

The DWARF Debugging Standard

Daniele Cattaneo

Politecnico di Milano

2024-04-03

Contents

- 1 Introduction**
- 2 Describing Data
- 3 Describing Code
- 4 Other information in DWARF
- 5 Data Representation
- 6 Conclusion

A normal day in a programmer's life

- Imagine you are debugging a program and you can't find a bug...
- You decide to single-step it in a **debugger**
- You get:
 - Evaluation of variables and expressions
 - Examination of stack frames
 - Current instruction pointer shown in the source code

A normal day in a programmer's life

- Imagine you are debugging a program and you can't find a bug...
- You decide to single-step it in a **debugger**
- You get:
 - Evaluation of variables and expressions
 - Examination of stack frames
 - Current instruction pointer shown in the source code
- **How does it all work?**

Inside the debugger

Debuggers consist of 2 parts:

- 1 A **monitor** component
 - Implements controlling the program
 - Start, Stop, Restart
 - **Breakpoints**, Single-Stepping...
 - Platform-dependent and operating-system dependent
- 2 An **analysis** component
 - Collects information about where things are in the program
 - Memory layout, data types...
 - Memory address for each source code line

Inside the debugger

The analysis components needs to get the information from somewhere...

And it cannot simply reimplement the compiler inside itself!

The user may have compiled their code however they liked, reversing the transformations done by the compiler is next to impossible.

We need a **data structure** that tells us where things are inside the compiled executable!

Inside the debugger

This data structure should be:

- Extensible
- Language-independent
- Compact
- Self-contained within the executable, or optionally delivered separately

Inside the debugger

We are talking about the **debugging information**

Basically it's the data that the compiler adds when you specify the -g flag

The specification that describes this data structure is called **DWARF**.

“Debugging With Attributed Record Formats”

Yeaah we all believe that...

DWARF

- Originally developed at Bell Labs together with the ELF object file format in 1988
- Standardized in 1992, but takes its current form in 1993 (DWARF 2).
 - Adoption is not instantaneous, DWARF coexists with a simpler format named “stabs”
- Becomes an open standard roughly at the turn of the century, adoption continues to grow
- Nowadays DWARF is the most used debugging format
 - All Unixes use DWARF as the standard debugging information format
 - Open-source compiler toolchains (GDB, LLVM) use DWARF in embedded development
 - DWARF is stored in standard sections in the executable, and it is loaded alongside the code: even OSes or binary formats not explicitly designed for it can support its usage

Other debugging formats

- Windows uses the CodeView debugging format. It is usually stored in separate files (.pdb extension)
- “stabs” is still available for use in most Unixes but nobody uses it anymore
- Other debugging formats exist but are either outdated or nobody uses them

The structure of DWARF

At the highest level of abstraction, DWARF consists of a **tree** structure.

Each **node** in the tree corresponds to an element of the program.

General idea is similar to a compiler's AST.

These nodes are named **DIEs** (Debugging Information Entries).

Every DIE has:

- 1 a **type** (named **tag**)
- 2 a set of **key-value** pairs containing the debug information itself

The structure of DWARF

You can examine the DIEs in an object file by using
dwarfdump

In the following discussion we will often see output from
llvm-dwarfdump

Same thing as normal GNU dwarfdump, but formats output in a clearer way

The DWARF version we will use as reference is **DWARF 4**

DWARF 5 is already out, but most compilers do not output it yet. It changes quite a number of details ~~for no reason~~ to improve efficiency.

Up next...

- 1 We will see **how DWARF models the memory of the program** using DIEs
 - Where data is stored
 - Where the code is stored
 - Other miscellaneous information
- 2 Then we will briefly go over how **DWARF serializes this information inside the object files**

Let's get started!

Contents

- 1 Introduction
- 2 Describing Data**
- 3 Describing Code
- 4 Other information in DWARF
- 5 Data Representation
- 6 Conclusion

Describing Data

A debugger like gdb is capable of **evaluating expressions**.

To do that, the debugger must be able to resolve each symbol to its current memory location, in order to read its value.

This is not as simple as it sounds!

Where is my data?

- Programming language symbols \neq symbols in executable!
 - Even though ELF already includes symbol names for several objects, they are meant for the **linker**, and are not useful to the debugger!
 - Names might have changed (some platform prepend an underscore to all symbol names)
 - static globals do not have a symbol name
 - Stack allocated variables are not symbolicated at all (the linker does not care about them)
- ⇒ DWARF must contain additional symbolication info!

Where is my data?

- The debugger does not have any intrinsic knowledge of the data types being used
 - Obvious example: custom structs
 - Non-obvious examples: floating point formats, big-endian vs. little-endian ints

⇒ DWARF must describe in detail all data types!
- A variable associated to a symbol may be moved to **different locations** during the execution of the program!
 - A frequently-used variable may be moved from the stack to a register, and then back to the stack

⇒ Each variable location must be relative to a certain region of code!

Data Types in DWARF

To support all this, DWARF uses specific kinds of DIE:

DW_TAG_base_type	A base type (int, char, float...)
DW_TAG_unspecified_type	An unknown type (void)
DW_TAG_typedef	A renamed type
DW_TAG_pointer_type	A pointer to another type
DW_TAG_array_type	An array
DW_TAG_structure	A structure
DW_TAG_union	An union
...	...

Let's look at some examples...

Data Types in DWARF

Integer scalar

```
int int_global;  
  
DW_TAG_base_type  
  DW_AT_byte_size (0x04)  
  DW_AT_encoding  (DW_ATE_signed)  
  DW_AT_name      ("int")
```

Integer pointer

```
int *p;  
  
DW_TAG_pointer_type  
  DW_AT_byte_size (0x08)  
  DW_AT_type      (0x00000042 "int")
```

Integer array

```
int int_global_array[10];  
  
DW_TAG_array_type  
  DW_AT_type      (0x00000042 "int")  
  DW_AT_sibling   (0x00000075)  
  
DW_TAG_subrange_type  
  DW_AT_type      (0x00000075  
                  "long unsigned int")  
  DW_AT_upper_bound (0x09)  
  
DW_TAG_base_type  
  DW_AT_byte_size (0x08)  
  DW_AT_encoding  (DW_ATE_unsigned)  
  DW_AT_name      ("long unsigned int")
```

Data Types in DWARF

Additional attributes (volatile, const...)
are represented by separate types

These separate types always point to the base type
(no parent-child relationships)

Constant integers

```
const int int_const = 5;
```

```
DW_TAG_base_type  
  DW_AT_byte_size (0x04)  
  DW_AT_encoding   (DWARF_ATE_signed)  
  DW_AT_name       ("int")
```

```
DW_TAG_const_type  
  DW_AT_type (0x00000042 "int")
```

Variables in DWARF

As opposed to types, **variables** & co. are represented by just 3 DIE classes:

DW_TAG_variable	A variable
DW_TAG_formal_parameter	A function parameter (i.e. inside the function)
DW_TAG_constant	A constant

Conceptually, there's nothing special here as well...

Variables in DWARF

Global integer scalar

```
int int_global;  
  
DW_TAG_variable  
  DW_AT_name   ("int_global")  
  DW_AT_type   (0x00000042 "int")  
  DW_AT_external (true)  
  DW_AT_location (DW_OP_addr 0x0)
```

Local integer scalar

```
int int_local;  
  
DW_TAG_variable  
  DW_AT_name   ("int_local")  
  DW_AT_type   (0x00000042 "int")  
  DW_AT_location (DW_OP_fbreg -28)
```

- Apart from the obvious name and type attributes, the `external` and `location` attributes appeared
- `external` specifies if the symbol is visible from outside the compilation unit
- `location` specifies where the data is located using a **DWARF expression**

DWARF expressions

- DWARF expressions are composed of **sequences of commands**
- They are interpreted by a simple **stack-based evaluator**
- Commands can optionally take parameters, use **prefix notation**
- There are commands for branches, function calls...
- The values in the evaluator are either **void pointers** or **base types** supported by the target machine
- All results of commands are pushed on top of the stack
- Commands are evaluated in the order they appear (bar jumps / calls)
- It's a mini assembly language!

DWARF expressions

DWARF expressions are powerful, but this power is seldom used or needed.

Most used commands:

DW_OP_fbreg x	The value of the current frame pointer plus x
DW_OP_reg n	The location of register n (0 to 31, represented in an opaque encoding)
DW_OP_addr x	The absolute address x

Remember: the DWARF data sections are
linked together with the rest of the program



Any address that appears in them can be fixed up
by a static or a dynamic linker!

DWARF Location Descriptors

Actually, DW_AT_location attributes can be associated to two kinds of values:

- 1 One DWARF expression describing where the data is allocated
(*Single location descriptions*)
Used when the data is always allocated in the same place
- 2 A map between code ranges and DWARF expressions
(*Location lists*)
Used when the data changes its location during its lifetime

DWARF Location Descriptors

C code

```
int *p = buf;
int a = 0;
for (int i=0; i<10; i++) {
    a += *p++;
}
```

Debug Info

```
DW_TAG_variable
DW_AT_name      ("p")
DW_AT_type      (0x000000117 "int*")
DW_AT_location (0x000000072
    [0x22, 0x2b): DW_OP_reg3 RBX
    [0x2b, 0x3c): DW_OP_reg5 RDI)
```

Assembly Code (-g -O2)

```
5: leaq 16(%rsp), %rbx
22: leaq 40(%rbx), %rdx    p: %rbx
26: xorl %eax, %eax       p: %rbx
28: movq %rbx, %rdi       p: %rbx
/-> 30: addq $4, %rdi       p: %rdi
| 34: addl -4(%rdi), %eax  p: %rdi
| 37: cmpq %rdx, %rdi     p: %rdi
\-- 3a: jne -12 <main+0x30> p: %rdi
3c: leaq 12(%rsp), %rdi
41: movl %eax, 12(%rsp)
```

GCC optimized out the `i` variable and moved `p` to two different registers.

DWARF Declaration Coordinates

All of these DIEs (along with many others) support specifying where each type is declared using specific attributes:

DW_AT_decl_file	File where the declaration was found
DW_AT_decl_line	Line number (1-based) of the declaration
DW_AT_decl_column	Column number (1-based)

Declaration Coordinates

```
DW_TAG_variable
  DW_AT_name      ("int_global")
  DW_AT_decl_file  ("/media/sf_example/dwarf-test.c")
  DW_AT_decl_line  (1)
  DW_AT_type       (0x00000042 "int")
  DW_AT_external   (true)
  DW_AT_location   (DW_OP_addr 0x0)
```

Contents

- 1 Introduction
- 2 Describing Data
- 3 Describing Code**
- 4 Other information in DWARF
- 5 Data Representation
- 6 Conclusion

Describing Code

We have seen how DWARF describes **data**. Of equal importance in a debugger is analyzing the **code being executed**.

Obvious requirements:

- Map to instruction to line of code in the source code
 - Essential for breakpoints, single stepping

Less obvious requirements:

- Map from code spans to compilation units
- Map from code spans to functions
- Identification of every function call in a function
 - Used for breakpoints on functions, to parse backtraces, to resolve imported functions
- Grouping of lexical blocks in a function
 - Used to determine the scope of variables

Describing Code

- Compilation units, functions and function calls are part of the DIE tree
 - Just like a compiler AST!
- The line number table is its own separate structure
 - This helps saving space inside the executable, as DIEs are not as space-efficient

Describing Code: Compilation Units

A compilation unit (`DW_TAG_compile_unit`) represents the result of a single compilation process (duh).

Common attributes:

<code>DW_AT_name</code>	File name that was compiled
<code>DW_AT_comp_dir</code>	Working directory where the compiler was invoked
<code>DW_AT_language</code>	Language of the source code file
<code>DW_AT_stmt_list</code>	Pointer to the line number table for this file
<code>DW_AT_low_pc</code>	Range of addresses of the compilation unit's code
<code>DW_AT_high_pc</code>	

Describing Code: Functions

A subprogram (DW_TAG_subprogram) represents a function. Function arguments are described by children DW_TAG_formal_parameter DIEs.

Common attributes:

DW_AT_name	Function name (as it appears in the source)
DW_AT_type	Type of the return value
DW_AT_frame_base	Expression establishing the address of the frame pointer (DW_OP_fbreg)
DW_AT_low_pc	Range of addresses of the function's code
DW_AT_high_pc	

Describing Code: Blocks

DW_TAG_lexical_blocks DIEs represent a block, in languages that support this concept. Not strictly required, but helps the debugger in handling local variables with a restricted scope.

Common attributes:

DW_AT_low_pc	Range of addresses of the function's code
DW_AT_high_pc	

And that's it!

Note that every time the DW_AT_low_pc and DW_AT_high_pc attributes are used together, they can be replaced by a single DW_AT_ranges attribute.

DW_AT_ranges supports describing multiple discontinuous ranges.

Describing Code: Line Numbers

The **line number table** is its own data structure.

- Maps each instruction to a line number and more
- Consecutive instructions mapping to the same line number are represented by just the first instruction
- Internal representation consists of a **line number program** that builds the table (and allows for compact representation of similar rows)
 - We will enter into details about this representation later

The GNU assembler supports building this data automatically using specific `.loc` directives.

Describing Code: Line Numbers

Address Address of the instruction

Op. Index 0 for non-VLIW architectures

File File where the line is found (index in a list of files)

Line Line in the file (1-based)

Column Column in the file (1-based)

Is Statement True if position is at the beginning of a statement

Basic Block True if at the beginning of a basic block

End Sequence True in the last row of the table (ending marker)

Prologue End True in the last instruction of a function prologue

Epilogue Begin True in the first instruction of a function epilogue

ISA The instruction set of the code

Discriminator Disambiguates multiple blocks on the same line

Contents

- 1 Introduction
- 2 Describing Data
- 3 Describing Code
- 4 Other information in DWARF**
- 5 Data Representation
- 6 Conclusion

Other information in DWARF

DWARF also provides some extra useful but not essential information:

- Call Frame Information
- Macro information (!)
- Identifier index for faster lookups

Call Frame Information

DWARF Call Frame Information (aka CFI) allows the debugger to reliably perform backtraces.

Why we need it?

- 1 The ABI already provides a way to perform backtraces without it, but it prevents optimizations like tail calls or (in the x86 architecture) using the EBP register for other purposes
- 2 DIs by themselves do not contain any information about the size of the activation record of each function, just how to access variable values

Call Frame Information

DWARF defines an activation record as the following set of values:

- The return address (for past frames) or the current PC (for the current frame)
- The area of memory in the stack corresponding to the call frame of the function (described as a single base pointer: the Canonical Frame Address)
- The set of registers that are preserved across the call corresponding to the activation record

Call Frame Information

The CFI maps each instruction in the program to a table that specifies:

- how to reconstruct each register value from the activation record alone, as it appears in that point of the subroutine
- where the Canonical Frame Address pointer is

This information is compressed using a bytecode system similar to DWARF expressions and the line number table.

The GNU assembler supports building this data inline (like the line number table) using specific `.cfi_XXX` directives.

Fun fact: the same tables used by DWARF are used as a method for implementing stack unwinding for C++ exceptions (hence why they are so slow!)

Macro Information

You wish you could **debug your C macros**?

DWARF includes a **Macro Information** section which specifies to the debugger where a macro declaration can be found in the source.

There is nothing else in there... which limits its usefulness. It's also 100% optional.

The format of macro information has changed in DWARF 5.

Accelerated Access

DWARF has a facility for building an index of symbols to make lookup of large DIE trees easier. It's implemented as a hashtable.

The index can contain references to:

- Subprograms
- Labels
- Variables
- Types
- Namespaces

(we didn't see some of these DIEs earlier, but they exist)

The index must be complete, this simplifies the lookup logic in case it exists (it's optional).

Just like with macro information, the format of the index has changed in DWARF 5.

Contents

- 1 Introduction
- 2 Describing Data
- 3 Describing Code
- 4 Other information in DWARF
- 5 Data Representation**
- 6 Conclusion

The physical reality (?) of DWARF

DWARF debug info is scattered in different sections, whose names all start with `.debug`:

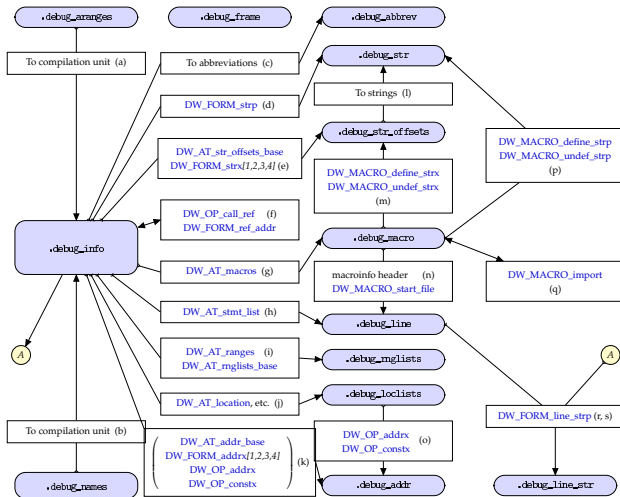
<code>.debug_info, .debug_abbrev</code>	DIE tree
<code>.debug_line</code>	Line number table
<code>.debug_frame</code>	Call Frame Information
<code>.debug_macro</code>	Macro information
<code>.debug_pubnames, .debug_pubtypes</code>	Index

The physical reality (?) of DWARF

The information for some structures is **scattered further** (for a mix of backwards compatibility and data compactness reasons) into **more sections**:

<code>.debug_addr</code>	Constant pool for DWARF expressions
<code>.debug_aranges</code>	Address ranges for compilation units
<code>.debug_line_str</code>	String pool for <code>.debug_line</code>
<code>.debug_loc</code>	Location lists addressed by <code>.debug_info</code>
<code>.debug_ranges</code>	Range list addressed by <code>.debug_loc</code>
<code>.debug_str</code>	String pool for <code>.debug_info</code>
...	...

A simple and logical organization



A simple and logical organization

We will focus on:

- Representation of the main DIE structure
- The line number table

These two structures are the most important anyway.

The .debug_info section

This section contains the main structure of DWARF: the tree of Debug Information Entries.

It contains:

- 1 An header
- 2 The serialized DIEs, enumerated in depth-first order
 - Each DIE has a flag specifying if it has children
 - A special flag marks the end of a chain of children

This structure is repeated for each compilation unit in the file (in the case of a non-linked object file there is always just one compilation unit)

The .debug_info section

The Header

The header of a .debug_info is simple:

uint32_t length;	Size of the info for this unit
uint16_t version;	DWARF version
uint32_t abbrev_offset;	Offset in the abbreviation table
uint8_t address_size;	Size of a pointer (in bytes)

Wait... what's an **abbreviation table**?

The .debug_abbrev section

The **abbreviation table** is the main element of the **compression scheme** used by DWARF to represent DIEs.

- The typical DIEs produced by a compiler have similarities
 - The same type of DIE often has the same type of attributes
 - Similarly structured DIEs will have different attributes values
- The format of a DIE may change in the future
 - We need to allow old implementations to read new DWARF records without getting confused

The abbreviation table **decouples the tag, the attributes, the order of appearance of the attributes, and the type of each attribute in the DIE from the values of the attributes**

The .debug_abbrev section

The abbreviation table contains a flat list of DIE patterns, in the following format:

Abbreviation table in pseudo-C

```
struct debug_abbrev {  
    struct {  
        uleb128 id;  
        uint8_t has_children;  
        struct {  
            uleb128 attr_id;  
            uleb128 format_id;  
        } [M];  
        uint16_t terminator = 0;  
    } [N];  
    uint8_t terminator = 0;  
}
```

- M is the number of attribute-value pairs in the abbreviated DIE, it changes for each abbreviation
- N is the total number of abbreviations

The uleb128 format

uleb128 is a **variable length numeric representation** used in DWARF for compactness.

It reduces the number of leading zero bytes in the representation by using bit 7 as a stop marker. Maximum length of 128 bits gives it its name.

- Bit 7 = 0 \Rightarrow This is the last and most significant byte
- Bit 7 = 1 \Rightarrow This is not the last byte, read another one

uleb128 example

```
0x0001 = 0b00000000_00000000_00000001 = 0x01
0x007F = 0b00000000_00000000_11111111 = 0x7F
0x0080 = 0b00000000_00000001_00000000 = 0x80 0x01
0xAA55 = 0b00000010_1010100_1010101 = 0xD5 0xD4 0x02
```

The .debug_info section

The Body

Back to the .debug_info section...

After the header, for each DIE the section contains:

- The identifier of the matching abbreviation, encoded as uleb128
- The values of the attributes, in the same order as in the abbreviation
 - Not all attribute types use the same number of bytes! There is no alignment, and some types use **zero bytes**.

Abbreviation number zero does not exist; zero is used as the **end-of-children mark**.

Whether a DIE has children or not, it's specified in the abbreviation!

The `.debug_line` section

In contrast with the `.debug_info` section, the `.debug_line` section is much simpler. It encodes the **line number table**.

It begins with a header specifying:

- Length, version (like `.debug_info`)
- Whether by default each line contains a statement (usually true)
- The list of include directories used by the compiler
- The list of files that are referenced by the line number table

(not an exhaustive list)

The list of files and include directories are specified as offsets of strings in the `.debug_line_str` section.

The .debug_line section

After the header, the **line number program** begins.

Every instruction can either:

- Modify a cell in the current row of the table
 - Each opcode corresponds to a different column
 - DW_LNS_advance_pc, DW_LNS_advance_line, DW_LNS_set_file,
...
- Create the next row by (mostly) copying the next row
 - DW_LNS_copy
 - The **Basic Block**, **Prologue End**, **Epilogue Begin** and **Discriminator** rows are set to zero regardless
- Instructions either take no operands, or a single uleb128.

Contents

- 1 Introduction
- 2 Describing Data
- 3 Describing Code
- 4 Other information in DWARF
- 5 Data Representation
- 6 Conclusion**

Conclusion

We have seen:

- A brief history of debugging formats, why they are needed, what DWARF is
- What is the big idea behind DWARF
- How DWARF is actually represented inside an object file

Conclusion

With this knowledge you could:

- Build your own compiler from scratch with debugging info support
- Build your own source-level debugger
- Improve debugging support in other compilers

One last note!

If you are a LLVM user... the implementation details are **abstracted** from you!

LLVM represents debug information using a more generic data structure known as **metadata**.

Debug information is important, and it is **up to the passes** to maintain its correctness, so keep this in mind when you work with LLVM.

LLVM documentation about debugging information:
<https://llvm.org/docs/SourceLevelDebugging.html>

I want to know more about dwarves!

`http://dwarfstd.org`

Or get in contact with your local DM ;)

Thank You!

Questions?

Bibliography I



Michael J. Eager.

Introduction to the dwarf debugging format.
2010.



DWARF Debugging Information Format – Version 4.

DWARF Debugging Information Format Committee, 2010.



Dwarf debugging standard website.

<http://dwarfstd.org>.