C6000 EABI Migration

This document describes the changes which may be needed to existing COFF ABI libraries and applications to be compatible with the new EABI released in the *C6000 Code Generation Tools version 7.2.0*. This is not an overview of EABI; only those details needed for migration are described here.

If you are not already familiar with the limitations of EABI support in the C6000 compiler, please see EABI Support in C6000 Compiler.

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

Contents

The C6000 EABI

C6000 code generation tools version 7.2.x introduces support for a new ELF-based ABI to support new features such as shared object files; this new ELF ABI is referred to as the EABI. 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 (http://www.ti.com/lit/sprab89)).

Documentation for features mentioned can be found in the TMS320C6000 Optimizing C Compiler User's Guide (SPRU187 (http://www.ti.com/lit/spru187), revision P or later) and the TMS320C6000 Assembly Language Tools User's Guide (SPRU186 (http://www.ti.com/lit/spru186), revision R or later).

Version Information

The first version of the C6000 compiler to support EABI is version 7.2.0. The last version to support COFF ABI is version 7.4.24. 7 years elapsed between these releases. As of this writing, there are no plans for additional releases that support COFF ABI.

Most Common User EABI Migration Issues

While this document details all the changes between COFF ABI and EABI and the changes needed to support both, most users will only need to perform a few changes to their code to move from COFF to ELF. The most common issues users are likely to encounter are:

- *40 bit type support*

  The size of long has changed from 40 bits to 32 bits, and a new native type __int40_t is now supported for 40 bit calculations.

- *No leading underscore on symbols*

  COFF adds a leading underscore to symbol names, but the EABI does not. Assembly file references to symbols will need special handling.

Migration Strategies

Before beginning work to convert a COFF project to ELF, consider whether any EABI features are desired. A working COFF program need not be converted to ELF immediately unless ELF-only features are needed. COFF will continue to be supported for some time. We encourage our customers to migrate to EABI for systems that are actively being developed.

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

Distribute Libraries in Both COFF and ELF Formats

Library vendors are strongly encouraged to distribute both COFF and ELF versions of each library. 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.

Support 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 possible.

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);
```

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.

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.

The long int type is 32 Bits

As required by the EABI standard, the long int (or just 'long') integer type is 32 bits wide in the EABI model, whereas it is 40 bits wide in the COFF ABI model. To facilitate code that is meant for both COFF and EABI applications, the new 40 bit wide native integer type __int40_t should be used. This type will provide 40 bits of precision when using either COFF or EABI.

When including stdint.h, the int40_t type can be used.

Use Cases for Declarations Involving long / int40_t

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

Typical long Use Cases

Bits Needed	Storage size a concern?	Execution speed a concern?	Native Types	Compatible Types from stdint.h
At least 32 bits	No	No	long or int	int32_t, int_least32_t, or int_fast32_t
At least 32 bits	Yes	--	int	int_least32_t
At least 32 bits	--	Yes	int	int_fast32_t
Exactly 32 bits	--	--	int	int32_t
At least 40 bits	--	No	__int40_t	int40_t, int_least40_t, or int_fast40_t
At least 40 bits	--	Yes	__int40_t	int_fast40_t
Exactly 40 bits (needs saturation and truncation)	--	--	Not recommended	**int40_t**

40-Bit Intrinsics Type Change

Because C6000 EABI does not support the 'long' native 40-bit integer type, the following native 40-bit intrinsics have new prototypes utilizing the __int40_t type, for full support under COFF and EABI:

```
__int40_t      _lsadd    (int, __int40_t)
__int40_t      _lssub    (int, __int40_t)
__int40_t      _labs     (__int40_t)
__int40_t      _dtol     (double)
__int40_t      _ldotp2   (int, int)
int            _sat      (__int40_t)
unsigned int   _lnorm    (__int40_t)
double         _ltod     (__int40_t)
```

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 to use __int40_t. Changing the prototype to use __int40_t will also work for COFF.

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.

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.

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; };
          11111111112222222233
01234567890123456789012345678901
aPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP (one 32-bit container)
```

Example 2:

```
struct S { int a:1; int b:1; };
          11111111112222222233
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).

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)
```

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.

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.

Enumerated type size

Many enumeration types have members with values that are small enough to fit into integer types smaller than int. COFF ABI and EABI will use int-sized containers to store variables of such enumeration types. 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.

In short, there is no COFF ABI to EABI migration issue regarding enumeration types when using the 7.2 compiler.

asm() Statements

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

Conflicts between variable and register names

Take, for example, the following code, where the variables A0 and A10 are named the same as the machine registers A0 and A10:

```
extern unsigned long a0;
       unsigned long a10;

void foo(void)
{
   a10 = a0;
}
```

When compiled under COFF ABI, the 'a0' and 'a10' symbols will get an underscore prefix in the assembly code that is generated by the compiler. Under the EABI model, however, the compiler does not add an underscore prefix to symbol names as explained above. The compiler will avoid conflicts between variable names in C/C++ and register names by using an escape character sequence around C/C++ variables *whose name matches a C6x register*. In the above example, the compiler will refer to the 'a0' and 'a10' variables using "||a0||" and "||a10||", respectively.

For example, compiling the above yields assembly like the following:

```
       .global foo
       .global ||a0||

       .global ||a10||
       .bss    ||a10||,4,4

       .sect   ".text"
foo:
          RETNOP  .S2     B3,4
||        LDW     .D2T2   *+DP(||a0||),B4

          STW     .D2T2   B4,*+DP(||a10||)
          ; BRANCH OCCURS {B3}
```

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.

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*.

For example, this source code:

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

Becomes in COFF ABI:

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

Note how the C/C++ symol 'x' has become '_x' in the assembly code. Assembly functions that attempt to use 'x' will use the name '_x'.

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 (see Conditional Redefinition Method or Double Label Method).

The same source code becomes in EABI:

```
   .bss x, 4, 4
func:
```

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.

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.

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.

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.

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.

This option takes effect only for hand-code assembly files and assembly files generated by the compiler from linear assembly files. This option will not affect the assembly files generated by the compiler from C and C++ source code, nor will it take effect in the linker command file.

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
```

For compiler-generated assembly code generated from linear assembly files, the assembler will recognize compiler helper function names (those prefixed with "__c6xabi_") and will not remove the COFF underscore.

Note: bug SDSCM00038757 caused the assembler to incorrectly remove the underscore from these names for C64x+ linear assembly files compiled using "-ms". As a workaround, do not use "-ms" to compile that linear assembly file.

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.

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.

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.

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

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 below).

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.

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
```

Linker Command File Changes

When porting a COFF ABI application to EABI, the most likely pace the user will need to make a change is the linker command file. The linker supports linker command file preprocessing. See The C6000 Assembly Language Tools User's Guide.

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 below for all the sections generated by the toolset.

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
```

To aid migrating COFF ABI to EABI, the v7.2 linker will automatically create the above GROUP and allocate it to the BMEM if the linker command file has the following:

```
    .bss > BMEM
```

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.

Read-Write Sections

EABI introduces the following read-write sections:

- .fardata initialized far data

The v7.2 linker will allocate the ELF section .fardata to the same memory where .far section is allocated to help the EABI migration.

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;
_symbol = 0x1234;
```

EABI:

```
mainaddr = main;
symbol = 0x1234;
```

Conditional Linking Feature

In COFF ABI mode, if an object file is explicitly included in the link step, or is pulled in from an object library to resolve a symbol definition, then, by default, the linker will include all of the sections in such an object file into the linked output file. This can be inefficient when a given object file contains many sections that are not needed in the link but are included anyway solely because one section in the file resolves a symbol.

To alleviate this inefficiency in COFF, the Code Generation Tools have for a long time provided a .clink assembler directive which allows the compiler or a user to indicate that a section is eligible for removal by the linker if it is not referenced. This linker process is referred to as conditional linking. In COFF ABI the compiler generates .clink directive for code sections automatically. That is, for COFF, compiler generated code sections are eligible for removal via conditional linking. (The compiler generated data sections are always linked by the linker.)

In EABI mode, all sections are eligible for removal via conditional linking *by default*. This means when migrating COFF ABI application to EABI, one must make sure that needed but unreferenced sections (such as overlays or debug functionality) are explicitly retained.

To help developers with the transition of existing COFF applications to the EABI model and the creation of new applications, the CGT 7.2 release supports the following:

1. The following pragma can be used in C/C++ source files

`#pragma RETAIN(<symbol>)`
When this pragma is applied to a function or data object symbol, it instructs the compiler to generate a .retain assembler directive into the section that contains the definition of the symbol. This provides a mechanism for the developer to indicate in their C/C++ source that a section containing the specified symbol is to be considered ineligible for removal during conditional linking.

> NOTE: When compiling code for an interrupt function that is written in C/C++, the compiler will generate a .retain directive into the section that contains the definition of the interrupt function. This can be overridden by applying a #pragma CLINK() to the interrupt function symbol.

2. The following directive can be used in the assembly source files:

`.retain ["<section name>"]`
If a "section name" argument is provided to the .retain directive, the assembler will mark the specified section as ineligible for removal by conditional linking. If no "section name" argument is specified, then the currently active initialized section is marked as ineligible for removal via conditional linking.

3. The linker option `--retain` can be used to specify a symbol or section to be retained by the linker.

Note that the compiler automatically detects interrupt vectors in version 7.2, and marks them ineligible for conditional linking to help users easily migrate from COFF ABI to EABI.

Miscellaneous

Relocation Expressions Are Not Supported

Assembler expressions 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)
```

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.

--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 as required by the ELF specification. 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.

Special Symbols

COFF ABI and EABI define special symbols for management of ABI functionality and memory usage. Some variables were renamed for EABI to bring them in line with the EABI standard. Note that a __TI prefix is now a standard part of many symbol names.

Symbol Name Changes

The COFF ABI name is the name as it would appear in assembly code or the linker command file. The EABI name is the new name for the symbol in EABI; uses of the old name in linker command files or assembler files should be updated or conditionalized to refer to the new name, as appropriate.

Special Symbols

COFF ABI Name	EABI Name	Purpose	COFF names in EABI?	Notes
___binit__ or binit	__binit__ or binit	Boot-time initialization		Note the change from 3 leading underscores to 2
___c_args__	__c_args__	Command-line arguments	Yes	Note the change from 3 leading underscores to 2
___cinit__ or cinit	__TI_CINIT_Base	Start of C global variable initializers table	No	New compressed table format in EABI. Not NULL terminated in EABI; see __TI_CINIT_Limit.
N/A	__TI_CINIT_Limit	End of C global variable initializer data		
___data__	*N/A*	Beginning of the .data section		
___edata__	*N/A*	End of the .data section		

___end__	*N/A*	End of the .bss section		
___etext__	*N/A*	End of the .text section		
___pinit__ or pinit	__TI_INITARRAY_BASE	Start of C++ global object initializers table	Yes	Not NULL terminated in EABI; see __TI_INITARRAY_Limit
N/A	__TI_INITARRAY_Limit	End of C++ global object initializer data		
___text__ or .text	*N/A*	Beginning of the .text section		
__bss__ .bss or $bss	__TI_STATIC_BASE	Start of DP-relative data	Yes	
__STACK_SIZE	__TI_STACK_SIZE	Size available for function frame stack	Yes	
__SYSMEM_SIZE	__TI_SYSMEM_SIZE	Size available for heap allocation	Yes	
__STACK_END	__TI_STACK_END	End of the .stack section	Yes	
C$$EXIT	C$$EXIT	Special host I/O trap		
C$$IO$$	C$$IO$$	Special host I/O trap		

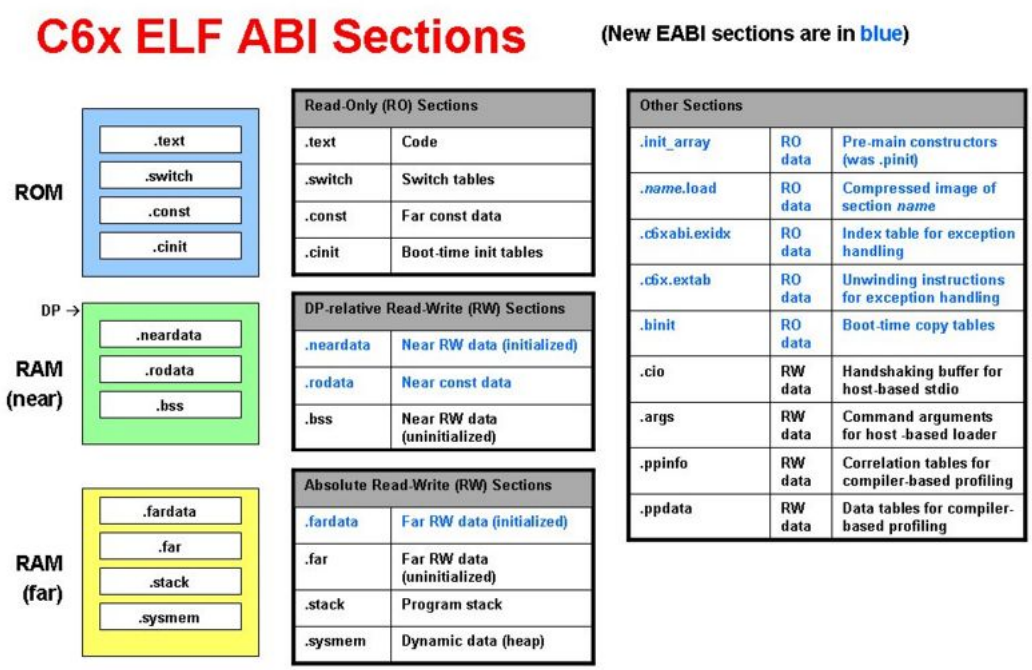
While in COFF ABI .pinit and .cinit are terminated with NULL records, in EABI the ending addresses of .init_array and .cinit are indicated by the corresponding LIMIT symbol. For more information about the format of .init_array and .cinit, see the TMS320C6000 Optimizing C Compiler User's Guide and the C6000 EABI Application Note.

Also note that in EABI the CINIT table has a new compressed format. For details, see the TMS320C6000 Optimizing C Compiler User's Guide, section 7.8.4.

Backward Compatibility

In an attempt to be backward compatible under EABI while users are migrating their code and process, the linker will still recognize several of the old symbols names and will define them at link time *if* they are used, while also generating a warning referring the user to their new name. Refer to the 'COFF names in EABI' column to identify which symbols are supported in this manner.

C6x EABI Sections



The linker tries to allocate most of the new EABI sections when the user doesn't allocate these sections explicitly.

.neardata and .rodata sections

When .bss is allocated in the linker command file and when .neardata and .rodata are not allocated, then the linker groups the .bss, .neardata and .rodata sections together and allocates them where .bss is allocated. For example, if the linker command file had the following and .neardata/.rodata are not specified:

```
.bss > BMEM
```

then the linker automatically allocates as if the following is specified:

```
GROUP (NEARDP_DATA)
{
     .neardata
     .rodata
```

```
      .bss
 }  > BMEM
```

When this is done, under -w option, the linker generates the following warnings:

```
warning: creating output section ".neardata" without a SECTIONS specification.
   For additional information on this section, please see the 'C6000 EABI
   Migration' guide at
   http://processors.wiki.ti.com/index.php/C6000_EABI:C6000_EABI_Migration#C6x_EABI_Sections
warning: creating output section ".rodata" without a SECTIONS specification.
   For additional information on this section, please see the 'C6000 EABI
   Migration' guide at
   http://processors.wiki.ti.com/index.php/C6000_EABI:C6000_EABI_Migration#C6x_EABI_Sections
warning: creating group "NEARDP_DATA" to co-locate output sections .neardata
   and .rodata with .bss
```

If you want to avoid these warning please update the linker command file to specify the GROUP for the near DP section allocation.

.fardata section

If .fardata is not allocated in the linker command file and .far section is allocated then the linker automatically allocates the .fardata section to the same memory area as .far section allocation. If -w option is used then the linker also generates the following warning:

```
warning: creating output section ".fardata" without a SECTIONS specification.
   For additional information on this section, please see the 'C6000 EABI
   Migration' guide at
   http://processors.wiki.ti.com/index.php/C6000_EABI:C6000_EABI_Migration#C6x_EABI_Sections
```

Explicitly allocate the .fardata section to avoid this warning.

.init_array

If .init_array is not allocated in the linker command file and .pinit section is allocated then the linker automatically allocates the .init_array section to the same memory area as .pinit section allocation. If -w option is used then the linker also generates the following warning:

```
warning: creating output section ".init_array" without a SECTIONS specification.
   For additional information on this section, please see the 'C6000 EABI
   Migration' guide at
   http://processors.wiki.ti.com/index.php/C6000_EABI:C6000_EABI_Migration#C6x_EABI_Sections
```

Either explicitly allocate the .init_array section in the linker command file or if there are no .pinit sections remove the .pinit allocation from the linker command file.