signature	Description
float32 _ _**mspabi_cvtdf**(float64 x);	Convert double-precision float to single-precision float
float64 _ _**mspabi_cvtfd**(float32 x);	Convert single-precision float to double-precision float
int16 _ _**mspabi_fixdi**(float64 x);	Convert double-precision float to int
int32 _ _**mspabi_fixdli**(float64 x);	Convert double-precision float to long int
int64 _ _**mspabi_fixdlli**(float64 x);	Convert double-precision float to long long int
uint16 _ _**mspabi_fixdu**(float64 x);	Convert double-precision float to unsigned int

Table 6. MSP430 Floating-Point and Integer Conversions (continued)

Signature	Description
uint32 _ _**mspabi_fixdul**(float64 x);	Convert double-precision float to unsigned long int
uint64 _ _**mspabi_fixdull**(float64 x);	Convert double-precision float to unsigned long long int
int16 _ _**mspabi_fixfi**(float32 x);	Convert single-precision float to int
int32 _ _**mspabi_fixfl**i(float32 x);	Convert single-precision float to long int
int64 _ _**mspabi_fixflli**(float32 x);	Convert single-precision float to long long int
uint16 _ _**mspabi_fixfu**(float32 x);	Convert single-precision float to unsigned int
uint32 _ _**mspabi_fixful**(float32 x);	Convert single-precision float to unsigned long int
uint64 _ _**mspabi_fixfull**(float32 x);	Convert single-precision float to unsigned long long int
float64 _ _**mspabi_fltid**(int16 x);	Convert int to double-precision float
float32 _ _**mspabi_fltif**(int16 x);	Convert int to single-precision float
float64 _ _**mspabi_fltlid**(int32 x);	Convert long int to double-precision float
float32 _ _**mspabi_fltlif**(int32 x);	Convert long int to single-precision float
float64 _ _**mspabi_fltud**(uint16 x);	Convert unsigned int to double-precision float
float32 _ _**mspabi_fltuf**(uint16 x);	Convert unsigned int to single-precision float
float64 _ _**mspabi_fltuld**(uint32 x);	Convert unsigned long int to double-precision float
float32 _ _**mspabi_fltulf**(uint32 x);	Convert unsigned long int to single-precision float

The functions in Table 7 perform floating-point comparisons in accordance with C semantics and the floating-point behavior specified by Section 6.1.

The _ _mspabi_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 remaining comparison functions are currently unsupported, but the names are reserved for possible future use.

Table 7. MSP430 Floating-Point Comparisons

Signature	Description
int16 _ _**mspabi_cmpd**(float64 x, float64 y);	Double-precision comparison
int16 _ _**mspabi_cmpf**(float32 x, float32 y);	Single-precision comparison
int16 _ _**mspabi_eqd**(float64 x, float64 y);	Double-precision comparison: x == y (Not currently supported)
int16 _ _**mspabi_geqd**float64 x, float64 y);	Double-precision comparison: x >= y (Not currently supported)
int16 _ _**mspabi_gtrd**(float64 x, float64 y);	Double-precision comparison: x > y (Not currently supported)
int16 _ _**mspabi_leqd**(float64 x, float64 y);	Double-precision comparison: x <= y (Not currently supported)
int16 _ _**mspabi_lssd**(float64 x, float64 y);	Double-precision comparison: x < y (Not currently supported)
int16 _ _**mspabi_neqd**(float64 x, float64 y);	Double-precision comparison: x != y (Not currently supported)

The functions in Table 8 perform floating-point arithmetic in accordance with C semantics and the floating-point behavior specified by Section 6.1.

Table 8. MSP430 Floating-Point Arithmetic

Signature	Description
float64 _ _**mspabi_addd**(float64 x, float64 y);	Add double-precision to double-precision
float32 _ _**mspabi_addf**(float32 x, float32 y);	Add single-precision to single-precision
float64 _ _**mspabi_divd**(float64 x, float64 y);	Divide double-precision by double-precision
float32 _ _**mspabi_divf**(float32 x, float32 y);	Divide single-precision by single-precision
float64 _ _**mspabi_mpyd**(float64 x, float64 y);	Multiply double-precision by double-precision
float32 _ _**mspabi_mpyf**(float32 x, float32 y);	Multiply single-precision by single-precision
float64 _ _**mspabi_subd**(float64 x, float64 y);	Subtract double-precision from double-precision
float32 _ _**mspabi_subf**(float32 x, float32 y);	Subtract single-precision from single-precision
float64 _ _**mspabi_negd**(float64 x);	Negate double-precision
float32 _ _**mspabi_negf**(float32 x);	Negate single-precision

The integer arithmetic functions in Table 9 operate according to C semantics.

Table 9. MSP430 Integer Multiply, Divide, and Remainder

Signature	Description
	Multiplication
int16 _ _**mspabi_mpyi**(int16 x, int16 y);	Multiply int by int.
int16 _ _**mspabi_mpyi_hw**(int16 x, int16 y);	Multiply int by int. Uses hardware MPY16.
int16 _ _**mspabi_mpyi_f5hw**(int16 x, int16 y);	Multiply int by int. Uses hardware MPY32 (F5xx devices and up).
int32 _ _**mspabi_mpyl**(int32 x, int32 y);	Multiply long by long.
int32 _ _**mspabi_mpyl_hw**(int32 x, int32 y);	Multiply long by long. Uses hardware MPY16.
int32 _ _**mspabi_mpyl_hw32**(int32 x, int32 y);	Multiply long by long. Uses hardware MPY32 (F4xx devices).
int32 _ _**mspabi_mpyl_f5hw**(int32 x, int32 y);	Multiply long by long. Uses hardware MPY32 (F5xx devices and up).
int64 _ _**mspabi_mpyll**(int64 x, int64 y);	Multiply long long by long long.
int64 _ _**mspabi_mpyll_hw**(int64 x, int64 y);	Multiply long long by long long. Uses hardware MPY16.
int64 _ _**mspabi_mpyll_hw32**(int64 x, int64 y);	Multiply long long by long long. Uses hardware MPY32 (F4xx devices).
int64 _ _**mspabi_mpyll_f5hw**(int64 x, int64 y);	Multiply long long by long long. Uses hardware MPY32 (F5xx devices and up).
int32 _ _**mspabi_mpysl**(int16 x, int16 y);	Multiply int by int; result is long.
int32 _ _**mspabi_mpysl_hw**(int16 x, int16 y);	Multiply int by int; result is long. Uses hardware MPY16.
int32 _ _**mspabi_mpysl_f5hw**(int16 x, int16 y);	Multiply int by int; result is long. Uses hardware MPY32 (F5xx devices and up).
int64 _ _**mspabi_mpysll**(int32 x, int32 y);	Multiply long by long; result is long long.
int64 _ _**mspabi_mpysll_hw**(int32 x, int32 y);	Multiply long by long; result is long long. Uses hardware MPY16.
int64 _ _**mspabi_mpysll_hw32**(int32 x, int32 y);	Multiply long by long; result is long long. Uses hardware MPY32 (F4xx devices).
int64 _ _**mspabi_mpysll_f5hw**(int32 x, int32 y);	Multiply long by long; result is long long. Uses hardware MPY32 (F5xx devices and up).
uint32 _ _**mspabi_mpyul**(uint16 x, uint16 y);	Multiply unsigned int by unsigned int; result is unsigned long.
uint32 _ _**mspabi_mpyul_hw**(uint16 x, uint16 y);	Multiply unsigned int by unsigned int; result is unsigned long. Uses hardware MPY16.
uint32 _ _**mspabi_mpyul_f5hw**(uint16 x, uint16 y);	Multiply unsigned int by unsigned int; result is unsigned long. Uses hardware MPY32 (F5xx devices and up).
uint64 _ _**mspabi_mpyull**(uint32 x, uint32 y);	Multiply unsigned long by unsigned long; result is unsigned long long.
uint64 _ _**mspabi_mpyull_hw**(uint32 x, uint32 y);	Multiply unsigned long by unsigned long; result is unsigned long long. Uses hardware MPY16.
uint64 _ _**mspabi_mpyull_hw32**(uint32 x, uint32 y);	Multiply unsigned long by unsigned long; result is unsigned long long. Uses hardware MPY32 (F4xx devices).

Table 9. MSP430 Integer Multiply, Divide, and Remainder (continued)

Signature	Description
uint64 _ _**mspabi_mpyull_f5hw**(uint32 x, uint32 y);	Multiply unsigned long by unsigned long; result is unsigned long long. Uses hardware MPY32 (F5xx devices and up).
Division	
int16 _ _**mspabi_divi**(int16 x, int16 y);	Divide int by int.
int32 _ _**mspabi_divli**(int32 x, int32 y);	Divide long by long.
int64 _ _**mspabi_divlli**(int64 x, int64 y);	Divide long long by long long.
uint16 _ _**mspabi_divu**(uint16 x, uint16 y);	Divide unsigned lint by unsigned lint.
uint32 _ _**mspabi_divlu**(uint32 x, uint32 y);	Divide unsigned long by unsigned long.
uint64 _ _**mspabi_divllu**(uint64 x, uint64 y);	Divide unsigned long long by unsigned long long.
Remainder (Modulus)	
int16 _ _**mspabi_remi**(int16 x, int16 y);	Remainder of int divided by int (x mod y)
int32 _ _**mspabi_remli**(int32 x, int32 y);	Remainder of long divided by long (x mod y)
int64 _ _**mspabi_remlli**(int64x. int64 y);	Remainder of long long divided by long long (x mod y)
uint16 _ _**mspabi_remu**(uint16 x, uint16 y);	Remainder of unsigned int divided by unsigned int (x mod y)
uint32 _ _**mspabi_remul**(uint32, uint32);	Remainder of unsigned long divided by unsigned long (x mod y)
uint64 _ _**mspabi_remull**(uint64, uint64);	Remainder of unsigned long long divided by unsigned long long (x mod y)

The bitwise operator functions are listed in Table 10. These functions are used by MSP430 and MSP430X.

Rotate left operations are performed with no carry. The most-significant bit is moved to the least-significant bit (LSB), and all other bits are shifted left. There is no rotation function for the long long data type.

Logical left shift operations are the same as an arithmetic left shift; a zero is shifted into the LSB.

Arithmetic right shift operations shift the most-significant bit (MSB), to the right and copy the old MSB into the new MSB. This preserves the sign in a signed value. The LSB is discarded.

Logical right shift operations insert a value of 0 in the MSB and discard the LSB.

The shift or rotation count in functions that accept a second argument may not be less than zero, even though it is handled as a signed int.

Table 10. MSP430 / MSP430X Bitwise Operations

Signature	Description
Rotation	
uint16 _ _**mspabi_rlli**(uint16 x, int16 n);	Rotate bits of an int left by n bits, where n can be up to 16.
uint16 _ _**mspabi_rlli_1**(uint16 x); . . . uint16 _ _**mspabi_rlli_15**(uint16 x);	Rotate bits of an int left by specified number of bits.
uint32 _ _**mspabi_rlll**(uint32 x, int16 n);	Rotate bits of a long left by n bits, where n can be up to 32.
Logical Left Shift	
uint16 _ _**mspabi_slli**(uint16 x, int16 n);	Perform a logical left shift on an int. Shift by n bits, where n can be up to 16.
uint16 _ _**mspabi_slli_1**(uint16 x); . . . uint16 _ _**mspabi_slli_15**(uint16 x);	Perform a logical left shift on an int. Shift left by specified number of bits.
uint32 _ _**mspabi_slll**(uint32 x, int16 n);	Perform a logical left shift on a long. Shift by n bits, where n can be up to 32.
uint32 _ _**mspabi_slll_1**(uint32 x); . . . uint32 _ _**mspabi_slll_15**(uint32 x);	Perform a logical left shift on a long. Shift left by specified number of bits. Shift counts above 15 use _ _mspabi_slll.
uint64 _ _**mspabi_sllll**(uint64 x, int16 n);	Perform a logical left shift on a long long. Shift by n bits, where n can be up to 64.
Arithmetic Right Shift	
int16 _ _**mspabi_srai**(int16 x, int16 n);	Perform an arithmetic right shift on an int. Shift by n bits, where n can be up to 16.

Table 10. MSP430 / MSP430X Bitwise Operations (continued)

Signature	Description
int16 _ _**mspabi_srai_1**(int16 x); . . . int32 _ _**mspabi_srai_15**(int16 x);	Perform an arithmetic right shift on an int. Shift by specified number of bits.
int16 _ _**mspabi_sral**(int32 x, int16 n);	Perform an arithmetic right shift on a long. Shift by n bits, where n can be up to 32.
int32 _ _**mspabi_sral_1**(int32 x); . . . int32 _ _**mspabi_sral_15**(int32 x);	Perform an arithmetic right shift on a long. Shift by specified number of bits. Shift counts above 15 use _ _mspabi_sral.
int64 _ _**mspabi_srall**(int64 x, int16 n);	Perform an arithmetic right shift on a long long. Shift by n bits, where n can be up to 64.
Logical Right Shift	
uint16 _ _**mspabi_srli**(uint16 x, int16 n);	Perform a logical right shift on an int. Shift by n bits, where n can be up to 16.
uint16 _ _**mspabi_srli_1**(uint16 x); . . . uint16 _ _**mspabi_srli_15**(uint16 x);	Perform a logical right shift on an int. Shift right by specified number of bits.
uint32 _ _**mspabi_srll**(uint32 x, int16 n);	Perform a logical right shift on a long. Shift by n bits, where n can be up to 32.
uint32 _ _**mspabi_srll_1**(uint32 x); . . . uint32 _ _**mspabi_srll_15**(uint32 x);	Perform a logical right shift on a long. Shift right by specified number of bits. Shift counts above 15 use _ _mspabi_srll.
uint64 _ _**mspabi_srlll**(uint64 x, int16 n);	Perform a logical right shift on a long long. Shift by n bits, where n can be up to 64.

The helper functions in Table 11 optimize restoring callee-saved registers. These functions are used by MSP430, but not MSP430X.

Table 11. MSP430 Epilog Helper Functions

Signature	Description
void _ _**mspabi_epilog_1**(void);	POP R10 and return.
void _ _**mspabi_epilog_2**(void);	POP R10 through R9 and return.
void _ _**mspabi_epilog_3**(void);	POP R10 through R8 and return.
void _ _**mspabi_epilog_4**(void);	POP R10 through R7 and return.
void _ _**mspabi_epilog_5**(void);	POP R10 through R6 and return.
void _ _**mspabi_epilog_6**(void);	POP R10 through R5 and return.
void _ _**mspabi_epilog_7**(void);	POP R10 through R4 and return.

The miscellaneous helper functions in Table 12 are described in the sections that follow.

Table 12. MSP430 Miscellaneous Helper Functions

Signature	Description
void _**abort_msg**(const char *string);	Report failed assertion

_abort_msg

The function _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.

6.3 Special Register Conventions for Helper Functions

The following functions take 2 64-bit values:

- _ _mspabi_mpyll
- _ _mspabi_divull
- _ _mspabi_remull
- _ _mspabi_divlli

- _ _mspabi_remlli
- _ _mspabi_srall
- _ _mspabi_srlll
- _ _mspabi_sllll
- _ _mspabi_addd
- _ _mspabi_subd
- _ _mspabi_mpyd
- _ _mspabi_divd
- _ _mspabi_cmpd

For MSP430 and MSP430X, these functions have a special calling convention. The first argument is in R8::R11 and the second argument is in R12::R15. That is:

```
First argument word 0 (LSW): R8
First argument word 1      : R9
First argument word 2      : R10
First argument word 3 (MSW): R11

Second argument word 0 (LSW): R12
Second argument word 1      : R13
Second argument word 2      : R14
Second argument word 3 (MSW): R15
```

Furthermore, for many of the helper functions, the library implementation must take care not to modify certain registers.

See Section 3.3.5 for more about how large arguments are stored in registers.

6.4 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 13. Reserved Floating-Point Classification Helper Functions

Signature	Description
int32 _ _**mspabi_isfinite**(float64 x);	True iff x is a representable value
int32 _ _**mspabi_isfinitef**(float32 x);	True iff x is a representable value
int32 _ _**mspabi_isinf**(float64 x);	True iff x represents "infinity"
int32 _ _**mspabi_isinff**(float32 x);	True iff x represents "infinity"
int32 _ _**mspabi_isnan**(float64 x);	True iff x represents "not a number"
int32 _ _**mspabi_isnanf**(float32 x);	True iff x represents "not a number"
int32 _ _**mspabi_isnormal**(float64 x);	True iff x is not denormalized
int32 _ _**mspabi_isnormalf**(float32 x);	True iff x is not denormalized
int32 _ _**mspabi_fpclassify**(float64 x);	Classify floating-point value
int32 _ _**mspabi_fpclassifyf**(float32 x);	Classify floating-point value

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

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

7 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 MSP EABI further requires that modules compiled using the header files from one implementation are compatible with the library from another implementation. This is called "header file compatibility." This requirement imposes additional limitations on the library header files beyond what is specified in the C standard.

The following sections describe any issues that apply to the C standard header files. These issues cover any requirements that are not specified in the ANSI C standard but which must be followed in order for a toolchain to support the MSP430 ABI.

The MSP430 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 MSP430. See the chapter on "The C Library Section by Section" in that document.

7.1 Reserved Symbols

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

- _ftable
- _ctypes_

In addition, any symbols listed in Section 11.4.4 or symbols with the prefixes listed in Section 11.1 are reserved.

7.2 <assert.h> Implementation

The library must implement assert as a macro. If its expression argument is false, it must eventually call a helper function _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 12.

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

7.3 <complex.h> Implementation

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

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

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

7.4 <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)

7.5 <errno.h> Implementation

errno is a global int defined as follows:

```
extern int errno;
```

The following are some of the constants defined for used with errno. See the errno.h file for a complete list.

```
#define EDOM 0x21
#define ERANGE 0x22
#define EILSEQ 0x58
#define ENOENT 0x2
#define EFPOS 0x98
```

7.6 <float.h> Implementation

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

7.7 <inttypes.h> Implementation

The macros, functions and typedefs in this file are defined in the natural way according to the integer types of the architecture. See Section 2.1.

7.8 <iso646.h> Implementation

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

7.9 <limits.h> Implementation

Aside from MB_LEN_MAX, the macros in this file are defined in the natural way according to the integer types of the architecture. See Section 2.1.

MB_LEN_MAX is defined as follows:

```
#define MB_LEN_MAX 1
```

7.10 <locale.h> Implementation

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

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

The order of the fields in the lconv struct is as follows:

MSP430/430X order: (These are the C89 fields. Additional fields added for C99 are not included.)

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

7.11 <math.h> Implementation

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

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

7.12 <setjmp.h> Implementation

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

With the small code and small data model combination, the size and alignment of jmp_buf is the same as an array of 9 "int"s (that is, 16 bits * 9). For all other code and data model combinations, the size and alignment of jmp_buf the same as an array of is 9 "long"s (that is, 32 bits * 9).

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

7.13 <signal.h> Implementation

TI's toolset 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
```

7.14 <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.

7.15 <stdbool.h> Implementation

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

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

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

7.16 <stddef.h> Implementation

The size and alignment of each type defined in stddef.h is defined in Section 2.4.

7.17 <stdint.h> Implementation

The macros and typedefs in this header file are defined in the natural way according to the integer types of the architecture. See Section 2.1.

7.18 <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 MSP430 EABI does not have a "compatibility mode" (like the mode in the ARM EABI) in which stdin, stdout, and stderr are link-time symbols, not macros. The lack of a compatibility mode means that linkers that need to interlink with a module that refers to stdin directly need to support _ftable.

If a program does not use the stdin, stdout, or stderr macros (or a function implemented as a macro that refers to one of these macros), there are no issues with the FILE array.

C I/O functions commonly implemented as macros—getc, putc, getchar, putchar—must not be inlined.

The fpos_t type is defined as a long.

7.19 <stdlib.h> Implementation

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

```
typedef struct { int quot; 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.

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.

7.20 <string.h> Implementation

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

7.21 <tgmath.h> Implementation

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

7.22 <time.h> Implementation

Some 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.

The type for CLOCKS_PER_SEC is clock_t.

7.23 <wchar.h> Implementation

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

```
typedef int wint_t;

#define WEOF ((wint_t)-1)
```

The type mbstate_t is the size and alignment of int.

7.24 <wctype.h> Implementation

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

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

8 C++ ABI

The C++ ABI specifies aspects of the implementation of the C++ language that must be standardized in order for code from different toolchains to interoperate. The MSP430 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.

8.1 Limits (GC++ABI 1.2)

The GC++ABI constrains the offset of a non-virtual base subobject in the full object containing it to be representable by a 56-bit signed integer, due to the RTTI implementation. For the MSP family, 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.

8.2 Export Template (GC++ABI 1.4.2)

Export templates are not currently specified by the ABI.

8.3 Data Layout (GC++ABI Chapter 2)

The layout of POD (Plain Old Data), is specified in 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 2.8.

8.4 Initialization Guard Variables (GC++ABI 2.8)

The guard variable is a one-byte field stored in the first byte of a 16-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 16 bits instead of 64.

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

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

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

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

8.5 Constructor Return Value (GC++ABI 3.1.5)

The MSP430 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 MSP430 ABI they are declared as pointers to functions that return void *, corresponding to 'this'.

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

The guard variable is an 8-bit field stored in the first byte of a 16-bit container. See Section 8.4.

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

The MSP430 ABI does not specify a mechanism to control object construction.

8.8 Demangler API (GC++ABI 3.4)

The MSP430 ABI suspends the requirement for an implementation to provide the function _ _cxa_demangle, which provides a run-time interface to the demangler.

8.9 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 MSP430 ABI forbids it altogether; the static variable and its guard variable must be defined in a single COMDAT group with the static variable's name as the signature.

8.10 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 MSP430 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.

8.11 Unwind Table Location (GC++ABI 5.3)

Exception handling is covered in Section 9 of this document.

9 Exception Handling

The MSP430 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.

9.1 Overview

The MSP430'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 MSP430-specific portions.

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

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

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

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

9.2 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_MSP430_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.

9.3 The Exception Index Table (EXIDX)

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

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

Each entry consists of two 32-bit words. The first word of each entry is a 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.

9.3.1 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

9.3.2 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)	

9.3.3 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 9.4, but with no descriptors and no terminating NULL. This saves 4 bytes that would have been a pointer to an out-of-line EXTAB entry plus 4 bytes for the terminating NULL.

31	30-28	27-24	23-0
0	PREL31 Representation of function address		
1	**000**	PR Index	Data for personality routine specified by 'index'

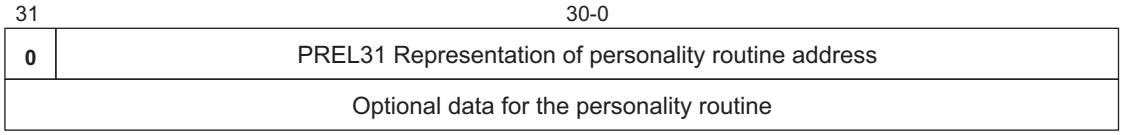
9.4 The Exception Handling Instruction Table (EXTAB)

Each EXTAB entry 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 14.

9.4.1 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.

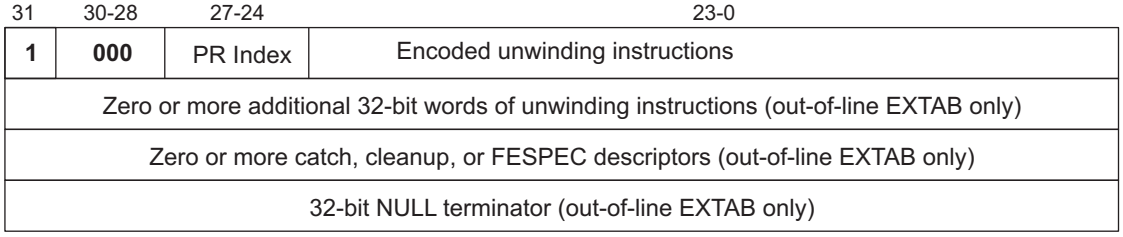


31	30-0
0	PREL31 Representation of personality routine address
Optional data for the personality routine	

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.

9.4.2 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)			

9.4.3 Personality Routines

The MSP430 has the following ABI-specified personality routines. They have the same format as the ARM EABI. The following table specifies the personality routines and their PR indexes.

Table 14. MSP430 TDEH Personality Routines

PR Index (bits 27-24)	Personality	Routine Name	Unwind Instructions	Width of Scope Fields	Notes
0000	PR0 (Su16)	__mspabi_unwind_cpp_pr0	Up to 3 one-byte instructions	16	
0001	PR1 (Lu16)	__mspabi_unwind_cpp_pr1	Unlimited onebyte instructions	16	
0010	PR2 (Lu32)	__mspabi_unwind_cpp_pr2	Unlimited onebyte instructions	32	Must be used if 16-bit scope fields will not reach

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_MSP430_NONE relocation.

9.5 Unwinding Instructions

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

The unwinding instructions make assumptions about the stack layout; in particular, *callee-saved register safe debug order* is always assumed.

9.5.1 Common Sequence

Abstractly, all unwinding sequences take the following form:

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

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 incrementing by a constant. In addition to the explicit increment, the SP is implicitly incremented to account for the size of the callee-saved area.

Step 2: Restore Registers

Abstractly, the callee-saved registers are restored in *register safe debug* order (Section 4.5.2) starting with the location pointed to by (the old) SP and moving to lower addresses.

Step 3: Return

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

9.5.2 Byte-Encoded Unwinding Instructions

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

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

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

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

Table 15. Stack Unwinding Instructions

Encoding	Instruction	Description
0xxx xxxx	POP bitmask (R10, R9, R8, R7, R6, R5, R4) + RET	Restore callee-saved registers and return.
11kk kkkk	SP += (kkkkkk << 1) + 2 [0x02-0x80]	Increment by small constant
1000 0001 kkkk	SP += (ULEB128 << 1) + 0x102 [0x102-max]	Increment by large constant
1000 0000 0000 0000	CANTUNWIND	Function cannot be unwound

All other bit patterns are reserved.

The following paragraphs detail the interpretation of the unwinding instructions.

POP + RET

The POP+RET instruction specifies a bitmask representing registers saved by this function's prolog. These registers must be popped in order, starting with R4 and proceeding through R10. When that is done, there are no more unwinding instructions. If none of the bits in the bitmask are set, this is simply a RET instruction.

Small Increment

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

9.6 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			

9.6.1 Encoding of Type Identifiers

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

9.6.2 Scope

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

The scope encodes a program address range by specifying an offset from the starting address of the function and a length, both in bytes. If the length and offset each fit in a 15-bit unsigned field, the scope uses the short form encoding and the rest of the EXTAB entry can be encoded for PR0 or PR1. 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 5. 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 6. 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

9.6.3 Cleanup Descriptor

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

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.

9.6.4 Catch Descriptor

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

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

31-0

Scope (long or short form)	0
R	PREL31 program address of landing pad
Type	

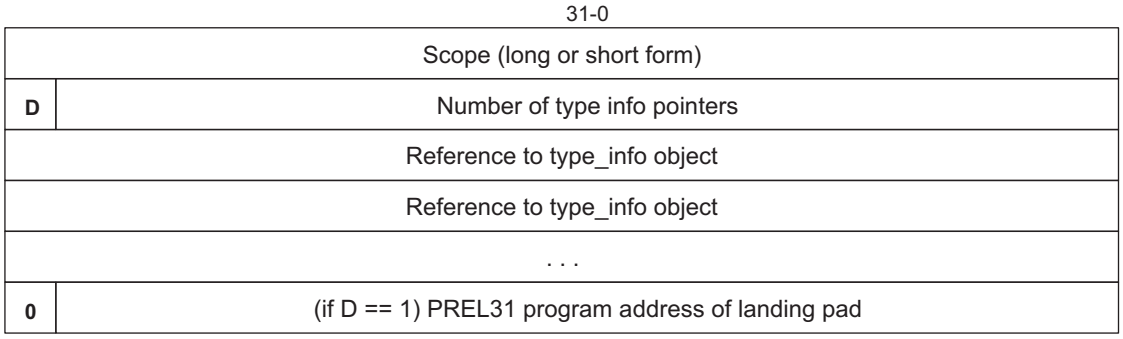
If bit R is 1, the type of the catch clause is a reference type represented by TYPE. If bit R is 0, the type is not a reference type.

The type field is either a reference to a type_info object or one of two special values:

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

9.6.5 Function Exception Specification (FESPEC) Descriptor

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



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_MSP430_NONE relocation to _ _cxa_call_unexpected.

9.7 Special Sections

All of the exception handling tables are stored in two sections. The EXIDX table is stored in a section called .mspabi.exidx with type SHT_MSP430_UNWIND. The linker must combine all the input .mspabi.exidx sections into one contiguous .mspabi.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 .mspabi.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.

9.8 Interaction With Non-C++ Code

9.8.1 Automatic EXIDX Entry Generation

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

9.8.2 Hand-Coded Assembly Functions

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

9.9 Interaction With System Features

9.9.1 Shared Libraries

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

9.9.2 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.

9.9.3 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.

9.10 Assembly Language Operators in the TI Toolchain

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

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

$EXIDX_FUNC
The argument is a function address to be encoded using the 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.)

$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.

10 DWARF

The MSP430 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 MSP430.

10.1 DWARF Register Names

DWARF3 registers use register name operators (see Section 2.6.1 of the DWARF3 standard). The operand of a register name operator is a register number representing an architecture register. Table 16 defines mappings from DWARF3 register numbers/names to MSP430 registers.

Table 16. DWARF3 Register Numbers for MSP

DWARF Name	MSP430 ISA Register	Description
0-15	R0-R15	See Table 3 for details.

10.2 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 16, indexed by their DWARF register numbers from the first column.

For the CFA, the MSP430 ABI follows the convention suggested in the DWARF3 standard, defining it as the value of SP (R1) at the call site in the previous frame (that of the calling procedure).

The unwinding table may include registers that are not present on all MSP430 ISAs. Therefore a situation may arise in which the ISA executing the program has registers that are not mentioned in the call frame information. In this situation, the interpreter should behave as follows:

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

10.3 Vendor Names

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

TI	MSP430 Code Generation Tools from Texas Instruments
GNU	The GNU Compiler Collection (GCC)

10.4 Vendor Extensions

The DWARF standard allows toolchain vendors to define additional tags 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 MSP430 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 17 defines TI vendor-specific DIE tags that apply to the MSP430. Table 17 defines TI vendor-specific attributes.

Table 17. 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 18. 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_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_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.

11 ELF Object Files (Processor Supplement)

The MSP430 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).

The subsections that follow contain MSP430 processor-specific supplements for Chapter 4 (Object Files) of the specification. Section 12 of this document contains processor-specific supplements for Chapter 5 (Program Loading and Dynamic Linking) of the specification.

11.1 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 19.

Table 19. Registered Vendors

Name	Vendor
cxa, _ _cxa	C++ ABI namespace. Applies to all symbols specified by the C++ ABI.
mspabi, _ _mspabi	Common namespace for symbols specified by the MSP430 EABI.
MSP430	Common namespace for symbols specified by the MSP430.
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_MSP430_EXIDX for MSP430, SHT_C2000_EXIDX for C2000, and so on.

11.2 ELF Header

The ELF header provides a number of fields that guide interpretation of the file. Most of these are specified in the System V ELF specification. This section augments the base standard with specific details for the MSP430.

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 20 specifies the values to be used for MSP430 object files.

Table 20. 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
EI_VERSION	EV_CURRENT	1	
EI_ABIVERSION		0	

The EI_OSABI field shall be ELFOSABI_NONE unless overridden by the conventions of a specific platform. No platforms for the MSP family override the default setting of the EI_OSABI field; its value is always ELFOSABI_NONE.

e_type

There are currently no MSP430-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_MSP430 (105, 0x69).

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 are no MSP430-specific flags for e_flags field.

11.3 Sections

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

11.3.1 Section Indexes

The MSP430 ABI does not define any special section indexes.

11.3.2 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 21.

Not all these section types are used in the MSP430 ABI. Some are specific to the TI toolchain but outside the ABI, and some are used by TI toolchains for architectures other than MSP430. They are documented here for completeness, and to reserve the tag values.

Table 21. ELF and TI Section Types

Name	Value	Comment
SHT_MSP430_UNWIND	0x70000001	Unwind function table for stack unwinding
SHT_MSP430_PREEMPTMAP	0x70000002	DLL dynamic linking pre-emption map (not supported by MSP430)
SHT_MSP430_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_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_MSP430_UNWIND identifies a section containing unwind function table for stack unwinding. See Section 9 for details.

SHT_MSP430_ATTRIBUTES identifies a section containing object compatibility attributes. See Section 13.

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 14.

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

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

SHT_TI_SH_PAGE is used only on targets that have distinct, possibly overlapping, address spaces (pages). The section contains a table that associates other sections with page numbers. This section type is not used on MSP430.

11.3.3 Extended Section Header Attributes

For the MSP430, the following processor-specific attribute flag may be used in the TI toolchain:

SHF_MSP_NOINIT identifies a section that contains variables that are not initialized. The NOINIT attribute can apply only to the .TI.noinit and .TI.persistent sections. For example:

```
".TI.noinit"     SHT_NOBITS      SHF_MSP_NOINIT
".TI.persistent" SHT_PROGBITS    SHF_MSP_NOINIT
```

Linkers should not create .cinit records for these sections.

11.3.4 Subsections

MSP430 object files use a section naming convention that provides improved granularity while retaining the convenience of default rules for combining sections at link time. A section whose name contains a colon is called a *subsection*. Subsections behave as normal sections in all respects, but their name guides the linker when combining sections into output files. The root name of a subsection is the name up to, but not including, the colon. The suffix includes all characters following the colon. By default, the linker combines all sections with matching roots into a single section with that name. For example, .text, text:func1, and .text:func2 are combined into a single section called .text. The user may be able to override this default behavior in toolchain-specific ways.

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

Subsections whose root names match special sections have the same ABI-defined properties as the section they match, as defined in Section 11.3.5. For example .text:func1 is an instance of a .text section.

11.3.5 Special Sections

The System V ABI, along with other base documents and other sections of this ABI, defines several sections with dedicated purposes. Table 22 consolidates dedicated sections used by the MSP430 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 22. MSP430 Special Sections

Prefix	Type	Attributes
Code Sections		
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
Data Sections		
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.TI.noinit	SHT_NOBITS	SHF_MSP_NOINIT
.TI.persistent	SHT_PROGBITS	SHF_MSP_NOINIT
.const	SHT_PROGBITS	SHF_ALLOC
Exception Handling Data Sections		
.mspabi.exidx	SHT_MSP430_UNWIND	SHF_ALLOC + SHF_LINK_ORDER
.mspabi.extab	SHT_PROGBITS	SHF_ALLOC
Initialization and Termination Sections		
.init_array	SHT_INIT_ARRAY	SHF_ALLOC + SHF_WRITE
ELF Structures		
.rel	SHT_REL	None
.rela	SHT_RELA	None
.symtab	SHT_SYMTAB	None
.symtab_shndx	SHT_SYMTAB_SHNDX	None
.strtab	SHT_STRTAB	SHF_STRINGS
.shstrtab	SHT_STRTAB	SHF_STRINGS
.note	SHT_NOTE	None
Build Attributes		
.mspabi.attributes	SHT_MSP430_ATTRIBUTES	None
Symbolic Debug Sections		
.debug [(1)]	SHT_PROGBITS	None
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
.binit	SHT_PROGBITS	SHF_ALLOC
.cinit	SHT_TI_INITINFO	SHF_ALLOC
.const:handler_table	SHT_PROGBITS	SHF_ALLOC
.ovly	SHT_PROGBITS	SHF_ALLOC
.ppdata	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.ppinfo	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.TI.crctab	SHT_PROGBITS	SHF_ALLOC
.TI.icode	SHT_TI_ICODE	None
.TI.xref	SHT_TI_XREF	None
.TI.section.flags	SHT_TI_SH_FLAGS	None
.TI.symbol.alias	SHT_TI_SYMALIAS	None

(1) 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.

Table 22. MSP430 Special Sections (continued)

Prefix	Type	Attributes
.TI.section.page	SHT_TI_SH_PAGE	None
Sections in the System V ABI but Unused by the MSP430 EABI [(2)]		
.comment		
.data1		
.dsbt		
.dynamic		
.dynstr		
.dynsym		
.far		
.fardata		
.fardata:const		
.fini		
.fini_array		
.gnu.version		
.gnu.version_d		
.gnu.version_r		
.got		
.hash		
.init		
.interp		
.line		
.neardata		
.plt		
.preinit_array		
.rodata		
.rodata1		
.tbss		
.tdata		
.tdata1		
.TI.tls_init		

(2) These section names are unused for MSP430, which does not support the features related to these sections. However, this section name is reserved and should not be used.

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

The "**Sections in the System V ABI but Unused by the MSP430 EABI**" sections in the previous table are specified by the System V ABI, but are not used or defined under the MSP430 ABI. Other sections are used by TI for other devices; these names are reserved.

In addition, .common is a section name used by the linker. This is an abstract section, not an actual section in the object files. The name is a convention in the linker command file for placing variables. This section should not be used for other purposes.

11.3.6 Section Alignment

Sections containing MSP430 code must be at least 16-byte aligned, and padded to 16-byte boundaries.

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

11.4 Symbol Table

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

The MSP430 ABI follows the ELF specification with respect to global and weak symbol definitions, and the meaning of symbol values.

11.4.1 Symbol Types

This specification adheres to the ARM ELF specification with respect to Symbol Types, namely:

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

11.4.2 Common Block Symbols

As described in the ELF specification, symbols with type STT_COMMON are allocated by the linker.

Common block symbols addressed with other addressing forms should have section index SHN_COMMON, as described in the base ELF specification.

11.4.3 Symbol Names

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

The MSP430 compiler follows the following naming convention for temporary symbols:

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

11.4.4 Reserved Symbol Names

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

- Local symbols (STB_LOCAL) beginning with $
- Global symbols (STB_GLOBAL, STB_WEAK) beginning with any of the vendor names listed in Table 19.
- 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 4.6)

11.4.5 Mapping Symbols

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

11.5 Relocation

The ELF relocations for MSP430 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 MSP430 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.

The effects of addressing modes on relocations is briefly described in Section 4.2.

11.5.1 Relocation Types

Relocation types are described using two tables.

Table 23 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 24 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 23.

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 24. 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. This may not be the same as the address of the instruction containing the relocation.

Table 23. MSP430 and MSP430X Relocation Types

Name	Value	Operation	Constraints
R_MSP430_NONE	0		
R_MSP430_ABS32	1	S + A	
R_MSP430_ABS16	2	S + A	
R_MSP430_ABS8	3	S + A	
R_MSP430_PCR16	4	S + A - PC	
R_MSP430X_PCR20_EXT_SRC	5	S + A - PC	
R_MSP430X_PCR20_EXT_DST	6	S + A - PC	
R_MSP430X_PCR20_EXT_ODST	7	S + A - PC	
R_MSP430X_ABS20_EXT_SRC	8	S + A	
R_MSP430X_ABS20_EXT_DST	9	S + A	
R_MSP430X_ABS20_EXT_ODST	10	S + A	
R_MSP430X_ABS20_ADR_SRC	11	S + A	
R_MSP430X_ABS20_ADR_DST	12	S + A	
R_MSP430X_PCR16	13	S + A - PC	
R_MSP430X_PCR20_CALL	14	S + A - PC	
R_MSP430X_ABS16	15	S + A	
R_MSP430_ABS_HI16	16	S + A	Rela only
R_MSP430_PREL31	17	S + A - PC	

11.5.1.1 Absolute Relocations

Absolute relocations directly encode the relocated address of a symbol. MSP "Indexed," "Immediate," and "Absolute" addressing modes all require absolute relocations. The relocation types with names containing "ABS" are all absolute relocation types.

11.5.1.2 PC-Relative Relocations

PC-relative relocations encode addresses as signed PC-relative offsets. MSP "Symbolic" addressing mode requires PC-relative relocations. The relocation types with names containing "PCR" are PC-relative relocation types.

In the assembler and linker, displacements are computed relative to the address of the container of the relocation as defined in Table 24, not the starting address of the instruction. Because the PC advances while reading the individual words of the instruction, the effective PC value that will be used by the hardware when performing the addressing mode may differ from the address of the relocation container. To compensate for this, the assembler must adjust the relocation addend by the difference.

11.5.1.3 Relocations in Data Sections

The R_MSP430_ABS16/32 relocation types directly encode the relocated address of a symbol into 16-, or 32-bit fields. These relocations are used to relocate addresses in initialized data sections. The signedness of the field is unspecified for R_MSP430_ABS16/32; that is, these relocation types are used for both signed and unsigned values. They are also used for some instruction relocations, as shown here:

```
.field X,32         ; R_MSP430_ABS32
.field X,16         ; R_MSP430_ABS16
```

11.5.1.4 Relocations for MSP430 Instructions

MSP430 instructions allow only a 16-bit field. MSP430 instruction relocations are not typically not checked for overflow. MSP430 instructions can also be used on MSP430X, but the assembler uses different relocations so that overflow can be checked.

R_MSP430_ABS16 and R_MSP430_PCR16 are used for MSP430 instructions. With one exception (R_MSP430_ABS16 for $LO16), neither is used for instructions on MSP430X. The use of R_MSP430_ABS16 for $LO16 for MSP430X is a special case intended to be half of a 32-bit immediate load; the other half is $HI16 (R_MSP430_ABS_HI16). See the description of R_MSP430_ABS_HI16 in Section 11.5.1.5.

R_MSP430_ABS16 is used for Absolute, Indexed, and Immediate addressing modes, but only on MSP430:

```
ADD.W #X, R5      ; R_MSP430_ABS16
ADD.W &X, R5      ; R_MSP430_ABS16
ADD.W R5, &X      ; R_MSP430_ABS16
ADD.W K(R4), R5   ; R_MSP430_ABS16
```

R_MSP430_PCR16 is used for Symbolic addressing mode, but only on MSP430. The address of the relocation container matches the effective PC, so the addend does not need to be adjusted.

```
MOV.W X, R5       ; R_MSP430_PCR16
MOV.W R5, X       ; R_MSP430_PCR16
MOV.W X, Y        ; R_MSP430_PCR16(X) and R_MSP430_PCR16(Y)
CALL X            ; R_MSP430_PCR16
```

R_MSP430X_ABS16 and R_MSP430X_PCR16 (both MSP430X only) are used for MSP430 instructions assembled for MSP430X. For instructions where MSP430 would use R_MSP430_ABS16, MSP430X uses R_MSP430X_ABS16 instead. R_MSP430_ABS16 is identical to R_MSP430X_ABS16 except that that the latter checks for overflow. R_MSP430_PCR16 and R_MSP430X_PCR16 are similar.

R_MSP430X_ABS16 is used for Absolute, Indexed, and Immediate addressing modes, but only on MSP430X:

```
ADD.W #X, R5      ; R_MSP430X_ABS16
ADD.W &X, R5      ; R_MSP430X_ABS16
ADD.W R5, &X      ; R_MSP430X_ABS16
ADD.W K(R4), R5   ; R_MSP430X_ABS16
```

R_MSP430X_PCR16 is used for Symbolic addressing mode, but only on MSP430X:

```
MOV.W X, R5      ; R_MSP430X_PCR16
MOV.W R5, X      ; R_MSP430X_PCR16
MOV.W X, Y       ; R_MSP430X_PCR16(X) and R_MSP430X_PCR16(Y)
CALL X           ; R_MSP430X_PCR16
```

R_MSP430X_ABS16 and R_MSP430X_PCR16 are also used for most of the MSP430X-only "address instructions" (MOVA, ADDA, SUBA, and CMPA, but not CALLA), because these instructions only allow a 16-bit immediate in "Indexed" addressing mode. (There are other instructions that allow a larger immediate in Indexed addressing mode.)

```
MOVA K(R4), R5   ; R_MSP430X_ABS16 (MSP430X only)
MOVA X, R5       ; R_MSP430X_PCR16
```

11.5.1.5 Relocations for MSP430X Instructions

See Section 11.5.1.4 for MSP430-native instructions assembled on MSP430X.

MSP430X has instruction encodings with immediate fields that are different than MSP430, requiring distinct relocations.

The "ABS20" and "PCR20" relocations are used for "MSP430X extended instructions." These instructions require an extra op-code prefix word to encode 20-bit addressing modes (MOVX, CMPX, ADDX, etc).

The "ABS20" relocations are also used for the "MSP430X address instructions." These instructions allow one 20-bit addressing mode without requiring an extra op-code prefix word (ADDA, MOVA, CMPA, SUBA, CALLA).

R_MSP430X_ABS20_EXT_SRC, R_MSP430X_ABS20_EXT_DST, and R_MSP430X_ABS20_EXT_ODST relocations are used for Absolute, Indexed, and Immediate addressing modes, but only on MSP430X. They are distinct because they encode different fields in the instruction encoding. SRC relocates the source operand; DST and ODST relocate the destination operand. ODST is used instead of DST if the addressing mode of the source operand requires an additional word to encode, including 20-bit addresses and 20-bit immediate constants that cannot be handled with the constant generator.

```
MOVX &X, R5       ; R_MSP430X_ABS20_EXT_SRC
MOVX #X, R5       ; R_MSP430X_ABS20_EXT_SRC
MOVX R5, &Y       ; R_MSP430X_ABS20_EXT_DST
MOVX #0xabcde, &Y ; R_MSP430X_ABS20_EXT_ODST (Y)
MOVX &X, &Y       ; R_MSP430X_ABS20_EXT_SRC (X) and R_MSP430X_ABS20_EXT_ODST (Y)
MOVX #X, &Y       ; R_MSP430X_ABS20_EXT_SRC (X) and R_MSP430X_ABS20_EXT_ODST (Y)
ADDX K(R4), R5    ; R_MSP430X_ABS20_EXT_SRC (K)
```

R_MSP430X_PCR20_EXT_SRC, R_MSP430X_PCR20_EXT_DST, and R_MSP430X_PCR20_EXT_ODST are similar to the EXT20 relocation types, except that they are used for Symbolic addressing mode.

```
MOVX X, R5       ; R_MSP430X_PCR20_EXT_SRC
MOVX R5, Y       ; R_MSP430X_PCR20_EXT_DST
MOVX #0xabcde, Y ; R_MSP430X_PCR20_EXT_ODST (Y)
MOVX X, Y        ; R_MSP430X_PCR20_EXT_SRC (X) and R_MSP430X_PCR20_EXT_ODST (Y)
```

R_MSP430X_PCR20_CALL is used for CALLA with Symbolic addressing mode:

```
CALLA  Y         ; R_MSP430X_PCR20_CALL
```

For R_MSP430X_PCR20_EXT_SRC, R_MSP430X_PCR20_EXT_DST, R_MSP430X_PCR20_EXT_ODST, and R_MSP430X_PCR20_CALL, the effective PC does not match the address of the relocation container. To compensate for this, the assembler must adjust the relocation addend by the difference. The addend must be adjusted by adding -4 for SRC and DST, -6 for ODST, and -2 for CALL.

R_MSP430X_ABS20_ADR_SRC and R_MSP430X_ABS20_ADR_DST are used for Absolute, Indexed, and Immediate addressing modes, but only on MSP430X. They are distinct because they encode different fields in the instruction encoding. SRC relocates the source operand; DST relocates the destination operand. CALLA uses DST.

```
MOVA &X, R5      ; R_MSP430X_ABS20_ADR_SRC
MOVA R5, &X      ; R_MSP430X_ABS20_ADR_DST
MOVA #X, R5      ; R_MSP430X_ABS20_ADR_SRC

CALLA #X         ; R_MSP430X_ABS20_ADR_DST
CALLA &X         ; R_MSP430X_ABS20_ADR_DST
```

R_MSP430_ABS_HI16 is used to load a 32-bit linker symbol value. A linker symbol value is larger than a 20-bit pointer, so a single instruction cannot be used to load the entire value. The value must be split into two relocations. This relocation type represents the MSW of the value. It is intended to be used as a pair with R_MSP430_ABS16 representing the LSW. This relocation type is legal but unused on MSP430.

Example: The TI compiler will generate this type of relocation pair from the following C code:

```
extern char X;

unsigned long fn()
{
    return _symval(&X);
}
```

```
 MOV #$LO16(X), R12    ; R_MSP430_ABS16
 MOV #$HI16(X), R13    ; R_MSP430_ABS_HI16
```

11.5.1.6 Other Relocation Types

The R_MSP430_NONE relocation type 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 referent section.

R_MSP430_PREL31 is used to encode code addresses in exception handling tables. See Section 9.2.

R_MSP430_EHTYPE is used to encode typeinfo addresses in exception handling tables. See Section 9.6.1.

11.5.2 Relocation Operations

Table 24 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. The notation [x,y]+[z,w] indicates that relocation occupies discontiguous bit ranges, which should be concatenated to form the field. When "F" is used in the addend column, it indicates that the field is already of the exact size of the address space.
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.
r_addend	The addend must be stored in a RELA field, and may not be stored in the relocation container.

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 … 2^{FS}).
- A relocation whose signedness is indicated as *either* overflows if the encoded value falls outside the half-open interval [-2^{FS-1}… 2^{FS}).

Table 24. MSP430 Relocation Operations

Relocation Name	Signedness	Container Size (CS)	Field [O, FS] (F)	Addend (A)	Result (R)	Overflow Check	Encoded Value (EV)
R_MSP430_NONE	None	32	[0,32]	None	None	No	None
R_MSP430_ABS32	Either	32	[0,32]	F	S + A	No	R
R_MSP430_ABS16	Either	16	[0,16]	SE(F)	S + A	No	R
R_MSP430_ABS8	Either	8	[0,8]	SE(F)	S + A	Yes	R
R_MSP430_PCR16	Signed	16	[0,16]	SE(F)	S + A - P	No	R
R_MSP430X_PCR20_EXT_SRC	Signed	48	[7,4]+[32,16]	SE(F)	S + A - P	Yes	R
R_MSP430X_PCR20_EXT_DST	Signed	48	[0,4]+[32,16]	SE(F)	S + A - P	Yes	R
R_MSP430X_PCR20_EXT_ODST	Signed	64	[0,4]+[48,16]	SE(F)	S + A - P	Yes	R
R_MSP430X_ABS20_EXT_SRC	Unsigned	48	[7,4]+[32,16]	ZE(F)	S + A	Yes	R
R_MSP430X_ABS20_EXT_DST	Unsigned	48	[0,4]+[32,16]	ZE(F)	S + A	Yes	R
R_MSP430X_ABS20_EXT_ODST	Unsigned	64	[0,4]+[48,16]	ZE(F)	S + A	Yes	R
R_MSP430X_ABS20_ADR_SRC	Unsigned	32	[8,4]+[16,16]	ZE(F)	S + A	Yes	R
R_MSP430X_ABS20_ADR_DST	Unsigned	32	[0,4]+[16,16]	ZE(F)	S + A	Yes	R
R_MSP430X_PCR16	Signed	16	[0,16]	SE(F)	S + A - P	Yes	R
R_MSP430X_PCR20_CALL	Signed	32	[0,4]+[16,16]	SE(F)	S + A - P	Yes	R
R_MSP430X_ABS16	Unsigned	16	[0,16]	SE(F)	S + A	Yes	R
R_MSP430_ABS_HI16	None	16	[0,16]	r_addend	S + A	No	R >> 16
R_MSP430_PREL31	Signed	32	[0,31]	SE(F)	S + A - P	No	R >> 1

* See Section 11.5.1.4.

11.5.3 Relocation of Unresolved Weak References

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

- References to weak functions shall be implemented using Immediate addressing mode.
- When used in an absolute relocation type (R_MSP430_ABS*) the reference resolves to zero.

All other cases are non-conformant with the ABI.

12 ELF Program Loading and Linking (Processor Supplement)

In general, *program loading* describes the steps involved in taking a program represented as an ELF file and beginning its execution. By its nature, this process is platform and system specific.

A system may use a subset of the mechanisms depending on its specific requirements.

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 MSP430-based systems.

12.1 Program Header

The program header contains the following fields.

p_type

The MSP430 defines no processor-specific segment types for the p_type field in the program header.

p_vaddr, p_paddr

The MSP430 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 are no processor-specific segment flags defined for MSP430.

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 MSP430 the meaning and setting of p_align is unspecified.

12.1.1 Base Address

MSP430 does not support position-independent code as described in the "Base Address" section of Chapter 5 of the System V ABI standard.

Segments that are not position independent must either be loaded at their specified address or relocated at load time.

12.1.2 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 MSP430 program may contain any number of code and data segments, including multiple code 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.

12.1.3 Thread-Local Storage

The ABI does not currently specify a standard mechanism for thread-local storage.

12.2 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 MSP430.

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 25, Table 26 and Table 27 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.

Table 25. Steps to Create a Process Image from an ELF Executable

Step	
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 12.1.
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.	Marshall command line arguments and environment variables. This step is platform specific.

Table 26. Steps to Initialize the Execution Environment

Step	
5.	Set SP. SP (R1) should be set to the value of the symbol _ _TI_STACK_END, properly aligned on an 8-byte boundary.
6.	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 14.
7.	Perform initialization calls. Generally these are calls to constructors for global objects defined in the module. Pointers to initialization functions are stored in a table. The table is delimited by a pair of global symbols: _ _TI_INITARRAY_Base and _ _TI_INITARRAY_Limit.
8.	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 27. Termination Steps

Step	
9.	Perform atexit calls. Functions registered by atexit are called, in reverse order of registration.

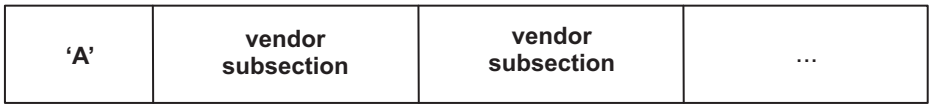
13 Build Attributes

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 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.

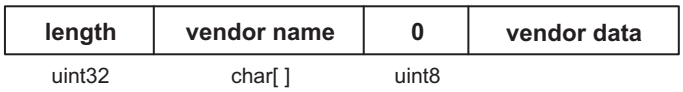
Build attributes are classified as vendor-specific or ABI-specific. The section documents build attributes that are ABI-specific. Vendors are free to implement additional toolchain-specific attributes.

Every ABI conforming relocatable file must contain the build attributes section of type SHT_MSP430_ATTRIBUTES (0x70000003), conventionally named mspabi.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.



Each subsection has the following format:

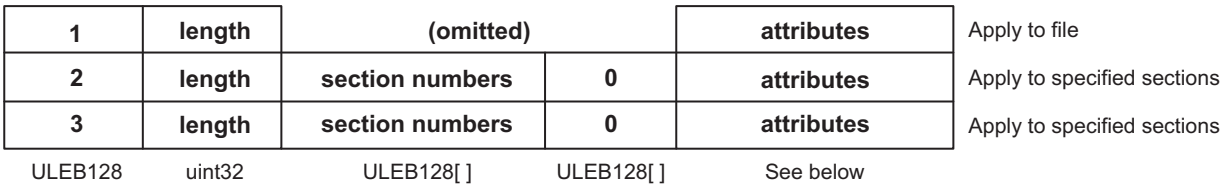


The length field specifies the length in bytes of the entire subsection. The vendor name “mspabi” is reserved for ABI-specified attributes. The format and interpretation of vendor data in other subsections is vendor-specific.

13.1 MSP430 ABI Build Attribute Subsection

Attributes that are specified by this ABI are recorded in the subsection with the vendor string mspabi. 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 mspabi 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:



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.

13.2 MSP430 Build Attribute Tags

OFBA_MSPABI_Tag_ISA (=4), ULEB128

This tag specifies the MSP430 ISA(s) that can execute the instructions encoded in the file. The following values are defined:

0 No ISA specified
1 MSP430
2 MSP430X

In order to link, all the object files in a build must have the same ISA tag.

OFBA_MSPABI_Tag_Code_Model, (=6), ULEB128

0 None
1 Small code model
2 Large code model

This tag specifies the code model used.

In order to link, all the object files in a build must have the same code model. The small code model requires the use of the small data model. The MSP430 supports only the small code model.

OFBA_MSPABI_Tag_Data_Model, (=8), ULEB128

0 None
1 Small data model
2 Large data model
3 Restricted large data model

This tag specifies the data model used. The small code model requires the use of the small data model. The MSP430 supports only the small data model.

In order to link, all the object files in a build must have the same data model.

OFBA_MSPABI_Tag_enum_size, (=10), ULEB128

0 None
1 Small (char/short)
2 Integer (default)
3 Don't care

This tag specifies the minimum container size used for enum types. In order to link, all the object files in a build must specify compatible enum size attributes.

- **None** - The enum size is not specified for the object file. The compiler considers this value compatible with all values. The user must ensure that individual enum types have a consistent size if referenced in multiple object files.
- **Small** - The smallest container that will hold the enum type is used. This value is only compatible with itself.
- **Integer** - The smallest container used is of type integer. This value is only compatible with itself. This is the default.
- **Don't care** - The object file does not reference any enum type across compilation units. This value is compatible with all values.

Table 28 summarizes the build attribute tags defined by the ABI.

Table 28. MSP430 ABI Build Attribute Tags

Tag	Tag Value	Parameter Type	Compatibility Rules
Tag_File	1	uint32	
Tag_Section	2	uint32	
Tag_Symbol	3	uint32	
OFBA_MSPABI_Tag_ISA	4	ULEB128	The ISA, code model, and data model tags cannot be mixed across object files.
OFBA_MSPABI_Tag_Code_Model	6	ULEB128	The ISA, code model, and data model tags cannot be mixed across object files.
OFBA_MSPABI_Tag_Data_Model	8	ULEB128	The ISA, code model, and data model tags cannot be mixed across object files.
OFBA_MSPABI_Tag_enum_size	10	ULEB128	See attribute description.

14 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 7 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.

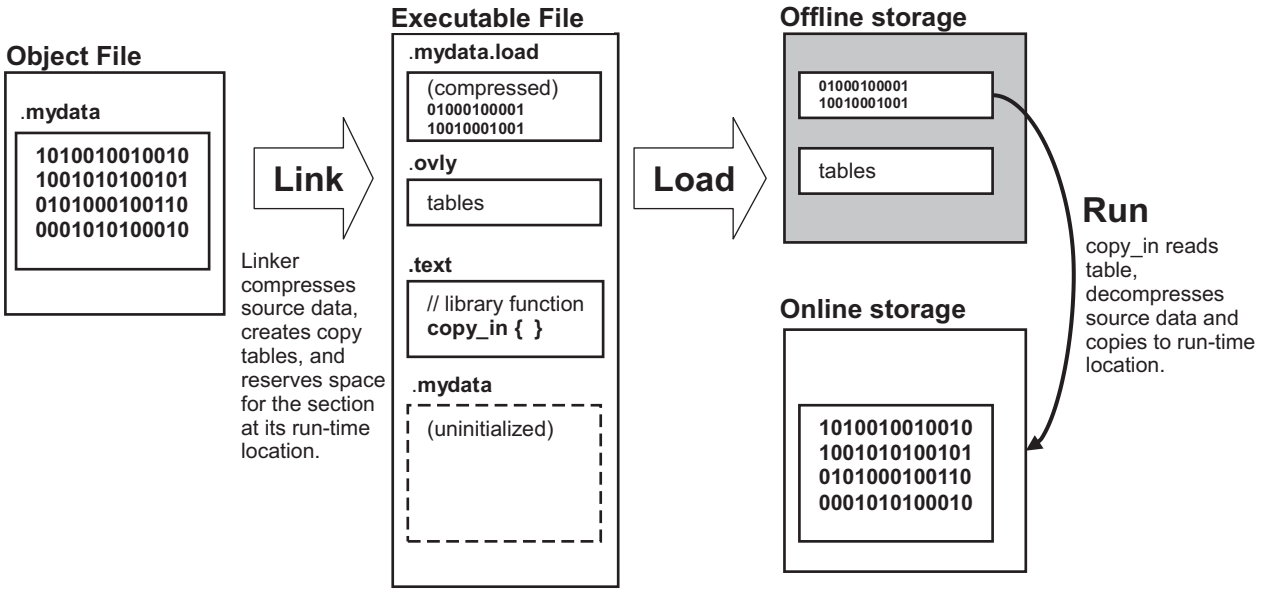


Figure 7. Copy Table Overview

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 14.3.
- **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.

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

14.1 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. The format of the COPY_RECORD structure depends on the code and data model being used.

For the small data model and small code model:

```
typedef struct
{
   void  * load_addr; /* 16-bit pointer */
   void  * run_addr;  /* 16-bit pointer */
   uint16  size;
} COPY_RECORD;
```

For the small data model and large code model:

```
typedef struct
{
   uint32  load_addr; /* 32-bit storage for data or code pointer */
   uint32  run_addr;  /* 32-bit storage for data or code pointer */
   uint32  size;
} COPY_RECORD;
```

For the large (or restricted) data model and large code model:

```
typedef struct
{
   void  * load_addr; /* 20-bit pointer */
   void  * run_addr;  /* 20-bit pointer */
   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 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.
- 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.

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:

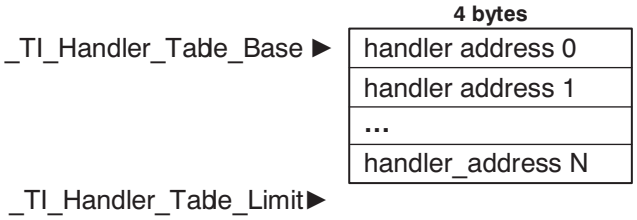


Figure 8. Handler Table Format

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 1 provides a reference implementation of the copy_in function:

Example 1. 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);
      }
   }
}
```

14.2 Compressed Data Formats

Abstractly, compressed source data has the following format:

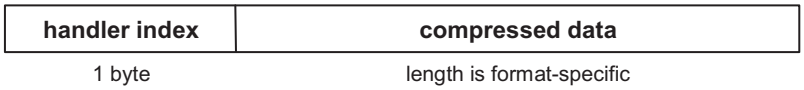


Figure 9. Compressed Source Data Format

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).

14.2.1 RLE

The data following the 8-bit index is compressed using run length encoded (RLE) format. The MSP430 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.

14.2.2 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 in the RTS source code for details on the format.

14.3 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 14. The initialization mechanism is conceptually similar to copy tables, but differs slightly in the details.

Figure 10 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.

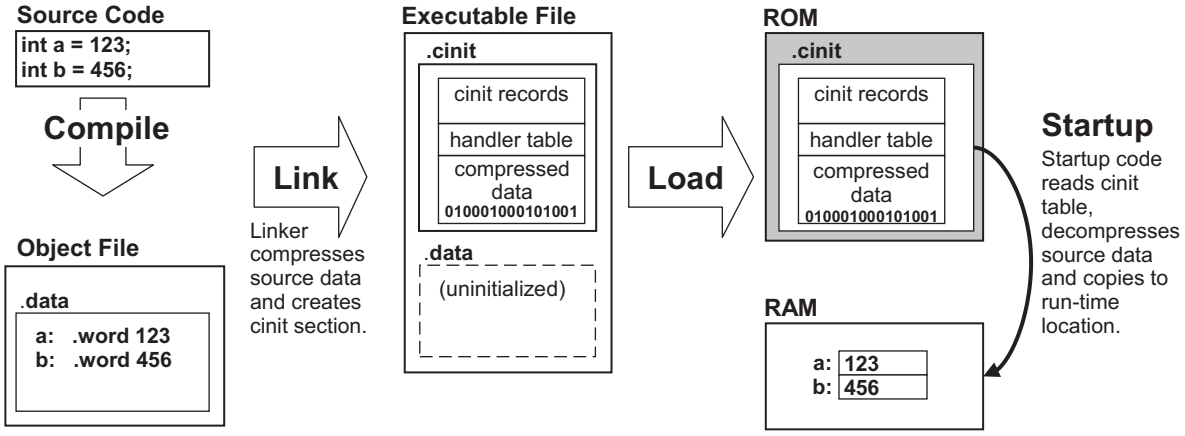


Figure 10. ROM-Based Variable Initialization Via cinit

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 14.1. 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 11 is a schematic depiction of the .cinit section.

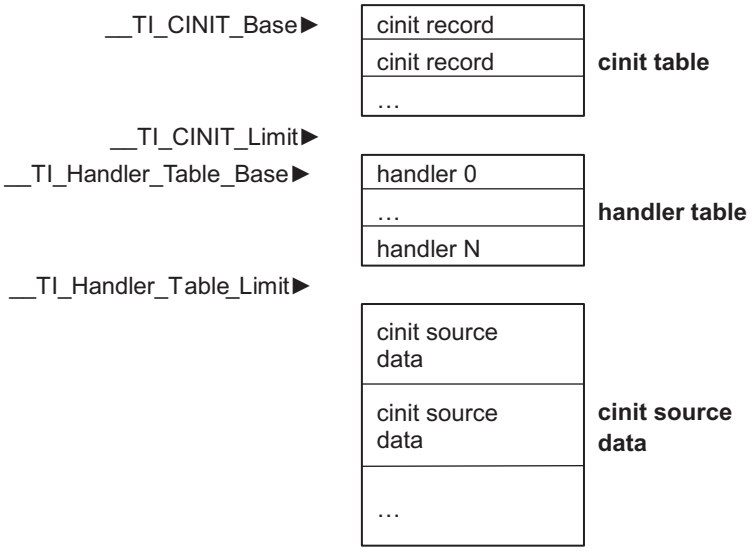


Figure 11. The .cinit Section

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.

The format of the CINT_RECORD structure depends on the code and data model being used.

For the small data model and small code model:

```
typedef struct  {
   void  * source_data; /* 16-bit pointer */
   void  * dest;        /* 16-bit pointer */
} CINIT_RECORD;
```

For the small data model and large code model:

```
typedef struct  {
   uint32  source_data; /* 32-bit storage for data or code pointer */
   uint32  dest;        /* 32-bit storage for data or code pointer */
} CINIT_RECORD;
```

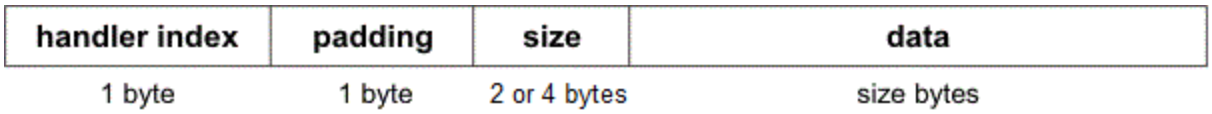
For the large (or restricted) data model and large code model:

```
typedef struct  {
   void  * source_data; /* 20-bit pointer */
   void  * dest;        /* 20-bit pointer */
} 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.

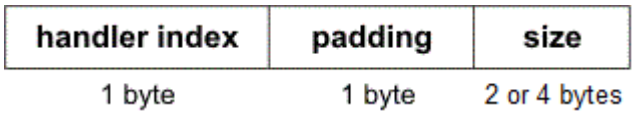
The source data has the same format as compressed copy table source data (see Section 14.1), and the handlers have the same interface. In addition to the RLE and LZSS formats, there are two additional formats defined for cinit records: uncompressed, and zero-initialized.

- 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:



  The encoded data includes a size field, which is aligned on the next 2-byte boundary following the handler index. The size of the "size" field depends on the memory model. For the small code and small data model, the size is 2 bytes. In all other memory model combinations, the size is 4 bytes. 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:



  The size field is aligned on the next 2-byte boundary following the handler index. The size of the "size" field depends on the memory model. For the small code and small data model, the size is 2 bytes. In all other memory model combinations, the size is 4 bytes. 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.

15 Revision History

Table 29 lists changes made since earlier versions of this document were published.

Table 29. Revision History

	Location	Additions / Modifications / Deletions
SLAA534A	Section 3.3.7, Section 3.3.8, Section 3.4, and Section 3.5,	Structures and unions are always passed and returned by reference, no matter their size.

IMPORTANT NOTICE AND DISCLAIMER

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

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

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

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