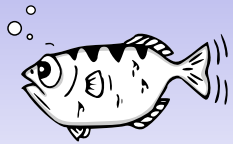


# DWARF Pieces

## And Other DWARF Location Woes

Andreas Arnez
February 3, 2018, FOSDEM

IBM

# DWARF Pieces

## Composite location description

Consists of one or more *pieces*.

Each piece is described by:

⟨*simple-location-description*⟩ ⟨*composition-operation*⟩

## Composition operations

| | | | |
|---|---|---|---|
| DW_OP_**piece** | since DWARF2 | full bytes only | no offset |
| DW_OP_**bit_piece** | since DWARF3 | bit-aligned | has bit offset |

# Why?

- Compiler optimization:
  - Some struct members in registers.
    Some replaced with constants…
    …or actually stored in memory
    …or not represented at all.
  - Same for bit fields.
  - Array spread across some vector registers.
  - High-order *bits* optimized out, to avoid sign extension.
- “Naturally” split objects, e.g.:
  - 128-bit `long double` in two 64-bit FPRs.
  - 128-bit struct in two 64-bit GPRs when passed as argument.

# What Can A Piece Be Taken From?

| **Simple location type** | **Sample DWARF** |
|---|---|
| Register | DW_OP_**regx** (37) |
| Memory | DW_OP_**breg6** (456) |
| Stack value | DW_OP_**const1u** (5); DW_OP_**stack_value** |
| Immediate value | DW_OP_**implicit_value** (0x3a 0x05 0x7b 0x51) |
| Empty (“optimized out”) | *empty* |

# Composite Location – Example

```
struct
  {
     char c;     /* constant 5 */
                 /* 3 bytes alignment gap */
     int x;      /* in register 4 */
     short y;    /* in register 4, at bit offset 32 */
  };
```

DW_OP_**lit5**
DW_OP_**stack_value**
DW_OP_**piece** (1)

DW_OP_**piece** (3)

DW_OP_**reg4**
DW_OP_**piece** (4)

DW_OP_**reg4**
DW_OP_**bit_piece** (16, 32)

# Piece Placement

## DW_OP_**piece**

*If the piece is located in a **register**, but does not occupy the entire register, the placement of the piece within that register is defined by the ABI.*

## DW_OP_**bit_piece**

*If the location is a **register**, the offset is from the least significant bit end of the register.*

# Piece Placement (2)

## DW_OP_**bit_piece**

*If the location is a **memory** address, the* DW_OP_**bit_piece** *operation describes a sequence of bits relative to the location whose address is on the top of the DWARF stack using the bit numbering and direction conventions that are appropriate to the current language on the target system.*

*If the location is any **implicit value** or **stack value**, the* DW_OP_**bit_piece** *operation describes a sequence of bits using the least significant bits of that value.*

# “Implicit Piece” Placement

Pieces are taken implicitly when type $<$ location:

| **Type** | **Location** |
|---|---|
| `short` | 64-bit general-purpose register |
| `char [3]` | 64-bit stack value |
| `float` | 128-bit vector register |
| `struct { float x; }` | 64-bit floating-point register |

Exercises:

1. Is DW_OP_**piece** / DW_OP_**bit_piece** with the type size then always a NOP?
2. What if type $>$ location?

# Composition Of Pieces

How are pieces concatenated?

- ✔ Straightforward when byte-aligned.
- ➔ Less obvious when bit-aligned.

Assumptions:

- No byte-alignment gaps.
- Next piece starts at or directly after last byte of previous piece.
- Within a byte, use ABI's bit allocation order for bit fields.





# Bit Order

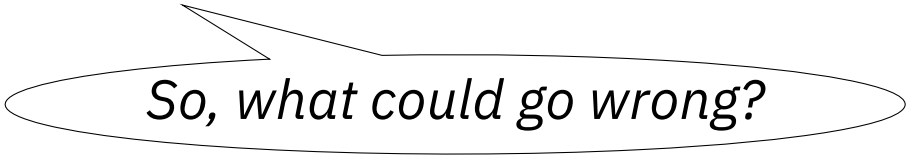
```
struct
  {
    char bit_0: 1;   /* Byte 0 */
    char bit_1: 1;
    char bit_2: 1;
    char bit_3: 1;
    char bit_4: 1;
    char bit_5: 1;
    char bit_6: 1;
    char bit_7: 1;
    char bit_8: 1;   /* Byte 1 */
    char bit_9: 1;
    …
  }
```

# Reading/Writing A Value From/To A Composite Location

In general:

| **The value…** | **Example** |
| --- | --- |
| can be somewhere within the composite location. | Struct member. |
| doesn’t need to be byte-aligned. | Bit field. |
| can spread multiple pieces. | Sub-struct. |
| doesn’t need to be piece-aligned. | Union. |

When assigning from another such value, the same applies there.
⇒ need a general bit-wise copy function.



# DWARF Piece Handling Bugs In GDB – Fixed

**Selected Fixes From 2016–2017**

1. Fix copy_bitwise()
2. write_pieced_value: Fix size capping logic
3. Take DWARF stack value pieces from LSB end
4. read/write_pieced_value: Respect value parent's offset
5. write_pieced_value: Fix copy/paste error in size calculation
6. write_pieced_value: Include transfer size in byte-wise check
7. write_pieced_value: Fix buffer offset for memory pieces
8. write_pieced_value: Transfer least significant bits into bit-field
9. Fix handling of DWARF register pieces on big-endian targets
10. Respect piece offset for DW_OP_bit_piece
11. Fix bit-/byte-offset mismatch in parameter to read_value_memory

# DWARF Location Handling Bugs in GDB – Open

**Selected Known Bugs As Of Feb 2018**

1. Pieces from vector registers on IBM Z are taken from the “least significant bits”, not compliant with ABI.
2. No support for unwinding part of a register; needed for sub-registers within a larger register.
3. DW_AT_endianity is not implemented at all.