MPLAB XC16 and XC-DSC DWARF Differences

Created by Calum Wilkie, last modified less than a minute ago

Although XCxx is based upon GCC and uses the common GCC DWARF generation code, the ELF container that holds this information has some oddities. These oddities are mostly related to the way we store data in the ELF file. Here we attempt to highlight these differences so that our DWARF files may be properly decoded.

In this tutorial I will be using the XCxx compiler and some of the binary utilities included with the compiler to inspect resulting object files. I will also be using the unix tool 'od' (octal dump) to dump raw files; any other appropriate tool (for Windows users) can be substituted.

XC16 vs XC-DSC

In this document I will refer to XC16 and XC-DSC together or interchangeably. XC-DSC is the successor to XC16 targeting Microchip DSC devices. Any command line that mentions xc16 or xc-dsc, can be substituted with the other command line.

dsPIC30F and similar devices

ELF Sections

Most ELF sections are padded with null bytes. This is usually thought of for 'Flash' type sections, so that we can fit a 24-bit word into a 16-bit address hole. An oddity of the architecture is that the address increments by 2 for each program word, and each program word is 24-bits wide. Therefore a word in program memory is represented with 4 bytes of data. For example, the program word 0x112233 is encoded as 0x112233xx, where xx (the padding byte) is typically null, although a customer might specify a different padding byte value with a command option.

To allow for address compatibility between data memory and program memory, each word in data memory (16 bits) is also represented with 4 bytes of data. This is accomplished by inserting a padding byte between each original data byte. For example, the data word 0x1122 is encoded as 0x11002200. All data sections are represented in this way, including DWARF debugging sections.

If an external tool is used to interpret DWARF debugging information for PIC24, dsPIC30F, or dsPIC33C/E/F devices, then the padding bytes must be recognized and discarded.

A Short Example

Consider this short example (dwarf.c):

```
int an_int = 0xF00D;

char *a_string = "This is a string";

main() {}
```

In this example, `an_int` and `a_string` are data objects, while the value pointed to by `a_string` is usually located in Flash. `main()` exists solely to get a linked executable.

Compiling this file (with -save-temps) will leave behind dwarf.o and a.out (and some other detritus); be sure to use -g to get relevant DWARF information. We can use xc16-objdump to inspect both the object file and executable file (these are both the same kind of ELF object file). The tool xc16-readelf can be used to inspect these ELF files in a more specific way and also to decode the DWARF information into a more readable form. For our purposes today, this will not be as helpful.

I compiled this file with: `xc16-gcc -g -save-temps dwarf.c` to produce `dwarf.o` and `a.out` I then use `xc16-objdump -h` to display the section information for the object file:

```
dwarf.o:     file format elf32-pic30
Sections:
```

```
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000006  00000000  00000000  00000034  2**1
                  CONTENTS, ALLOC, LOAD, CODE
  1 .data         00000000  00000000  00000000  00000040  2**1
                  ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000040  2**1
                  ALLOC
  3 .debug_abbrev 0000004e  00000000  00000000  00000040  2**0
                  CONTENTS, DEBUGGING
  4 .debug_info   000000bb  00000000  00000000  000000dc  2**0
                  CONTENTS, RELOC, DEBUGGING
  5 .debug_line   00000033  00000000  00000000  00000252  2**0
                  CONTENTS, RELOC, DEBUGGING
  6 .ndata        00000004  00000000  00000000  000002b8  2**1
                  CONTENTS, ALLOC, LOAD, RELOC, DATA, NEAR
  7 .const        00000012  00000000  00000000  000002c0  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, PSV, PAGE
  8 .debug_frame  0000002e  00000000  00000000  000002e4  2**0
                  CONTENTS, RELOC, DEBUGGING
  9 .debug_pubnames 00000033  00000000  00000000  00000340  2**0
                  CONTENTS, RELOC, DEBUGGING
 10 .debug_aranges 00000018  00000000  00000000  000003a6  2**0
                  CONTENTS, RELOC, DEBUGGING
 11 .debug_str    00000000  00000000  00000000  000003d6  2**0
                  DEBUGGING
 12 __c30_info    00000000  00000000  00000000  000003d6  2**0
                  DEBUGGING
 13 __c30_signature 00000006  00000000  00000000  000003d6  2**0
                  CONTENTS, DEBUGGING
```

We can use the options `-s` and `-j <name>` to dump the raw data for a particular section. I will dump the data for section `.ndata` (a near data section which should contain `an_int`) and `.const` (the flash const section which should contain the data for our string):

```
dwarf.o:     file format elf32-pic30
Contents of section .ndata:
 0000 0df00000                             ....


dwarf.o:     file format elf32-pic30
Contents of section .const:
 0000 54686973 20697320 61207374 72696e67  This is a string
 0010 0000                                 ..
```

I have high-lit the value for an_int; the string is quite visible.

The 'FIle off' output from xc16-objdump gives the raw offset for the data for each file. Using a hex editor, or od (octal dump), we can inspect the raw data at each offset. Here is the offset for the `.ndata` section; the addresses are in hex and line is displayed in single byte hex and single byte (escaped) ASCII, I have highlighted the raw data starting at 0x2B8 (there are 4 bytes):

```
00002b0    04  00  00  00  01  00  01  00  0d  00  f0  00  00  00  00  00
          004  \0  \0  \0 001  \0 001  \0  \r  \0 360  \0  \0  \0  \0  \0
```

And for the `.const` section:

```
00002c0    54  00  68  00  69  00  73  00  20  00  69  00  73  00  20  00
           T  \0   h  \0   i  \0   s  \0      \0   i  \0   s  \0      \0
00002d0    61  00  20  00  73  00  74  00  72  00  69  00  6e  00  67  00
           a  \0      \0   s  \0   t  \0   r  \0   i  \0   n  \0   g  \0
00002e0    00  00  00  00  10  00  00  00  00  00  00  00  ff  00  ff  00
          \0  \0  \0  \0 020  \0  \0  \0  \0  \0  \0  \0 377  \0 377  \0
```

In each case, the real data is followed by a byte of padding (in green).

The content for a.out (the linked executable) is similar, but addresses have been filled in.

DWARF Content

How does this relate to the DWARF content? DWARF information is content, like any other, and is stored as ELF sections in a similar way.

debug_abbrev:

Contains padding bytes:

```
0000040    01  00  11  00  01  00  25  00  08  00  13  00  0b  00  03  00
         001  \0 021  \0 001  \0   %  \0  \b  \0 023  \0  \v  \0 003  \0
```

debug_info:

Hard to see, but ... contains padding bytes:

```
00000d0    0b  00  49  00  13  00  00  00  00  00  00  00  b7  00  00  00
          \v  \0   I  \0 023  \0  \0  \0  \0  \0  \0 267  \0  \0  \0
00000e0    00  00  00  00  02  00  00  00  00  00  00  00  00  00  00  00
          \0  \0  \0  \0 002  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
00000f0    04  00  01  00  47  00  4e  00  55  00  20  00  43  00  20  00
         004  \0 001  \0   G  \0   N  \0   U  \0      \0   C  \0      \0
```

and so on...

Comparing Raw data with xc16-readelf

Using xc16-readelf to dump the `.debug_info` section into a readable form, I will highlight one particular abbreviation:

```
 <1><62>: Abbrev Number: 2 (DW_TAG_subprogram)
     DW_AT_external    : 1
     DW_AT_name        : main
     DW_AT_decl_file   : 1
     DW_AT_decl_line   : 7
     DW_AT_type        : <79>
     DW_AT_low_pc      : 0 0
     DW_AT_high_pc     : 0 0
     DW_AT_frame_base  : 1 byte block: 5e       (DW_OP_reg14)

 <1><79>: Abbrev Number: 3 (DW_TAG_base_type)
     DW_AT_byte_size   : 2
     DW_AT_encoding    : 5      (signed)
     DW_AT_name        : int
```

This abbreviation is located at offset 0x62 from the start of the `.debug_info` section. This offset is in 'unpadded' bytes. To calculate the actual file offset, we can start with the file offset for the section (0xDC). The location of this entry is 0x62 unpadded bytes from the start of the section, or file offset 0xDC+(0x62*) == 0x1A0. Here is some of the raw, with padding, data from that area of the file:

```
00001a0    02  00  01  00  6d  00  61  00  69  00  6e  00  00  00  01  00
         002  \0 001  \0   m  \0   a  \0   i  \0   n  \0  \0  \0 001  \0
00001b0    07  00  79  00  00  00  00  00  00  00  00  00  00  00  00  00
          \a  \0   y  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
00001c0    00  00  00  00  00  00  00  00  00  00  01  00  5e  00  03  00
          \0  \0  \0  \0  \0  \0  \0  \0  \0  \0 001  \0   ^  \0 003  \0
```

Without going to the specifics of the format, or knowing much about the sizes of each element, we can see roughly how it all lines up. Squint, to remove the padding, and you can find each of the values (high-lit above to help with the

squinting)!

Most integers are stored in SLEB128 or ULEB128, which stores values in a variable length. The value is chunked into 7-bit quantities and emitted in least endian form; if the remaining bits are all 0 we stop emitting the 7-bit chunks, if there is more data to follow we set the high-bit in the 8-bit value. Ie, the value 0x79 would be encoded in 1 byte (0x79) but the value 0x80 would be encoded in two bytes (0x80 0x01) - the first byte gives the low 7-bits (000.0000) and says 'more to come' and the next byte gives the next 7-bits (000.0001) and says 'finshed'.

Note here that the value '0x79' which gives an offset for the function type (DW_TAG_subprogram) is unpadded.

To Make a Long Story Short

To decode the data from our DWARF sections when compiled for PIC24, dsPIC30F, or dsPIC33C/E/F devices; skip every other byte. Remember FILE offsets will refer to the padded offset, but DWARF offsets will refer to the unpadded offset.

Microchip MPLAB XC-DSC uses the same elf machine number for all dsPIC type devices. To determine if padding bytes require removal, examine the E_FLAGS field in the ELF header; bit 31 is clear to indicate that padding is included and set if it is not. All devices supported by MPLAB XC16 will have this bit clear.

The XC16 and XC-DSC compilers define the following flags in the ELF file header field:

```
#define EF_PIC30_PROC 0x0000FFFF /* 2 bytes in the e_flags field
                                    for encoding processor ID */
#define EF_PIC30_NO_PHANTOM_BYTE 0x80000000  /* set if encoding has no phantom byte */
```

If an external tool is used to interpret DWARF debugging information created by the XC16 and/or XC-DSC compilers, the EF_PIC30_NO_PHANTOM_BYTE flag can be used to determine if phantom bytes in DWARF sections must be recognized and discarded.

For example:

- When EF_PIC30_NO_PHANTOM_BYTE is clear, DWARF sections include phantom bytes.
- When EF_PIC30_NO_PHANTOM_BYTE is set, DWARF sections do not include phantom bytes.

No labels