# TMS470 ARM ABI Migration

## Revision History

| Version | Primary Author(s) | Description of Version | Date Completed |
|---|---|---|---|
| V0.1 | Anbu Gopalrajan | Initial Draft | 10/29/2006 |
| V0.2 | Anbu Gopalrajan | Added C auto initialization | 04/04/2007 |
| | | | |
| | | | |

# 1. Introduction

## *1.1 ABI*

ABI stands for Application Binary Interface and it defines the low level interface between component parts of an application. ABI allows compiled object code to function without changes on any system using a compatible ABI. An ABI for a bare-metal system primarily defines function calling convention, object format and debug format.

TMS470 compiler toolset version 4.2 supported two ABIs, namely TI ABI and TI ARM9 ABI. Please refer to the TMS470 C/C++ Compiler Getting Started Guide, included with the compiler release, for more information on these ABIs and how to migrate from TI ABI to TI ARM9 ABI.

## *1.2 ARM ABIv2 or EABI*

An industry consortium founded by ARM Ltd defined a standard ABI for binary code intended for ARM architecture. This ABI is called the Application Binary Interface (ABI) for the ARM Architecture Version 2 (ARM ABIv2). This ABI is also referred to as Embedded Application Binary Interface (EABI). Both ABIv2 and EABI are used in this document interchangeably.

ARM ABIv2 has become an industry standard for ARM architecture and has the following advantages:

- It enables interlinking of objects built with different tool chains. For example, this enables a library built with RVCT to be linked in with an application built with TMS470 4.4 toolset.
- It is well documented and the documentation is a reference point for users writing utilities that requires an understanding of the ABI. The complete ARM ABI specifications can be found at http://www.arm.com/products/DevTools/ABI.html.
- It is modern. EABI requires ELF object file format which enables supporting modern language features like early template instantiation and export inline functions support.

TMS470 compiler toolset version 4.4 supports ARM ABIv2 (or EABI) along with TI ABI and TI ARM9 ABI. To leverage the advantages listed above the user needs to migrate to EABI. This document is intended for users migrating from TI ARM9 ABI to EABI.

Please note that EABI is incompatible with TIARM9ABI. This means, if you have code built with TI ARM9 ABI, you need to recompile your entire application including your libraries to migrate to EABI.

A companion document named ‘ARM ABI Interlinking Using TI Tools’ captures issues related to interlinking object from other compiler vendors with TI TMS470 objects.

# 2. Migration steps

## *2.1 Shell options to use*

The shell cl470 supports --abi option that can take arguments “tiabi”, "ti_arm9_abi" or "eabi". Use --abi=eabi in the shell before -z option to build EABI objects and executables. When using --abi=eabi option, Runtime Support (RTS) libraries built with --abi=eabi should be used. Please look in the release notes to specify the right library.

```
> cl470 --abi=eabi hello.c -z -llnk.cmd -lrts32eabi.lib
```

The 4.4 compiler has a new auto library selection support. If no runtime support library is specified in the command line and building a C/C++ executable, the linker will automatically choose the best match from the list of RTS libraries delivered with the 4.4 compiler.

## *2.2 Changes to Linker command file*

In TI ARM9 ABI mode C++ global constructor vector is in the section .pinit. Under EABI, this vector is in the section .init_array. If you have sections specification in the linker command file for .pinit section, change to .init_array.

## *2.3 Changes to assembly source*

EABI is a set of standards and one of the standards is AAPCS (Procedure Call Standard for the ARM Architecture) which documents the procedure calling convention and data layout. TMS470 compiler under EABI generates code that conforms to this procedure calling conventions. If you have mixed assembly and C code and asm functions are called from C or vice versa, the assembly function should conform to AAPCS.

### 2.3.1 Stack Alignment

EABI requires that 64-bit data (type long long and long double) be aligned at 64-bits. This requires that the stack be aligned at 64-bit boundary at function entry so that local 64-bit variables are allocated in stack with correct alignment. Assembly functions calling a C function should make sure the stack is aligned at 64-bits at the call site. The assembly function is guaranteed to have stack aligned at 64-bit at entry if it is called from a C function. An assembly function allocating 64-bit data should align it properly if its address is passed over call boundary.

### 2.3.2 Function Argument Passing and Return

- In TI ARM9 ABI, 64-bit data is 4-byte aligned when passed in stack and is passed using any of the argument registers r0-r3. In EABI, 64-bit data passed in stack should be aligned at 64-bit boundary. When such data is passed as argument in ARM core registers, it can only be passed in r0:r1 or r2:r3. It cannot be passed in r1:r2. If int and long long are passed as first and second arguments, this means that r1 is not used.
- In TI ARM9 ABI, when struct or union is passed by value in a C code, the compiler internally passes them by reference. In EABI, such structs and unions are actually passed by value by the compiler. Assembly functions passing or accepting such arguments to/from C function should be updated. For more details please refer to Appendix A.
- In TI ARM9 ABI, chars and shorts are passed as is and they take 1-byte and 2-byte respectively when passed in stack. In EABI, shorts and chars are widened to int before passing as argument. This means shorts and chars will take 4-bytes when passed in stack.
- In TI ARM9 ABI char and shorts are returned as is in r0. In EABI, chars and shorts are zero or sign extended to a 4-byte word and returned in r0.
- In TI ARM9 ABI, all structures are returned by reference. In EABI, small structs of size 4 bytes are returned in register r0. Larger structs are returned by reference as in TI ARM9 ABI.
- In TI ARM9 ABI, variable arguments are always passed in stack even if the argument registers (r0-r3) are available. In EABI, variable arguments are passed in argument registers if available similar to non-variable arguments.

### 2.3.3 Data Layout

- In TI ARM9 ABI, 64-bit data has 4-byte alignment. In EABI, 64-bit data has 8-byte alignment.
- In TI ARM9 EABI enumerator (enum) uses int or higher type as the underlying type. In EABI, enumerator uses the smallest underlying type that can represent the enumeration range. The compiler can generate int sized enums when --enum_type=int command line option is used.
- Bit fields are laid out differently under EABI. Please refer to Appendix B for more details.

- TI ABI uses 4-byte alignment for arrays of type char and short to optimize the access of members of these arrays. EABI requires that the arrays have the alignment of the base type. This may not have any ramifications if the user uses only TI tools.

### 2.3.4 Name mangling

EABI uses ELF object format which requires that the link time symbol that names a C or assembly language entity should have the name of that entity. For example, a C function foo() generates a symbol called foo (not _foo or $foo). If your assembly code is relying on the leading underscore or $ to access C symbols, they need to be changed.

If the C variable name clashes with assembler directives or assembler internal symbols, surround the symbol name with '||' in the assembly source. For example, C variable named 'r0' will conflict with assembler register r0 without the leading underscore or '$'. If you need to access such C variables, refer to them as ||r0||. For ex,

```
.global ||r0||
```

EABI uses C++ name mangling as documented in Itanium C++ ABI (IA64) which is different from the C++ name mangling used by TI ABI. If asm functions access C++ symbols the mangling name should be updated as required by Itanium C++ ABI.

# 3. Migration Issues

## *3.1 Unused Section Elimination*

The TMS470 compiler toolset in COFF mode performs unused section elimination by marking .text sections as conditional link sections. The linker links in conditional link sections only when they are needed to resolve any references. A conditional link section has the COFF section flag STYP_CLINK.

In EABI mode, the ELF linker by default links in sections only when they are required for resolving any references in the executable. The compiler does not mark any section as conditional link sections but the linker treats all input sections as conditional link sections. This means the user could see the following behaviors:

1. If user does not specify an entry point usually no sections are linked in. In that case the linker generates the following error message:
   > error: no input sections are linked in
   User can use -c or -cr to let the linker define the standard entry point (_c_int00) or -e to specify an entry point. If compile and link are performed in a single step and if C files are included in the compile the shell will automatically include -c option for the link.
2. Interrupt vectors usually don't have any references from source code. These are not linked in unless the user tells the linker to retain the interrupt vector sections.
3. In some applications, user data is only referenced by using a link time symbol as below:

   ```
   data_table  { *(data_table)  } START(dstart), END(dend) > FLASH
   ```

   The linker defines dstart and dend symbols which a program can use to parse the table. In this case there is no direct reference to the input sections data_table and hence they are not linked in.

The ELF linker supports two options to control the unused section elimination.

--unused_section_elimination[=off,on]
In order to minimize the foot print, the ELF linker does not include a section that is not needed to resolve any references in the final executable.
--unused_section_elimination=off can be used to disable this optimization. (Default:on)

*--retain=sym_or_scn_spec*
When --unused_section_elimination is on, the ELF linker does not include a section in the final link if it is not needed in the executable to resolve references. This option can be used to tell the linker to retain a list of sections that would other-wise be not retained. This option accepts the wild cards '*' and '?' and when wild cards are used, the argument should be in quotes. The following option parameters are accepted:

*--retain=symbol_spec* retains sections that define symbol matching the argument. For ex, `--retain='init*'` retains sections that define symbols that start with 'init'. User may not specify `--retain='*'`.

*--retain=file_spec(scn_spec [,scn_spec ...])* retains sections matching scn_spec(s) from files matching file_spec. For ex, `--retain='*(.initvec)'` causes the linker to retain .initvec sections from all input files.
NOTE: User can specify `--retain='*(*)'` to retain all the sections from all the input object files. Note that this does not prevent sections from library members from being optimized out. If you want to totally disable unused section elimination, please use --unused_section_elimination=off.

*--retain=ar_spec<mem_spec, [mem_spec ...]>(scn_spec, [scn_spec ...])* retains sections matching scn_spec(s) from members matching mem_spec(s) from archive files matching ar_spec. For ex, `--retain='rts32eabi.lib<printf.obj>(.text)'` causes the linker to retain .text section from printf.obj in rts32eabi.lib. If the library is specified with -l option (-lrts32eabi.lib) the library search path is used to search the library. User may not specify `'*<*>(*)'`.

# 4. Automatic Initialization of Variables

Any global variables declared as preinitialized must have initial values assigned to them before a C/C++ program starts running. The process of retrieving these variables' data and initializing the variables with the data is called autoinitialization. Please refer to the TMS470 Compiler user guide for details on TI ARM9 ABI automatic initialization of variables.

The following apply to the automatic initialization of variables in EABI mode.

## *4.2 Zero Initialization*

In ANSI C, global and static variables that are not explicitly initialized must be set to 0 before program execution. The C/C++ EABI compiler supports preinitialization of uninitialized variables by default. This can be turned off by specifying the linker option --zero_init=off. TI ARM9 ABI does not support zero initialization.

## 4.3 EABI Direct Initialization

The EABI compiler uses direct initialization to initialize global variables. For example, consider the following C code:

```
int i      = 23;
int a[5] = { 1, 2, 3, 4, 5 };
```

The compiler allocates the variables ‘i’ and ‘a[] to .data section and the initial values are placed directly.

```
        .global i
        .data
        .align  4
i:
        .field          23,32                   ; i @ 0

        .global a
        .data
        .align  4
a:
        .field          1,32                    ; a[0] @ 0
        .field          2,32                    ; a[1] @ 32
        .field          3,32                    ; a[2] @ 64
        .field          4,32                    ; a[3] @ 96
        .field          5,32                    ; a[4] @ 128
```

Each compiled module that defines static or global variables contains these .data sections.

## 4.4 Initialization of variables at run time in EABI mode

Auto initializing variables at run time is the default method of autoinitialization. To use this method invoke the link step with the --rom_model (-c) option. Using this method, the linker creates initialization table and initialization data from the direct initialized sections in the compile module. This table and data are used by the C/C++ boot routine to initialize variables in RAM using the table and data in ROM.

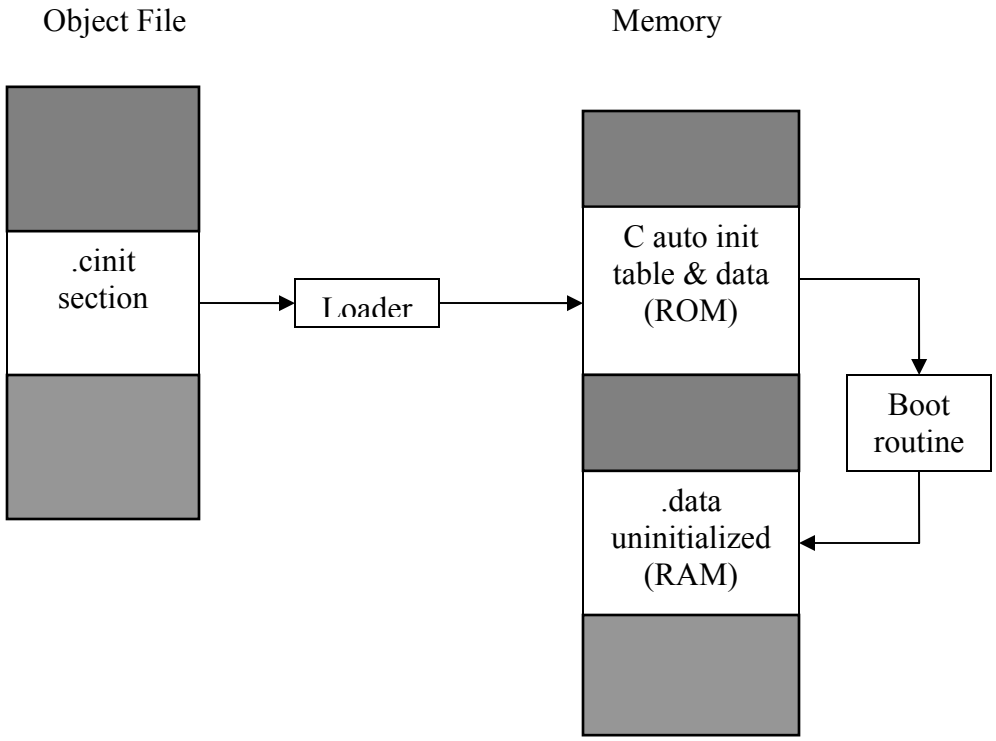


**Fig 1. Auto Initialization at Run Time**

### 4.4.1 EABI Auto Initialization Table

In EABI mode, the compiled modules (obj files) don't have initialization table. The variables are direct initialized. The linker, when -c option is specified, creates C auto initialization table and the initialization data (load data). The linker creates both the table and the initialization data in an output section named .cinit.

> **NOTE:**
> The name .cinit is used primarily to simplify migration from COFF to ELF format and the .cinit section created by the linker has nothing in common (except the name) with the COFF cinit records.

The C auto initialization table has the following format:

__TI_CINIT_Base:

| 32-bit load address | 32-bit run address |
|---|---|
| • | • |
| • | • |
| • | • |
| 32-bit load address | 32-bit run address |

__TI_CINIT_Limit:

The linker defined symbols __TI_CINIT_Base and __TI_CINIT_Limit point to the start and end of the table, respectively. Each entry in this table corresponds to one output section that needs to be

initialized. The initialization data for each output section could be encoded using different encoding.

The load address in the C auto initialization record points to initialization data with the following format:

| 8-bit Index | Encoded Data |
|---|---|

The first 8-bits of the initialization data is the hander index. It indexes into a handler table to get the address of a handler function that knows how to decode the following data.

The handler table is a list of 32-bit function pointers.

__TI_Handler_Table_Base:

| 32-bit Handler Address 1 |
|---|
| • |
| • |
| • |
| 32-bit Handler Address N |

__TI_Handler_Table_Limit:

The 'Encoded Data' that follows the 8-bit index can be one of the following. For clarity the 8-bit index is also depicted in the format below.

**Length followed by data:**

| 8-bit Index | 24-bit padding | 32-bit length (N) | N byte initialization data (not compressed) |
|---|---|---|---|

24-bit padding is used to align the length field to a 32-bit boundary. The 32-bit length field encodes the length of the initialization data in bytes (N). N byte initialization data is not compressed and copied to the run address as is.

The TMS470 Run Time Support (RTS) library has the function __TI_decompress_none() process this type of initialization data. The first argument to this function is the address pointing to the byte after the 8-bit Index and the second argument is the run address from the C auto initialization record.

**Zero Initialization:**

| 8-bit Index | 24-bit padding | 32-bit length (N) |
|---|---|---|

24-bit padding is used to align the length field to a 32-bit boundary. The 32-bit length field encodes the number of bytes to be zero initialized.

The TMS470 RTS library has a function __TI_zero_init() to process the zero initialization. The first argument to this function is the address pointing to the byte after the 8-bit Index and the second argument is the run address from the C auto initialization record.

**Run Length Encoding (RLE):**

| 8-bit Index | Initialization data compressed using Run Length Encoding |
|---|---|

The data following the 8-bit index is compressed using Run Length Encoded (RLE). TMS470 uses a simple Run Length Encoding that can be decompressed using the following algorithm:

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

The TMS470 RTS library has the routine __TI_decompress_rle() to decompress data compressed using RLE. The first argument to this function is the address pointing to the byte after the 8-bit Index and the second argument is the run address from the C auto initialization record.

**Lempel-Ziv Storer and Szymanski compression (LZSS):**

| 8-bit Index | Initialization data compressed using LZSS |
|---|---|

The data following the 8-bit index is compressed using LZSS compression. The TMS470 RTS library has the routine __TI_decompress_lzss() to decompress the data compressed using LZSS. The first argument to this function is the address pointing to the byte after the 8-bit Index and the second argument is the run address from the C auto initialization record.

## 4.4.2 Sample C Code to process the C Auto Initialization Table:

The TMS470 RTS boot routine has code to process the C Auto Initialization Table. The following C code illustrates how the Auto Initialization Table can be processed on the target.

```
typedef void (*handler_fptr)(const unsigned char *in,
                             unsigned char *out);

#define HANDLER_TABLE   __TI_Handler_Table_Base
#pragma WEAK(HANDLER_TABLE)
extern  unsigned int   HANDLER_TABLE;
extern  unsigned char *__TI_CINIT_Base;
extern  unsigned char *__TI_CINIT_Limit;

void auto_initialize()
{
   unsigned char **table_ptr;
   unsigned char **table_limit;

   /*-------------------------------------------------------------*/
   /* Check if Handler table has entries.                         */
   /*-------------------------------------------------------------*/
   if (&__TI_Handler_Table_Base >= &__TI_Handler_Table_Limit)
```

```
      return;

   /*-------------------------------------------------------------*/
   /* Get the Start and End of the CINIT Table.                   */
   /*-------------------------------------------------------------*/
   table_ptr   = (unsigned char **)&__TI_CINIT_Base;
   table_limit = (unsigned char **)&__TI_CINIT_Limit;
   while (table_ptr < table_limit)
   {
      /*----------------------------------------------------------*/
      /* 1. Get the Load and Run address.                         */
      /* 2. Read the 8-bit index from the load address.           */
      /* 3. Get the hander function pointer using the index from  */
      /*    handler table.                                        */
      /*----------------------------------------------------------*/
      unsigned char *load_addr   = *table_ptr++;
      unsigned char *run_addr    = *table_ptr++;
      unsigned char  handler_idx = *load_addr++;
      handler_fptr   handler     =
                  (handler_fptr)(&HANDLER_TABLE)[handler_idx];

      /*----------------------------------------------------------*/
      /* 4. Call the handler and pass the pointer to the load data*/
      /*    after index and the run address.                      */
      /*----------------------------------------------------------*/
      (*handler)((const unsigned char *)load_addr, run_addr);
   }
}
```

## *4.5 Initialization of variables at load time:*

Initialization of variables at load time enhances performance by reducing boot time and by saving the memory used by the initialization tables. To use this method, invoke the link step with the --ram_model (-cr) option.

When you use the --ram_model link option, the linker does not generate C auto initialization tables and data. The direct initialized sections in the compile modules are combined as per the linker command file to generate initialized output sections. The loader loads the initialized output sections into memory. After the load, the variables have their initial values.

Since the linker does not generate the C auto initialization tables, no boot time initialization is performed.

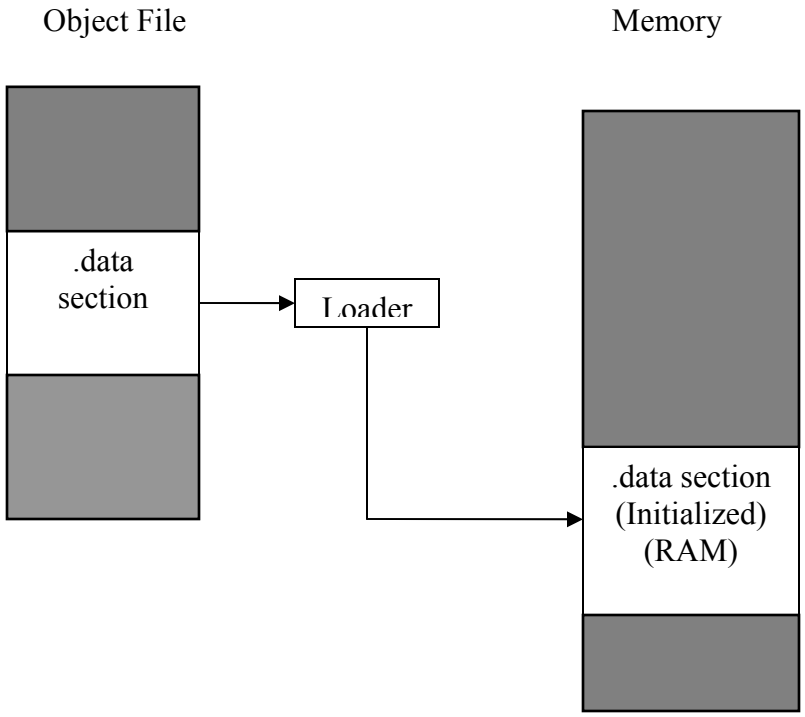


**Fig 2. Auto Initialization at Load Time**

## 5. Global Constructors

All global C++ variables that have constructors must have their constructor called before main(). The compiler builds a table of global constructor addresses that must be called, in order, before main() in a section called .init_array. The link step combines the .init_array section form each input file to form a single table in the .init_array section. The boot routine uses this table to execute the constructors. The linker defines two symbols to identify the combined .init_array table as shown below. This table is not null terminated by the linker.

SHT$$INIT_ARRAY$$Base:

| Address of constructor 1 |
|---|
| Address of constructor 2 |
| • • • |
| Address of constructor n |

SHT$$INIT_ARRAY$$Limit:

# APPENDIX A. Struct/Union Passing Rules under EABI

In TI ARM9 ABI, a struct or union passed by value in C code is internally passed by reference by the compiler. For example, consider the following C code:

```
struct st {int a, b; } st_var;
asm_func(st);
```

In TI ARM9 ABI, asm_func is passed the address of st_var in r0.

In EABI, struct passed by value is actually passed by value by the compiler. The above mentioned asm_func is passed the struct st_var in registers r0 and r1 in EABI mode. The following rules apply while passing structs by value under EABI:

- If no argument registers are available the struct is passed in stack. The structure is aligned at 4-byte boundary if its alignment is less than 4-bytes.
- If there are argument registers (r0-r3) available and the entire struct fits in the available registers, the struct is passed entirely in registers.
- If argument registers are available to pass part of the struct, then the initial part of the struct is passed in registers and the remaining struct is passed in stack. When this happens, the called function should push the registers that contain the struct to stack and access the entire struct from stack.
- If the struct alignment is 8-bytes and is passed in registers, it can start only in r0 or r2 registers.

# APPENDIX B. Bit Fields

1. In TI ARM9 ABI, plain int bit fields are signed. In EABI they are unsigned. Consider the following C code,

```
struct st
{
   int a:5;
} S;

foo()
{
   if (S.a < 0)
      bar();
}
```

In EABI, bar() is never called as bit field 'a' is unsigned. Use signed int if you need signed bit field in EABI mode.

2. In TI ARM9 ABI, bit fields of type long long is not allowed. In EABI, long long bit fields are supported.

3. In TI ARM9 ABI, all bit fields are treated as signed or unsigned int type. In EABI, bit fields are treated as the declared type.

4. In TI ARM9 ABI, the size and alignment a bit field contributes to the struct containing it depends on the number of bits in the bit field. In EABI, the size and alignment of the struct containing the bit field depends on the declared type of the bit field. For example,

struct st { int a:4; } takes up 1-byte and is aligned at 1-byte in TI ARM9 ABI. The same struct uses up 4-bytes and is aligned at 4-bytes in EABI.

5. In TI ARM9 ABI, unnamed bit fields and zero sized bit fields don't affect the struct or union alignment. In EABI, such fields affect the alignment of the struct or union. Consider the struct

struct st { char a:4; int :22; };

In TI ARM9 ABI, this struct uses 4-bytes and aligned at 1-byte boundary. In EABI, this struct uses 4-bytes and aligned at 4-bytes.