Application Report
SPRAB90–January 2010

The TMS320C6000 EABI Migration Guide

C6x CGT v7.0 Development Team

ABSTRACT

This document describes the changes which must be made to existing COFF ABI libraries and applications to add support for the new EABI. This is not an overview of EABI; only those details needed for migration are described here.

This document's audience is object library vendors and developers who have been supporting COFF and wish to migrate their code base to ELF.

Contents

List of Figures

List of Tables

1 The C6000 EABI

C6000 code generation tools version 7.0 introduces support for a new ELF-based ABI to support new features such as shared object files. This document does not describe ELF or the C6000 EABI, nor does it describe the new features available only in EABI. This document is focused on migration of COFF ABI applications to EABI and producing code which works equally well with both COFF ABI and EABI.

The details of the C6000 EABI can be found in *The C6000 Embedded Application Binary Interface Application Report* (SPRAB89).

Documentation for features mentioned can be found in the *TMS320C6000 Optimizing C Compiler User's Guide* (SPRU187, revision P or later) and the *TMS320C6000 Assembly Language Tools User's Guide* (SPRU186, revision R or later).

2 Migration Strategy

There are several strategies for dealing with the migration of code from COFF to ELF. The recommended strategy is to update all code bases to work for both COFF and ELF.

2.1 Will COFF Support Be Eliminated?

ELF and EABI will eventually completely displace COFF and COFF ABI; however, COFF will continue to be supported for the foreseeable future. The ELF format is necessary to add certain new features such as dynamic shared objects, so support for such features cannot be added to COFF. While COFF will not be upgraded, it will continue to be fully supported.

2.2 Should an Existing COFF Program Be Converted to ELF?

A working COFF program need not be converted to ELF unless ELF-only features such as dynamic linking are needed. COFF will continue to be supported for the foreseeable future. This strategy is best if the program is self-contained and will not share code with ELF projects.

Library code which will be reused in a later ELF project may need adjustment to work for both COFF and ELF.

2.3 Distribute Libraries in Both COFF and ELF Formats

Library vendors are strongly encouraged to distribute both COFF and ELF versions of each library. Libraries which are not available in both formats will be unusable to a significant portion of the customer base. For portably-written C code, the effort to support both COFF and ELF is minor, and for assembly code is typically a matter of renaming global symbols using conditional compilation.

2.4 Supporting Both COFF and ELF

By using conditional compilation judiciously, it is easy to make code work with both COFF and ELF; however, two sets of object files will be necessary, as linking COFF and ELF object files together is not allowed.

2.4.1 Predefined Symbol: __TI_EABI__

Both the compiler and assembler pre-define the symbol __TI_EABI__ to indicate that the source is being compiled under EABI. This option is defined when the --abi=eabi option is specified. Where the C code or assembly code cannot be written in a way that works for both COFF ABI and EABI, use this symbol to conditionally compile the appropriate version of the code.

```
#if defined(__TI_EABI__)
static char abi[] = "EABI";
#else
static char abi[] = "COFF ABI";
#endif
printf("ABI used: %s\n", abi);
```

2.4.2 Dealing With COFF-Only Object Libraries

To convert an object file from COFF ABI to EABI, it is strongly recommended that you have access to at least the assembly code so that it can be appropriately modified and reassembled. If you do not have source code, such as the case when you only have an object library from a vendor, the best choices are to either leave the application as a COFF ABI application, or to request the vendor release an EABI version. There is no tool support for converting a COFF object file to an ELF object file; reverse-engineering the assembly code by using a disassembler is error-prone and could violate licensing agreements for some packages.

3 C and C++ Implementation-Defined Language Changes

Programs written entirely in C or C++ will have the easiest migration path. Portably written C and C++ code will probably not need any changes at all, so such code can be shared unmodified between COFF and ELF projects.

Maximally portable C and C++:

- Does not rely on exact sizes of types beyond what the C standard guarantees
- Does not assume a particular bit-field layout
- Does not assume a particular enum type size
- Does not use intrinsics
- Does not use asm("") statements

If your code avoids these non-portable assumptions, the code may be reused unmodified without inspection. Code which does make one of these assumptions will need to be examined to determine if the code will behave differently for EABI and COFF ABI. This section describes where EABI and COFF ABI differ with regard to C and C++ language features.

3.1 The long int Type is 32 Bits

The long int (or long) integer type is 32 bits wide in the EABI model, whereas it is 40 bits wide in the COFF ABI model. This change has far-reaching consequences.

3.1.1 No Native 40-Bit Integer Type

C6000 EABI does not support a native 40-bit integer type. This means that types related to int40_t are not available under EABI, and expressions involving types related to long can have different results under EABI. Automatic backward compatibility is not possible because each occurrence must be individually examined to determine the appropriate change for EABI.

Uses of long in C code in EABI will necessarily use either a 32-bit type (int or long) or a 64-bit type (long long). If a 32-bit type is appropriate, the code need not be changed. However, if a 40-bit or wider type is required, the code must be changed to use a 64-bit type.

Code using C6x intrinsics involving a long type will need to be rewritten using the new 40-bit arithmetic intrinsics described in Section 3.1.6.

3.1.2 Examine the Intent of Each Declaration and Expression Involving Type long

The predominant use of long in the industry is to indicate that the object should be at least 32 bits. C6000 COFF ABI defines long to be 40 bits, which conforms to the C standard, but can cause problems in programs that were written assuming that long was exactly 32 bits. For general-purpose code written without knowledge of C implementations where long is not exactly 32 bits, EABI can be used without modifying the code. However, for programs written specifically for C6000, it's not unlikely that the user may be relying on long being 40 bits to gain extra precision. However, it is also not unlikely that the type long was used reflexively to mean a 32 bit type, as this behavior is pervasive in industry. The intent of the programmer must be considered for each occurrence, and the types adjusted accordingly.

The best practice for portable code is to use intention-revealing names, for which the types defined in stdint.h are particularly useful. Using these types, it is possible to write code which uses the correct types for both COFF ABI and EABI, and the fact that the types are correct will be obvious to someone who reads the code.

3.1.3 Use Cases for Declarations Involving long

Table 1 shows typical use cases for long, along with the suggested type to use in code shared between COFF ABI and EABI:

Table 1. Typical long Use Cases

Bits Needed	Storage size a concern?	Execution speed a concern?	Native Types	stdint.h Types
At least 32 bits	No	No	long(1) or int(2)	int32_t, int_least32_t, or int_fast32_t
At least 32 bits	Yes	--	int[2]	int_least32_t
At least 32 bits	--	Yes	int[2]	int_fast32_t
Exactly 32 bits	--	--	int[2] may be used, but int32_t is preferred	int32_t
At least 40 bits	--	No	COFF: long[1] EABI: long long	COFF: int40_t, int_least40_t, or int_fast40_t EABI: int64_t, int_least64_t, or int_fast64_t
At least 40 bits	--	Yes	COFF: long[1] EABI: long long	COFF: int_fast40_t EABI: int_fast64_t
Exactly 40 bits (needs saturation and truncation)	--	--	Not recommended	COFF: int40_t EABI: int64_t plus 40-bit intrinsics

(1) The type long can be left in the code, but it is better to use intention-revealing types from stdint.h

(2) The type int can be used, but this type is not portable to implementations where int is only 16 bits.

Another use common use case is that there is an API in use which requires that the type be long. For instance, standard C library functions like printf may require certain arguments to be of type long. In these instances, continue to use the type long.

3.1.4 No 40-Bit Intrinsics

Because C6000 EABI does not support a native 40-bit integer type, the following intrinsics have different prototypes under EABI. Where COFF ABI uses a 40-bit type, EABI uses a 64-bit type. The EABI prototypes are:

- long long _lsadd(int src1, long long src2)
- long long _lssub(int src1, long long src2)
- long long _labs(long long src)
- int _sat(long long src)
- int _lnorm(long long src)
- unsigned long long _dtol(double src)
- double _ltod(unsigned long long src)
- long long _ldotp2(int src1, int src2)

The following new intrinsics operate on 64-bit integer types. These intrinsics can be used where code was written for COFF ABI using a native 40-bit integer type and the programmer intended to rely on 40-bit saturation and truncation. This allows the compiler to use the more efficient 40-bit arithmetic assembly instructions to perform these operations.

- long long _add40_s32 (int src1, int src2);
- unsigned long long _add40_u32 (unsigned src1, unsigned src2);
- long long _add40_s40 (int src1, long long src2);
- unsigned long long _add40_u40 (unsigned src1, unsigned long long src2);
- int _cmpeq40 (int src1, long long src2);
- int _cmpgt40 (int src1, long long src2);
- int _cmplt40 (int src1, long long src2);
- int _cmpltu40 (int src1, long long src2);
- int _cmpgtu40 (int src1, long long src2);
- long long _mov40 (long long src);

- long long _neg40 (long long src);
- long long _labs40 (long long src);
- long long _shl40 (long long src1, int src2);
- long long _shr40 (long long src1, int src2);
- unsigned long long _shru40 (unsigned long long src1, int src2);
- unsigned long long _shl40_s32 (int src1, int src2);
- long long _sub40_s32 (int src1, int src2);
- unsigned long long _sub40_u32 (unsigned src1, unsigned src2);
- long long _sub40_s40 (int src1, long long src2);
- long long _zero40 (src);

These are examples of intrinsic usage:

Example 1. COFF ABI (1)

```
#if defined(_TMS320C6X) && !defined(__TI_EABI__)
extern int a;
extern long b;
long result = _lsadd(a, b);
#endif
```

Example 2. EABI (1)

```
#if defined(_TMS320C6X) && defined(__TI_EABI__)
extern int a;
extern long long b;
long long result = _lsadd(a, b);
#endif
```

Example 3. COFF ABI (2)

```
#if defined(_TMS320C6X) && !defined(__TI_EABI__)
extern int a;
extern long b;
long result = a + b;
#endif
```

C6000 recognizes this case and generates an efficient 32x40 ADD.

Example 4. EABI (2)

```
#if defined(_TMS320C6X) && defined(__TI_EABI__)
extern int a;
extern long long b;
long long result = _add40_s40(a, b);
#endif
```

Without using this intrinsic, the compiler may not be able to recognize the efficient 32x40 ADD.

3.1.5 C Declarations of Assembly Functions Involving long int

A C-callable assembly function written for COFF ABI which accepts or returns a 40-bit value in a register pair will have been declared in the C code with a prototype involving the long int type. Prototypes for such functions will need to be changed in EABI.

Any 40-bit values are stored in 64-bit containers, either a 64-bit register pair or a 64-bit double word in memory, including when passed to or returned from a function.

Because the container for a 40-bit value is the same size and alignment as that for a 64-bit value, you can usually change the prototype for such functions to use the long long int type instead of long int without altering the function's behavior. In the unusual case that the assembly function does not sign-extend returned 40-bit values into the 64-bit return register pair, the calling function will need to explicitly handle the sign extension in EABI.

- COFF ABI C Source

```
extern long blue_fish(long l_arg);

long red_fish()
{
    return blue_fish(0x1234);
}
```

- Compiler Generated Assembly

```
_red_fish:
;----------------------------------------------------------------------
;   6 | return blue_fish(0x1234);
;----------------------------------------------------------------------
        CALLRET .S1     _blue_fish        ; |6|
        ZERO    .L1     A5                ; |6|
        MVK     .S1     0x1234,A4         ; |6|
        NOP             3
```

With long integer types of size 32-bits under the EABI, you would need to update the C source as follows:

- EABI C Source

```
extern long long blue_fish(long long l_arg);

long long red_fish()
{
   return blue_fish(0x1234);
}
```

- Compiler Generated Assembly

```
red_fish:
;----------------------------------------------------------------------
;   6 | return blue_fish(0x1234);
;----------------------------------------------------------------------
        CALLRET .S1     blue_fish        ; |6|
        ZERO    .L1     A5                ; |6|
        MVK     .S1     0x1234,A4         ; |6|
        NOP             3
```

NOTE:

- The compiler does not add leading underscore to C names (red_fish and blue_fish) under the EABI (see Section 4.1). Otherwise, the generated assembly is the same under EABI vs. COFF ABI.
- Need to update declaration of blue_fish()
- Need to update return type of red_fish() per use of return value from blue_fish()

One way to write the above C source that would be compatible under both the COFF ABI and EABI models is to write the following:

- C Source

```
#include "int40.h"

extern INT40 blue_fish(INT40 l_arg);

INT40 red_fish()
{
   return blue_fish(0x1234);
}
```

- Header File

```
#include <stdint.h>
#if defined(__TI_EABI__) && !defined(__TI_40BIT_LONG__)

typedef long long           INT40;
typedef unsigned long long UINT40;
#else
typedef long                INT40;
typedef unsigned long long UINT40;
#endif
```

3.1.6 Backwards Compatibility: --long_precision_bits

Source code which makes the assumption that long is 40 bits must eventually be changed.

As a temporary transition aid, the compiler supports the --long_precision_bits=40 option. Type long becomes 40 bits, so expressions involving long will have the same value as they do for COFF ABI the original intrinsics are available. int40_t is available.

This is strictly a transitional aid, and will be deprecated. This option should only be used for relatively self-contained projects. Use the option only during migration to EABI. This option effectively creates a distinct, incompatible ABI

Do not distribute libraries compiled with this option, because vendor support for this ABI will be very limited.

Pre-built RTS libraries compatible with long_precision_bits=40 EABI mode are not included with the product. The user must explicitly compile them to use them.

Object files produced using this option are not compatible with the libraries that come with the toolset. The libraries must be rebuilt using long_precision_bits=40. Instructions for rebuilding the library can be found in the *TMS320C6000 Optimizing C Compiler User's Guide*, in the *Using Run-Time-Support Functions and Building Libraries* chapter.

Predefined symbols __TI_32BIT_LONG__ __TI_40BIT_LONG__

The compiler defines one of two preprocessor symbols depending on the mode. The other symbol is left undefined.

- __TI_32BIT_LONG__ is defined in EABI (--abi=eabi) when --long_precision_bits=40 is not specified.
- __TI_40BIT_LONG__ is always defined in COFF ABI, and also in EABI when --long_precision_bits=40 is specified.

Instead of assuming that long is 32 bits in EABI, these macros can be used:

```
#if defined(_TMS320C6X) && defined(__TI_40BIT_LONG__)
extern int a;
extern long b;
long result = _lsadd(a, b);
#else
extern int a;
extern long long b;
long long result = _lsadd(a, b);
#endif
```

3.2 Bit-Field Layout

The declared type of a bit-field is now the container type. This means that some structures will have a different layout in COFF ABI and in EABI.

For code that must be portable between COFF ABI and EABI, bit-fields should not be used. If they must be used, the bit-field may need to declared with distinct conditionally-compiled code.

3.2.1 C and C++ Standard Requirements for Bit-Fields

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.

C89, C99, and C++ 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.

The TI compiler supports using any integral type as the declared type in both C and C++, but only in EABI. For COFF ABI, bit-fields must have declared type int, unsigned int, or signed int.

3.2.2 EABI Layout Scheme

For EABI, the declared type is also used as the container type. This has two major consequences:

- The containing structure will be at least as large as the declared type
- If there is not enough unused space in the current container, the bit-field will be aligned to the next container.

If a 1-bit field has declared type int, the EABI layout will allocate an entire int container for the bit-field. Other fields can share the container, but each field is guaranteed to be stored in some container exactly the size of the bit-field.

Example 1 (P stands for padding):

```
struct S { int a:1; };

          11111111112222222223
01234567890123456789012345678901
aPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP (one 32-bit container)
```

Example 2:

```
struct S { int a:1; int b:1; };

          11111111112222222223
01234567890123456789012345678901
abPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP (one 32-bit container
```

Example 3:

```
struct S { char a:7; char b:2; };

          111111
0123456789012345
aaaaaaaPbbPPPPPP (two 8-bit containers)
```

Example 4:

```
struct S { char a:2; short b:15; };

          11111111112222222233
01234567890123456789012345678901
aaPPPPPPPPPPPPPPbbbbbbbbbbbbbbbP

(one 8-bit container, one 8-bit pad, and one 16-bit
container)
```

Further reading on the bit-field layout can be found in the IA64 C++ ABI specification (http://www.codesourcery.com/public/cxx-abi/abi.html).

3.2.3 COFF ABI Layout Scheme

The COFF ABI scheme uses a different strategy. It starts by using the smallest possible container, and will grow the current container if growing it will allow the bit-field to be allocated at the current position.

Example 1:

```
struct S { int a:1; };

01234567
aPPPPPPP (one 8-bit container)
```

Example 2:

```
struct S { int a:1; int b:1; };

01234567
abPPPPPP (one 8-bit container)
```

Example 3:

```
struct S { char a:7; char b:2; };

          111111
0123456789012345
aaaaaaabbPPPPPPP (one 16-bit container)
```

Example 4:

```
struct S { char a:2; short b:15; };

          11111111112222222233
01234567890123456789012345678901
aabbbbbbbbbbbbbbbPPPPPPPPPPPPPPP (one 32-bit container)
```

3.2.4 Compatibility Impact of EABI

EABI can produce a layout that is not quite the same as it would be in COFF ABI. Programs which rely on using bit-fields for precise data layout, such as for reading a binary file or setting bits in a status register should be examined for compatibility. Such test cases may need to use conditional compilation to change the declared types of bit-field definitions. However, many existing test cases will be unchanged.

Incompatibilities fall into one of two categories: structures that are larger than expected, and bit-fields that are at different positions.

Structures can be larger with EABI if they contain bit-fields with mostly unused bits. If the structure needs to use the smaller size that would have been used with COFF ABI, the declared type needs to be changed to a type of the desired size, such as char.

Bit-fields can usually only be at a different position in cases when there is enough space left over in the current container to fit the field width of the next bit-field, but not a properly-aligned object of the declared type. The narrower the declared type on a bit-field, the more likely there will be an incompatibility. Declaring all bit-fields with an int-sized type (as is typical of code written for C89), will minimize incompatibility of bit-field position.

3.2.5 Access Type

For efficiency, the compiler may access a bit-field with a type which does not match either the declared type or the container type. The declared type and container type are strictly used to determine bit field packing and alignment. The type used by the CG to actually load the bit-field is the access type. It can be a narrower type, computed from the size and offset of the bit-field. For instance, in the following EABI example, the container type is 32 bits, but the bit-field will be loaded using an 8-bit access:

```
struct S { int :8; int bf:8; };
```

For EABI, the compiler will not use a narrower type for volatile bit-fields (bit fields declared with a volatile-qualified type); it will instead use exactly the declared type.

3.3 Enumerated type size

Many enumeration types have members with values that are small enough to fit into integer types smaller than int. COFF ABI always uses int-sized containers to store variables of such enumeration types. EABI will use types smaller than int when possible. For C++ code, both COFF ABI and EABI will use integer types wider than int for enumeration types with values larger than will fit into int.

3.4 asm() Statements

The contents of asm() statements are really assembly code, and need to be changed as shown in Section 4.

4 Assembly Code Changes (C and C++ ABI Changes)

The C ABI is how the compiler expresses C code programs in assembly language. Assembly code that defines a C-callable function or calls a C function must conform to the ABI. This section describes changes which must be made to assembly code due to the changes made by EABI to the way C and C++ features are implemented in assembly code.

The changes that will be necessary to existing assembly code are primarily limited to places where the assembly code interfaces with C or C++ code. Assembly functions which do not interface with C or C++ code directly do not need to be changed.

4.1 COFF Underscore Name Mangling

COFF ABI uses underscores to keep the assembly code name space and the C code namespace separate. The C compiler prepends an underscore to every externally-visible identifier so that it will not collide with an assembly object with the same name. We call this the *COFF underscore*.

This source code:

```
int x;
int func(int y) { }
```

Becomes in COFF ABI:

```
        .bss _x, 4, 4
_func:
```

EABI does not add the COFF underscore. This is a generic ELF requirement. The user is responsible for making sure user-defined names don't collide. Assembly code which is intended to work for both COFF ABI and EABI will need to handle the difference in mangling.

```
int x;
int func(int y) { }
```

Becomes in EABI:

```
        .bss x, 4, 4
_func:
```

4.2 Removing the COFF Underscore

COFF ABI adds a leading underscore to C and C++ symbols to prevent name collisions with symbols defined in hand-coded assembly, but EABI does not add this underscore. When using COFF ABI, a function named red_fish written in C will produce a function entry label with the name _red_fish in the assembly source. Under the EABI, the name of the function as it appears in the assembly source will be exactly as it appears in the C code, so the function entry label for red_fish will be red_fish.

Functions and variables may be defined in assembly code and used in C code. To use functions and variables in a hand-coded assembly file from a COFF ABI program in EABI, the symbol label needs to be changed, or augmented with a second label. There are several approaches to this issue.

4.2.1 Conditional Redefinition Method

The preferred solution that will be compatible with both the COFF and EABIs is to replace the COFF ABI mangled name with an EABI C name using an .asg assembler directive. For example, a function red_fish called from C will have a definition in the COFF ABI assembly code with a function entry label named _red_fish. Insert a conditional .asg directive in front of the definition as follows:

```
            .if     __TI_EABI__
            .asg    red_fish, _red_fish
            .endif

            .global _red_fish
_red_fish:
            <start of function>
```

In the above example, all instances of _red_fish will be replaced with red_fish due to substitution symbol expansion. The assembler will define the label, red_fish and make it visible externally via the .global directive.

4.2.2 Double Label Method

Another easy solution is to provide two labels, one providing the COFF ABI mangled name, and the other providing the EABI name.

```
           .global _red_fish, red_fish
_red_fish:
red_fish:
            <start of function>
```

A drawback to this solution is that there remains an extra symbol name which might collide with a user-defined name.

4.2.3 Preprocessor Redefinition Method

For projects where the assembly code cannot be readily modified, the assembler's substitution symbol mechanism can be used to redefine individual symbols. The technique is to create either a C source header file or an assembly include file which redefines each symbol. This include file can then be implicitly included in an assembly file by using the --include_file assembler option.

4.2.4 Backward Compatibility: --strip_coff_underscore

For projects where the assembly code cannot be readily modified, the compiler provides the --strip_coff_underscore option which instructs the assembler to translate COFF ABI mangled external symbol names to EABI by removing one leading underscore. This provides some assistance in migrating assembly source files to the EABI by reducing the need to alter your assembly source files.

For many programs, just using this option will be sufficient to use COFF ABI assembly code in EABI programs. However, name collisions are possible if the assembly source already has two external symbols with names that collide when the underscore is removed, such as sym and _sym.

Further changes may still be necessary; this option does not handle symbol names appearing in C asm() statements or linker command files. Those cases must be modified manually.

It is very likely that a particular hand-coded assembly file will only need the COFF underscore removed to be valid for EABI. For assembly files of this nature, compiler option --strip_coff_underscore instructs the assembler to strip the underscore from every external identifier.

For example, to use this source code:

```
main.c:
        int main() { red_fish(); }
fish.asm:
                .global _red_fish
        _red_fish: ...
```

For COFF ABI enter this on the command line:

```
cl6x main.c fish.asm -z
```

For EABI enter this on the command line:

```
cl6x main.c fish.asm --abi=eabi --strip_coff_underscore -z
```

4.3 C++ Name Mangling

The compiler uses name mangling to encode into the name of C++ functions the types of its parameters so that the linker can distinguish overloaded functions.

COFF ABI and EABI use different name mangling schemes for C++ functions, so assembly code which refers to the mangled names directly will need to be changed to use the EABI mangling.

This is an example of difference in name mangling:

	`int func(int);`	`int func(float);`
COFF ABI	`_func__Fi`	`_func__Ff`
EABI	`_Z4funci`	`_Z4funcf`

Direct references to mangled C++ names are unlikely unless the output assembly file from compiling a C++ file was captured and hand-modified. The best migration path is to just re-compile the original C++ file. If the hand-modifications are too extensive to do this, the fastest method to find the EABI mangled names is to re-compile the original C++ file and examine the generated assembly code to see the EABI mangled names.

Pass the --abi=elfabi option to dem6x to demangle EABI C++ names.

4.4 Structures Passed or Returned By Value

In COFF ABI, all structs that are passed or returned by value in C code are transformed by the compiler so that in the generated assembly code they are passed by reference. The compiler puts the struct in a temporary location and passes a pointer to this temporary in place of the struct.

In EABI, small structures (64 bits or smaller) passed or returned by value in C code are passed or returned by value in the generated assembly code, either in a register or on the stack as appropriate. Larger structures are passed by reference as in COFF ABI.

C-callable assembly functions that accept, return, or pass small structures by value need to be re-written to follow this convention.

4.5 Legacy .cinit in Assembly Source

The COFF ABI uses the .cinit mechanism to initialize global variables. This is intended to be used only by the compiler, but some hand-coded assembly source encodes variable initialization with hand-encoded .cinit tables. This will work under COFF ABI as long as the encoding is correct. However, this method will not work in EABI, because it uses direct initialization instead, which means the linker creates all .cinit records.

The recommended migration path is to rewrite the .cinit initialization as direct initialization and let the linker handle creating the initialization record. For example, the following .cinit record can be rewritten as shown:

```
glob:   .usect ".far", 8, 4 ; 8 byte object aligned to 4 bytes in uninitialized section ".far"
        .sect ".cinit"
        .align 8
        .field 8, 32     ; length in bytes
        .field glob, 32  ; address of memory to initialize
        .field 2, 32     ; initialize first word to 2
        .field 3, 32     ; initialize second word to 3

        .sect ".fardata", RW ; 8 byte object in initialized section ".fardata"
        .align 4
glob:   .field 2, 32     ; directly initialize first word to 2
        .field 3, 32     ; directly initialize first word to 3
```

For more information on using direct initialization, see the *TMS320C6000 Optimizing C Compiler Tools User's Guide*.

4.6 Legacy STABS Directives in Assembly Source

Some COFF ABI assembly code can contain STABS (COFF debug) directives, particularly if the assembly code was originally generated by the compiler.

ELF does not support STABS, and the assembler will give an error message if the input file contains STABS directives. To reuse the file for EABI, strip out all of the STABS directives.

Example STABS directives: .file, .func, .block, .sym

4.7 DP-Relative Data Pointers

The compiler places some data, typically non-aggregate objects, in the .bss section. This section is intended to be accessed by DP-relative addressing, also called *near* addressing.

All of the objects in the .bss section need to be addressable through the limited offset range from the DP register, so the linker takes care to collect all the input .bss sections into a contiguous output .bss section. This output section may be placed anywhere in memory. The address of the output section is placed in the DP register by the bootstrap routine.

In COFF ABI, the .bss section is the only near section. The symbolic name for the address of this section is $bss, and this is the symbol used to initialize DP. In EABI, the .bss section is not the only near section, so $bss may not accurately reflect the proper DP initialization value. Instead, the symbol used by EABI to initialize the DP register is __TI_STATIC_BASE. References to $bss in COFF ABI hand-coded assembly need to be changed to references to __TI_STATIC_BASE for EABI.

Because the symbol $bss may be used frequently in certain idioms where the address of a variable is loaded into a register, in EABI mode the assembler will recognize some uses of $bss and automatically change them to relocations involving __TI_STATIC_BASE. In COFF ABI, expressions involving $bss would have been handled with a relocation expression. A relocation expression is a series of stack-machine like instructions that dictate how to compute the value of a relocatable expression. The TI ELF object format does not support relocation expressions (see Section 6.1).

DP-relative data accessed with a *near* access works the same in EABI as COFF ABI:

```
LDW *+DP(x), A4
```

However, code using a COFF ABI *far* idiom must be changed:

```
MVK (x-$bss), A4
ADD DP, A4, A4
LDW *A4, A4
```

to this:

```
MVK $DPR_byte(x), A4
ADD DP, A4, A4
LDW *A4, A4
```

The assembler will recognize the COFF idiom above and make the change automatically, but only for the following expressions. The available operators are $DPR_word, $DPR_hword, and $DPR_byte. Refer to the *TMS320C6000 Assembly Language Tools User's Guide* (SPRU186) for further details.

```
(x - $bss)       --> $DPR_byte(x)
(x - $bss) / 2   --> $DPR_hword(x)
(x - $bss) >> 1  --> $DPR_hword(x)
(x - $bss) / 4   --> $DPR_word(x)
(x - $bss) >> 2  --> $DPR_word(x)
```

Other references to $bss, such as in the linker command file, will be changed to be references to __TI_STATIC_BASE. In this way, the assembler and linker will automatically change most references to $bss to __TI_STATIC_BASE. Any other uses of $bss, such as in arithmetic expressions involving $bss, will need to be changed by the user.

4.8 Run-Time-Support Library Helper Functions

The library contains some *helper* functions to perform complicated operations for certain high-level language features. It is not expected that hand-coded assembly code would call these functions, but it is possible, particularly if the output of the compiler is tweaked by hand and transformed to an assembly input file. These helper functions have different names in EABI. If the assembly code directly calls a library helper function, the code will need to use the new name for the function. The easiest way to deal with these function is to use the assembler --include_file option to include a list of assembler defines to change the names of the old functions.

For example, create a C header file (coff_to_elf_helpers.h)

```
#define __divi __c6xabi_divi
#define __divu __c6xabi_divu
```

Include this file in another header (coff_to_elf_helpers.i):

```
.cdecls C, LIST, "coff_to_elf_helpers.h"
```

And include this file at the beginning of every assembly file:

```
cl6x --include_file=coff_to_elf_helpers.i
```

Attached to this paper you should find header files with redefinitions for all of the C ABI run-time support functions from the COFF ABI name to the EABI name. For more information, see Appendix C.

5 Linker Command File Changes

When porting a COFF ABI application to EABI, the most likely place the user will need to make a change is the linker command file. The linker supports linker command file preprocessing. See the *TMS320C6000 Assembly Language Tools User's Guide* (SPRU186).

5.1 EABI Sections

EABI re-uses most compiler-generated section names used by COFF ABI, and also introduces new section names. Each section needs to be allocated to appropriate memory. See Appendix B for all the sections generated by the toolset.

5.1.1 DP-relative Data Sections

EABI introduces the following DP-relative data sections:

.rodata	initialized read-only data
.neardata	initialized near read-write data

These sections are similar to .bss, except that they are initialized (contain data in the object file). The three DP-relative sections must be contiguous, which is most easily accomplished by using GROUP in the linker command file:

```
GROUP (NEAR_DP_RELATIVE)
{
       .neardata
       .rodata
       .bss
}>BMEM
```

5.1.2 Read-Only Sections

EABI introduces the following read-only sections

.init_array	data, used to register C++ global variable constructors
.c6xabi.exidx	data, index table for C++ exception handling
.c6xabi.extab	data, unwinding instructions for C++ exception hand

The data section .init_array serves the same purpose .pinit does for COFF ABI. EABI does not use the name .pinit.

5.1.3 Read-Write Sections

EABI introduces the following read-write sections:

.fardata	initialized far data
.far	uninitialized far data

COFF ABI also uses the section name .far for uninitialized far data.

5.2 No Leading Underscores

The symbol names used in linker command files are the names as they appear in object files, which for COFF means the mangled names. For EABI, the object file names are the same as the high-level language names, so any reference or definition of a symbol in a linker command file will need to be changed. For instance, to set a symbol to the value of the function main.

COFF ABI:

```
mainaddr = _main;
```

EABI:

```
mainaddr = main;
```

COFF ABI:

```
_symbol = 0x1234;
```

EABI:

```
symbol = 0x1234;
```

5.3 Conditional Linking On by Default

Conditional linking, or dead-code removal, is a process performed by the linker in which the linker creates a reference graph of all input sections that were presented to the linker. If a given input section is marked as a candidate for removal and there are no references to the input section that can be traced to an entry point or interrupt function in the application, then that section will not be included in the link. Conditional linking is enabled by default in ELF.

To disable conditional linking entirely, use the –unused_section_elimination=off option. All sections in all input files will be retained.

To force the linker to retain an individual symbol, use the –-undef_sym=*symbolname* (or -u *symbolname*) option in the linker command file. The --retain=*symbolname* option can also be used to retain a symbol.

To retain a section by name, use --retain=*filename*(*sectionname*). If the section is in a library, use --retain=*libraryname*<*filename*>(*sectionname*). This option allows a wildcard pattern.

The .symdepend directive may also be useful to retain sections with symbols which appear to have no references by informing the linker of an implicit dependence as shown in Example 5.

Example 5. Using the .symdepend Directive to Retain Sections With Unreferenced Symbols

```
        .sect "vanishing_section"
vanishing_symbol:
        .word 1

        .sect "reached_from_c_int00"
        .symdepend vanishing_symbol, myfunc
myfunc:
```

6 Miscellaneous

6.1 Relocation Expressions Are Not Supported

Assembler expression involving two or more relocatable symbols cannot be represented in C6000 ELF object files. Any such expression will need to be rewritten into two or more instructions. The COFF ABI DP-relative idioms recognized by the assembler are the exceptions.

For example, the following will not work if both symbols are resolved at link time:

```
thing_size:   .word (thing_end - thing_begin)
```

6.2 Partial Linking

Relocation entries are not processed during a partial link under the EABI. Relocation entries involving a static base reference will simply be carried forward until the user is ready to create an executable output file with the linker. At that point, the linker will define a value for the __TI_STATIC_BASE symbol that is used in the resolution of any static-base relative relocation that is encountered.

6.3 –symdebug:coff and –symdebug:profile_coff Are Not Supported

These options request the use of STABS debugging information, which is available only for COFF files. ELF files must use DWARF. If there are any STABS debug directives in an assembly file (this typically only happens for assembly code generated by the compiler), these directives must be deleted or conditionally compiled out; the assembler will reject these directives when assembling to an ELF file.

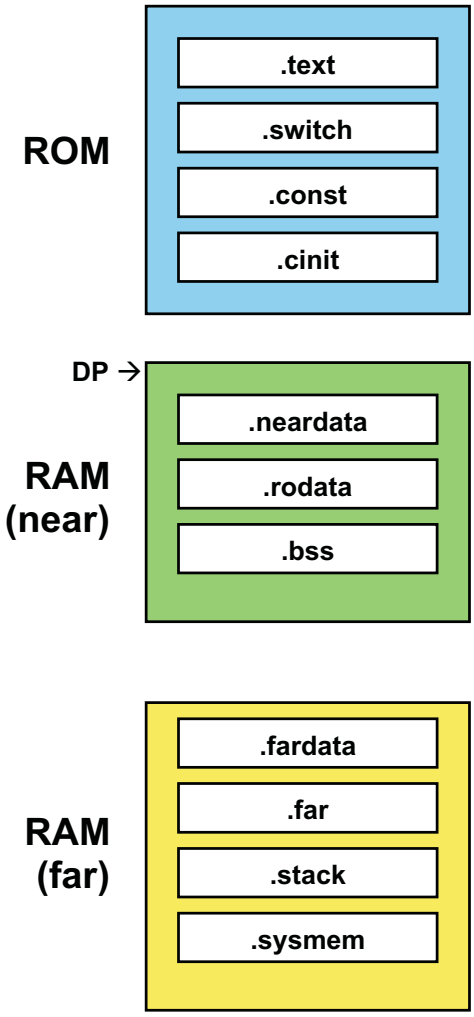
6.4 Symbol Name Changes

The linker defines special symbols that can be referred to in the linker command file or in source code. Some symbols used by COFF ABI were renamed for EABI. A complete list of special symbols supported by the COFF ABI can be found in Appendix B with the corresponding EABI name.

Appendix A C6x EABI Sections

New C6x EABI sections in Figure 1 and Table 2 are in blue.

Figure 1. C6x EABI Sections



Read-Only (RO) Sections	
.text	Code
.switch	Switch tables
.const	Far const data
.cinit	Boot-time init tables

DP-relative Read-Write (RW) Sections	
.neardata	Near RW data (initialized)
.rodata	Near const data
.bss	Near RW data (uninitialized)

Absolute Read-Write (RW) Sections	
.fardata	Far RW data (initialized)
.far	Far RW data (uninitialized)
.stack	Program stack
.sysmem	Dynamic data (heap)

Table 2. Other C6000 EABI Sections

Section	Access	Description
.init_array	RO data	Pre-main constructors (was .pinit)
.name.load	RO data	Compressed image of section name
.c6xabi.exidx	RO data	Index table for exception handling
.c6x.extab	RO data	Unwinding instructions for exception handling
.binit	RO data	Boot-time copy tables
.cio	RW data	Handshaking buffer for host-based stdio
.args	RW data	Command arguments for host-based loader
.ppinfo	RW data	Correlation tables for compiler-based profiling
.ppdata	RW data	Data tables for compiler-based profiling

Appendix B Special Symbols

COFF ABI and EABI define special symbols for management of ABI functionality. Some variables were renamed for EABI. The *COFF ABI name* is the name as it would appear in assembly code or the linker command file. The special symbols are listed below:

Table 3. Special Symbols

COFF ABI Name	EABI Name	Purpose
___binit__ or binit	deprecated	Boot-time initialization
___c_args__	__c_args__	Command-line arguments
___cinit__ or cinit	__TI_CINIT_Base[(1)]	Start of C global variable initializers
___data__	deprecated	Beginning of the .data section
___edata__	--	End of the .data section
___end__	--	End of the .bss section
___etext__	--	End of the .text section
___pinit__ or pinit	__TI_INITARRAY_BASE[(1)]	Start of C++ global object initializers
___text__ or .text	--	Beginning of the .text section
__bss__ .bss or $bss	__TI_STATIC_BASE	Start of DP-relative data
__STACK_SIZE	__TI_STACK_SIZE	Size available for function frame stack
__SYSMEM_SIZE	__TI_SYSMEM_SIZE	Size available for heap allocation
C$$EXIT	C$$EXIT	Special host I/O trap
C$$IO$$	C$$IO$$	Special host I/O trap
__STACK_END	__TI_STACK_END	End of the .stack section

(1) While in COFF ABI .pinit and .cinit are NULL-terminated, in EABI ending addresses of .init_array and .cinit are indicated by the corresponding LIMIT symbol.

For more information about the format of .init_array and .cinit, see the *TMS320C6000 Optimizing C Compiler User's Guide* and *The C6000 Embedded Application Binary Interface Application Report*.

Appendix C Helper Functions

The compiler_helper_functions_coff_to_eabi.i file translates the mangled names of COFF compiler helper functions into the EABI equivalents. The file is intended to be included in an assembly file that was originally generated by the compiler, and may have calls to compiler helper functions. It would be better if the file were recompiled from the C or C++ source for EABI, but if there have been hand tweaks to the assembly code, that may not be possible.

The compiler_helper_functions_coff_to_eabi.i file may be preincluded by using the --include_file option:

```
cl6x --include_file=compiler_helper_functions_coff_to_eabi.i
```

Example 6. COFF Helper Functions to EABI Equivalents File

```
        .asg __c6xabi_call_stub, __call_stub
        .asg __c6xabi_call_stub, __call_stub
        .asg __c6xabi_push_rts, __push_rts
        .asg __c6xabi_pop_rts, __pop_rts
        .asg __c6xabi_strasgi_64plus, __strasgi_64plus
        .asg __c6xabi_strasgi, __strasgi
        .asg __c6xabi_abort_msg, __abort_msg
        .asg __c6xabi_divi, __divi
        .asg __c6xabi_divu, __divu
        .asg __c6xabi_divul, __divul
        .asg __c6xabi_divli, __divli
        .asg __c6xabi_divull, __divull
        .asg __c6xabi_divlli, __divlli
        .asg __c6xabi_divd, __divd
        .asg __c6xabi_divf, __divf
        .asg __c6xabi_negll, __negll
        .asg __c6xabi_mpyll, __mpyll
        .asg __c6xabi_mpyiill, __mpyiill
        .asg __c6xabi_mpyuiill, __mpyuiill
        .asg __c6xabi_remi, __remi
        .asg __c6xabi_remu, __remu
        .asg __c6xabi_remul, __remul
        .asg __c6xabi_remull, __remull
        .asg __c6xabi_remli, __remli
        .asg __c6xabi_remlli, __remlli
        .asg __c6xabi_llshr, __llshr
        .asg __c6xabi_llshru, __llshru
        .asg __c6xabi_llshl, __llshl
        .asg __c6xabi_isfinite, __isfinite
        .asg __c6xabi_isfinitef, __isfinitef
        .asg __c6xabi_isinf, __isinf
        .asg __c6xabi_isinff, __isinff
        .asg __c6xabi_isnan, __isnan
        .asg __c6xabi_isnanf, __isnanf
        .asg __c6xabi_isnormal, __isnormal
        .asg __c6xabi_isnormalf, __isnormalf
        .asg __c6xabi_fpclassify, __fpclassify
        .asg __c6xabi_fpclassifyf, __fpclassifyf
        .asg __c6xabi_nround, __nround
        .asg __c6xabi_roundf, __roundf
        .asg __c6xabi_roundl, __roundl
        .asg __c6xabi_trunc, __trunc
        .asg __c6xabi_truncf, __truncf
        .asg __c6xabi_truncl, __truncl
        .asg __c6xabi_fixdi, __fixdi
        .asg __c6xabi_fixdli, __fixdli
        .asg __c6xabi_fixdlli, __fixdlli
        .asg __c6xabi_fixdlu, __fixdlu
        .asg __c6xabi_fixdu, __fixdu
        .asg __c6xabi_fixdul, __fixdul
        .asg __c6xabi_fixdull, __fixdull
        .asg __c6xabi_fixfi, __fixfi
        .asg __c6xabi_fixfli, __fixfli
        .asg __c6xabi_fixflli, __fixflli
        .asg __c6xabi_fixfu, __fixfu
```

Example 6. COFF Helper Functions to EABI Equivalents File (continued)

```
        .asg __c6xabi_fixful, __fixful
        .asg __c6xabi_fixfull, __fixfull
        .asg __c6xabi_fltid, __fltid
        .asg __c6xabi_fltif, __fltif
        .asg __c6xabi_fltlid, __fltlid
        .asg __c6xabi_fltlif, __fltlif
        .asg __c6xabi_fltllid, __fltllid
        .asg __c6xabi_fltllif, __fltllif
        .asg __c6xabi_fltud, __fltud
        .asg __c6xabi_fltuf, __fltuf
        .asg __c6xabi_fltuld, __fltuld
        .asg __c6xabi_fltulf, __fltulf
        .asg __c6xabi_fltulld, __fltulld
        .asg __c6xabi_fltullf, __fltullf
        .asg __c6xabi_cvtdf, __cvtdf
        .asg __c6xabi_cvtfd, __cvtfd
        .asg __c6xabi_cmpd, __cmpd
        .asg __c6xabi_cmpf, __cmpf
        .asg __c6xabi_eqld, __eqld
        .asg __c6xabi_eqlf, __eqlf
        .asg __c6xabi_geqd, __geqd
        .asg __c6xabi_geqf, __geqf
        .asg __c6xabi_gtrd, __gtrd
        .asg __c6xabi_gtrf, __gtrf
        .asg __c6xabi_leqd, __leqd
        .asg __c6xabi_leqf, __leqf
        .asg __c6xabi_lssd, __lssd
        .asg __c6xabi_lssf, __lssf
        .asg __c6xabi_neqd, __neqd
        .asg __c6xabi_neqf, __neqf
        .asg __c6xabi_absd, __absd
        .asg __c6xabi_absf, __absf
        .asg __c6xabi_addd, __addd
        .asg __c6xabi_addf, __addf
        .asg __c6xabi_mpyd, __mpyd
        .asg __c6xabi_mpyf, __mpyf
        .asg __c6xabi_negd, __negd
        .asg __c6xabi_negf, __negf
        .asg __c6xabi_subd, __subd
        .asg __c6xabi_subf, __subf
```

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

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