Application Report
SPRAAB5–August 2005

# The Impact of DWARF on TI Object Files

Don Darling
Software Development Systems

**ABSTRACT**

This document identifies the differences in object file content when DWARF replaces COFF as the primary debug information format. Please refer to *DWARF Debugging Information Format Specification Version 2.0* for a description of this format.

**Contents**



## Trademarks

C54x, C62x, C28x are trademarks of Texas Instruments.

## 1 Introduction

The DWARF debug information format provides a much more expressive representation of symbolic debugging information than COFF debug, and overcomes many of COFF debug's limitations. The following benefits are obtained by making DWARF the primary debug format used by the TI compiler:

- Support for C++
- Support for stepping through inline functions
- Support for stepping through #include files
- Support for source files with more than 65535 lines
- Support for arrays with more than four dimensions
- Support for source column information

This document identifies the differences in object file content when DWARF replaces COFF debug. It is also intended to supplement the *DWARF Debugging Information Format Specification Version 2.0* (hereinafter referred to as the DWARF2 specification) listed in Section 5, giving specific details about the debug information generated by the TI compiler.

# 2 Changes to the COFF Object File

## 2.1 A Single-Purpose Symbol Table

In the past, the symbol table has served two purposes:

1. Provide name and address pairs to the linker, so it can resolve external references and perform symbol-relative relocation.
2. Provide additional debug information about a symbol (i.e., function parameters and local variables, variable type information, etc.)

In object files compiled with DWARF, the symbol table exists only for the first purpose listed. Any and all debug information about a symbol must now be retrieved from the DWARF sections. The compiler will not generate redundant debug information in the symbol table when it is generating DWARF. This would lead to confusion in object file consumers, and may result in conflicting information since DWARF is a more expressive format.

If an object file compiled with DWARF debug is linked with an object file compiled with COFF debug, the resulting object file will have a mix of COFF and DWARF debug in it. In such cases, DWARF information should be checked first when looking for information about a symbol, as its entry in the symbol table will not contain debug information.

## 2.2 Line Number Entries

DWARF object files will not have any COFF line number entries. All source correspondence information must be retrieved from the DWARF `.debug_line` section.

# 3 TI-Specific Features of the DWARF2 Format

## 3.1 Scope of DWARF Support

The TI compiler currently generates these DWARF sections:

```
.debug_abbrev
.debug_info
.debug_line
.debug_frame
```

## 3.2 Endianness of DWARF Sections

The endianness of the DWARF information always reflects the endianness of the target data in the object file. To the linker, DWARF sections appear as copy sections, and the endianness must be consistent with the target for relocation to be performed correctly.

## 3.3 Void Pointers

No base type is given in the DWARF2 specification documentation that can accurately represent the C and C++ *void* type. To represent void pointers, a `DW_TAG pointer_type` type modifier is used without a `DW_AT_type` attribute to reference any base type. For qualified void pointers, the `DW_TAG_pointer_type` may point to a qualifier type that does not have a `DW_AT_type` attribute. This follows the convention described in section 3.3.2, Subroutine and Entry Point Return Types of the DWARF2 specification for specifying the return type of void functions.

## 3.4 Call Frame Conventions

To standardize the generation of call frame information among the different architectures, some conventions are established that may not seem natural to a given architecture, but do not violate the DWARF2 specification.

### 3.4.1 Frame Boundaries

A function's call frame as defined in the `.debug_frame` section is not necessarily the same as the one defined by the compiler's calling convention. For example, a program running on a C54x™ pushes a return value onto the stack during a function call, but the compiler does not consider that return value to be part of any function's frame.

The `.debug_frame` section calculates the canonical frame address (CFA) for the current frame to be value of the stack pointer just before the current function was called. By this definition, a function's frame would then include the return address on a C54x. Therefore, it should not be assumed that the CFA value is the same as the frame pointer (if a frame pointer exists).

### 3.4.2 Return Address Location

Since some architectures push return addresses on the stack and others pass them in registers, the following abstract method for specifying the location of the return address is used.

The header of a function will always show that the return address is being passed in a register named `"CIE_RETA"`. The CIE or FDE entry will then show that `CIE_RETA` is saved to the correct location prior to reaching the first instruction of the function.

Here are two examples for a “Hello World!” program, one for C62x™ DSP, and one for C54x™ DSP:

C62x:

```
FDE ENTRY FOR FUNCTION main (0x00000000)
Return Reg:      CIE_RETA
0x00000000   CIE_RETA -> B3
0x00000008   CIE_RETA -> B3, B3 -> [SP]+16
0x0000002c   CIE_RETA -> B3
```

C54x:

```
FDE ENTRY FOR FUNCTION main (0x00000000)
Return Reg:      CIE_RETA
0x00000000   CIE_RETA -> [SP]+0
0x00000002   CIE_RETA -> [SP]+3
0x00000008   CIE_RETA -> [SP]+0
```

As shown above, `CIE_RETA` is saved to register B3 on C62x, and to SP+0 on C54x before reaching the first instruction of `main`.

### 3.4.3 The `DW_CFA_def_cfa` and `DW_CFA_def_cfa_offset` Instructions

Section 6.4.1, Structure of Call Frame Information of the DWARF2 specification states that the `data_alignment_factor` is to be factored out of all offset instructions. The descriptions of `DW_CFA_offset` and `DW_CFA_offset_extended` explicitly show the offset being multiplied by `data_alignment_factor`, where the definitions of `DW_CFA_def_cfa` and `DW_CFA_def_cfa_offset` do not. The TI compiler does **not** divide the offsets for `DW_CFA_def_cfa` and `DW_CFA_def_cfa_offset` by `data_alignment_factor`.

## *3.5 Locations of Function Parameters*

Without support for the `.debug_loc` section, there is no way to track the location of local variable values if they change during the execution of a function. However, a method is provided to find the locations of function parameters that are copied to the stack during the execution of the function prologue. A function contains both a `DW_TAG_formal_parameter` as well as a `DW_TAG_variable` entry for parameters of this kind.

Example Function:

```
int func(register int a, int b)
{
   int c;
   /* function body */
   return c;
}
```

For the above function, the DWARF information would look something like the following textual representation, with indentation indicating parent/child relationships:

```
DW_TAG_subprogram (0x19 bytes)
  DW_AT_name func
  DW_AT_low_pc 0x0000
  DW_AT_high_pc 0x0020
  DW_AT_external true
  DW_AT_type 0x306 (int)
  DW_AT_TI_symbol_name _func
    DW_TAG_formal_parameter (0xc bytes)
     DW_AT_location { DW_OP_reg4 }
     DW_AT_name a
     DW_AT_type 0x306 (int)
     DW_AT_TI_symbol_name _a
    DW_TAG_formal_parameter (0xc bytes)
     DW_AT_location { DW_OP_reg20 }
     DW_AT_name b
     DW_AT_type 0x306 (int)
     DW_AT_TI_symbol_name _b
    DW_TAG_variable (0xd bytes)
     DW_AT_location { DW_OP_breg31 0x00000004 }
     DW_AT_name b
     DW_AT_type 0x306 (int)
     DW_AT_TI_symbol_name _b
    DW_TAG_variable (0xd bytes)
     DW_AT_location { DW_OP_breg31 0x00000008 }
     DW_AT_name c
     DW_AT_type 0x306 (int)
     DW_AT_TI_symbol_name _c
```

The above variable *b* has two entries:

- one for its location when it was passed in as a parameter
- one for the location it is stored on the stack during the execution of the function

**Note:**

The variable *a* only has one entry. In this case, the compiler decided not to copy *a* to the stack, and it will live in `DW_OP_reg4` for the duration of the function.

There is not enough information to determine when the location of a parameter changes, so that the debugger knows when it should switch from using the location specified by the `DW_TAG_formal_parameter` to that of the `DW_TAG_variable`. TI's recommendation is to use the former only on the first instruction of the function, or if possible, detect the copy at runtime.

## 4 TI Extensions to DWARF

To comply with the vendor extensibility requirements of the DWARF2 specification (section 7.1), some extensions to DWARF have been added to satisfy the needs of TI DSP architectures that were not accounted for by the TIS committee. A document describing the complete list of extensions is not available at the time of this writing, but most can be safely ignored.

This section briefly documents the extensions that are the most useful to TI object file consumers.

## 4.1 Tag Definitions

*Type Modifiers*

These additional type modifiers are used in the same manner as those described in section 5.2, Type Modifier Entries of DWARF2 specification.

| Tag Name | Value |
|---|---|
| DW_TAG_TI_far_type | 0x4080 |
| DW_TAG_TI_near_type | 0x4081 |
| DW_TAG_TI_ioport_type | 0x4083 |
| DW_TAG_TI_restrict_type | 0x4084 |
| DW_TAG_TI_onchip_type | 0x4085 |

Pointer sizes should not be inferred from the existence of far or near qualifiers. They will only exist in the DWARF information if they exist in the source code. For example, when compiling code for C28x™ using the large memory model, all pointer values are 32-bits regardless of whether the far qualifier was specified. Pointer sizes are determined using the address class of a pointer as described in Section 4.3.

*Register Mapping*

The DWARF specification refers to registers using register name operators such as `DW_OP_reg0`, `DW_OP reg1`, etc., and assumes that the DWARF producers and consumers have agreed upon a mapping of these operators to actual machine registers.

This method of mapping is not suitable in our environment, since our compilers generate debug information for a variety of architectures that all have different register sets. Further, we do not control the source code for consumers written by customers. To solve this problem, we extended DWARF to provide a register map that is not fixed, rather, is determined at compile time. This extension requires the following new tag:

| Tag Name | Value |
|---|---|
| DW_TAG_TI_assign_register | 0x4082 |

A series of `DW_TAG_TI_assign_register` DIEs will appear within the immediate scope of each compile unit in the program. Each DIE will have a `DW_AT_name` attribute that will be a string indicating the name of a machine register, and a `DW_AT_location` attribute that indicates the DWARF register name operator that represents it.

You may also notice some registers in this map that do not correspond to any known machine register. All of these can be safely ignored, with the exception of the one named `"CIE_RETA"`. Refer to Section 3.4.2, for more information about the use of this register.

## 4.2 Attribute Definitions

The following attribute definitions were added:

| Attribute Name | Value | Classes |
|---|---|---|
| DW_AT_TI_veneer | 0x2000 | flag |
| DW_AT_TI_symbol_name | 0x2001 | string |
| DW_AT_TI_version | 0x200b | constant |
| DW_AT_TI_asm | 0x200c | flag |
| DW_AT_TI_skeletal | 0x200e | flag |
| DW_AT_TI_interrupt | 0x2011 | flag |

`DW_AT_TI_veneer` indicates that the current function DIE is a veneer function. This attribute is unique to the TMS470.

`DW_AT_TI_symbol_name` provides the linkage name of the current DIE. This provides more information than the `DW_AT_name` attribute, which is always the simple, unqualified name of the symbol as it appears in the source code.

DW_AT_TI_version is a compile unit attribute that indicates TI's internal version stamp for the DWARF information in the current compile unit. In the future, this version number will allow for backwards compatibility with old object files. At the time of this writing, we are always generating a value of 0x1 for this attribute.

DW_AT_TI_asm is a function attribute indicating that we are describing a range of assembly code that was manually marked as a function by the assembly programmer. Functions with this attribute will always appear to have empty parameter lists, and the return type will always appear to be void. Further, there will be no call frame information for these functions in the .debug_frame section. A stack size may be indicated using the DW_AT_frame_base attribute, but its correctness is also determined by the assembly programmer.

DW_AT_TI_skeletal is a function attribute that is reserved for internal use. If a function DIE contains this attribute, it should be assumed that the debug information for that function is incomplete, and should not be used.

DW_AT_TI_interrupt is a function attribute that indicates when a function was declared using the interrupt qualifier.

### 4.3 Address Class Definitions

The following address classes were defined:

| Address Class Name | Value | Definition |
|---|---|---|
| DW_ADDR_TI_PTR8 | 0x0008 | 8-bit pointer value |
| DW_ADDR_TI_PTR16 | 0x0010 | 16-bit pointer value |
| DW_ADDR_TI_PTR22 | 0x0016 | 22-bit pointer value |
| DW_ADDR_TI_PTR23 | 0x0017 | 23-bit pointer value |
| DW_ADDR_TI_PTR24 | 0x0018 | 24-bit pointer value |
| DW_ADDR_TI_PTR32 | 0x0020 | 32-bit pointer value |

### 4.4 Call Frame Instructions

The following call frame instructions were added:

| Call Frame Instruction | High 2 Bits | Low 6 Bits | Operand 1 | Operand 2 |
|---|---|---|---|---|
| DW_CFA_TI_soffset_extended<br>DW_CFA_TI_def_cfa_soffset | 0<br>0 | 0x1c<br>0x1d | ULEB128 register<br>SLEB128 offset | SLEB128 offset |

These instructions are similar to DW_CFA_offset_extended and DW_CFA_def_cfa_offset, but take signed offset values. These instructions were needed to support architectures such as C28x™ that have a stack that grows from low memory to high memory.

## 5 References

1. TIS Committee. *DWARF Debugging Information Format Specification Version 2.0*, May 1995.