Introduction to the DWARF Debugging Format

Michael J. Eager, Eager Consulting
April, 2012

It would be wonderful if we could write programs that were guaranteed to work correctly and never needed to be debugged. Until that halcyon day, the normal programming cycle is going to involve writing a program, compiling it, executing it, and then the (somewhat) dreaded scourge of debugging it. And then repeat until the program works as expected.

It is possible to debug programs by inserting code that prints values of selected interesting variables. Indeed, in some situations, such as debugging kernel drivers, this may be the preferred method. There are low-level debuggers that allow you to step through the executable program, instruction by instruction, displaying registers and memory contents in binary.

But it is much easier to use a source-level debugger which allows you to step through a program's source, set breakpoints, print variable values, and perhaps a few other functions such as allowing you to call a function in your program while in the debugger. The problem is how to coordinate two completely different programs, the compiler and the debugger, so that the program can be debugged.

Translating from Source to Executable

The process of compiling a program from human-readable form into the binary form that a processor executes is quite complex, but it essentially involves successively recasting the source into simpler and simpler forms, discarding information at each step until, eventually, the result is the sequence of simple operations, registers, memory addresses, and binary values which the processor actually understands. After all, the processor really doesn't care whether you used object oriented programming, templates, or smart pointers; it only understands a very simple set of operations on a limited number of registers and memory locations containing binary values.

Michael Eager is Principal Consultant at Eager Consulting (www.eagercon.com), specializing in development tools for embedded systems. He was a member of PLSIG's DWARF standardization committee and has been Chair of the DWARF Standards Committee since 1999. Michael can be contacted at eager@eagercon.com.

As a compiler reads and parses the source of a program, it collects a variety of information about the program, such as the line numbers where a variable or function is declared or used. Semantic analysis extends this information to fill in details such as the types of variables and arguments of functions. Optimizations may move parts of the program around, combine similar pieces, expand inline functions, or remove parts which are unneeded. Finally, code generation takes this internal representation of the program and generates the actual machine instructions. Often, there is another pass over the machine code to perform what are called "peephole" optimizations that may further rearrange or modify the code, for example, to eliminate duplicate instructions.

All-in-all, the compiler's task is to take the well-crafted and understandable source code and convert it into efficient but essentially unintelligible machine language. The better the compiler achieves the goal of creating tight and fast code, the more likely it is that the result will be difficult to understand.

During this translation process, the compiler collects information about the program which will be useful later when the program is debugged. There are two challenges to doing this well. The first is that in the later parts of this process, it may be difficult for the compiler to relate the changes it is making to the program to the original source code that the programmer wrote. For example, the peephole optimizer may remove an instruction because it was able to switch around the order of a test in code that was generated by an inline function in the instantiation of a C++ template. By the time it gets its metaphorical hands on the program, the optimizer may have a difficult time connecting its manipulations of low-level code to the original source which generated it.

The second challenge is how to describe the executable program and its relationship to the original source with enough detail to allow a debugger to provide the programmer useful information. At the same time, the description has to be concise enough so that it does not take up an extreme amount of space or require significant processor time to interpret. This is where the DWARF Debugging Format comes in: it is a compact representation of the relationship between the executable program and the source in a way that is reasonably efficient for a debugger to process.

The Debugging Process

When a programmer runs a program under a debugger, there are some common operations which he or she may want to do. The most common of these are setting a breakpoint to stop the debugger at a particular point in the source, either by specifying the line number or a function name. When this breakpoint is hit, the programmer usually would like to display the values of local or global variables, or the arguments to the function. Displaying the call stack lets the programmer know how the program arrived at the breakpoint in cases where there are multiple execution paths. After reviewing this information, the programmer can ask the debugger to continue execution of the program under test.

There are a number of additional operations that are useful in debugging. For example, it may be helpful to be able to step through a program line by line, either entering or stepping over called functions. Setting a breakpoint at every instance of a template or inline function can be important for debugging C++ programs. It can be helpful to stop just before the end of a function so that the return value can be displayed or changed. Sometimes the programmer may want to bypass execution of a function, returning a known value instead of what the function would have (possibly incorrectly) computed.

There are also data related operations that are useful. For example, displaying the type of a variable can avoid having to look up the type in the source files. Displaying the value of a variable in different formats, or displaying a memory or register in a specified format is helpful.

There are some operations which might be called advanced debugging functions: for example, being able to debug multi-threaded programs or programs stored in read-only memory. One might want a debugger (or some other program analysis tool) to keep track of whether certain sections of code had been executed or not. Some debuggers allow the programmer to call functions in the program being tested. In the not-so-distant past, debugging programs that had been optimized would have been considered an advanced feature.

The task of a debugger is to provide the programmer with a view of the executing program in as natural and understandable fashion as possible, while permitting a wide range of control over its execution. This means that the debugger has to essentially reverse much of the compiler's carefully crafted transformations, converting the program's data and state back into the terms that the programmer originally used in the program's source.

The challenge of a debugging data format, like DWARF, is to make this possible and even easy.

Debugging Formats

There are several debugging formats: stabs, COFF, PE-COFF, OMF, IEEE-695, and two variants[1] of DWARF, to name some common ones. I'm not going to describe these in any detail. The intent here is only to mention them to place the DWARF Debugging Format in context.

The name *stabs* comes from *s*ymbol *tab*le *s*trings, since the debugging data were originally saved as strings in Unix's *a.out* object file's symbol table. Stabs encodes the information about a program in text strings. Initially quite simple, stabs has evolved over time into a quite complex, occasionally cryptic and less-than-consistent debugging format. Stabs is not standardized nor well documented[2]. Sun Microsystems has made a number of extensions to stabs. GCC has made other extensions, while attempting to reverse engineer the Sun extensions. Nonetheless, stabs is still widely used.

COFF stands for Common Object File Format and originated with Unix System V Release 3. Rudimentary debugging information was defined with the COFF format, but since COFF includes support for named sections, a variety of different debugging formats such as stabs have been used with COFF. The most significant problem with COFF is that despite the Common in its name, it isn't the same in each architecture which uses the format. There are many variations in COFF, including XCOFF (used on IBM RS/6000), ECOFF (used on MIPS and Alpha), and Windows PE-COFF. Documentation of these variants is available to varying degrees but neither the object module format nor the debugging information is standardized.

PE-COFF is the object module format used by Microsoft Windows beginning with Windows 95. It is based on the COFF format and contains both COFF debugging data and Microsoft's own proprietary CodeView or CV4 debugging data format. Documentation on the debugging format is both sketchy and difficult to obtain.

OMF stands for Object Module Format and is the object file format used in CP/M, DOS and OS/2 systems, as well as a small number of embedded systems. OMF defines public name and line number information for debuggers and can also contain Microsoft CV, IBM PM, or AIX format debugging data. OMF only provides the most rudimentary support for debuggers.

IEEE-695 is a standard object file and debugging format developed jointly by Microtec Research and HP in the late 1980's for embedded environments. It became an IEEE standard in 1990. It is a very flexible specification, intended to be usable with almost any machine architecture. The debugging format is block structured, which corresponds to the organization of the source better than other formats. Although it is an IEEE standard, in many ways IEEE-695 is more like the proprietary formats. Although the original standard is readily available from IEEE, Microtec Research made extensions to support C++ and optimized code which are poorly documented. The IEEE standard was never revised to incorporate the Microtec Research or other changes. Despite being an IEEE standard, it's use is limited to a few small processors.

A Brief History of DWARF

DWARF 1 — Unix SVR4 *sdb* and PLSIG

DWARF[3] was developed by Brian Russell, Ph.D., at Bell Labs in 1988 for use with the C compiler and *sdb* debugger in Unix System V Release 4 (SVR4). The Programming Languages Special Interest Group (PLSIG), part of Unix International (UI), documented the DWARF generated by SVR4 as DWARF Version 1 in 1992. Although the original DWARF had several clear shortcomings, most notably that it was not very compact, the PLSIG decided to standardize the SVR4 format with only minimal modification. It was widely adopted within the embedded sector where it continues to be used today, especially for small processors.

DWARF 2 — PLSIG

The PLSIG continued to develop and document extensions to DWARF to address several issues, the most important of which was to reduce the size of debugging data that were generated. There were also additions to support new languages such as the up-and-coming C++ language. DWARF Version 2 was released as a draft standard in 1993.

In an example of the domino theory in action, shortly after PLSIG released the draft standard, fatal flaws were discovered in Motorola's 88000 microprocessor. Motorola pulled the plug on the processor, which in turn resulted in the demise of Open88, a consortium of companies that were developing computers using the 88000. Open88 in turn was a supporter of Unix International, sponsor of PLSIG, which resulted in UI being disbanded. When UI folded, all that remained of the PLSIG was a mailing list and a variety of ftp sites that had various versions of the DWARF 2 draft standard. A final standard was never released.

Since Unix International had disappeared and PLSIG disbanded, several organizations independently decided to extend DWARF 1 and 2. Some of these extensions were specific to a single architecture, but others might be applicable to any architecture. Unfortunately, the different organizations didn't work together on these extensions. Documentation on the extensions is

[1] DWARF Version 1 is significantly different from Versions 2 and later.

[2] In 1992, the author wrote an extensive document describing the stabs generated by Sun Microsytems' compilers. Unfortunately, it was never widely distributed.

[3] The name DWARF is something of a pun, since it was developed along with the ELF object file format. The name is an acronym for "Debugging With Arbitrary Record Formats".

generally spotty or difficult to obtain. Or as a GCC developer might suggest, tongue firmly in cheek, the extensions were well documented: all you have to do is read the compiler source code. DWARF was well on its way to following COFF and becoming a collection of divergent implementations rather than being an industry standard.

DWARF 3 — Free Standards Group

Despite several on-line discussions about DWARF on the PLSIG email list (which survived under X/Open [later Open Group] sponsorship after UI's demise), there was little impetus to revise (or even finalize) the document until the end of 1999. At that time, there was interest in extending DWARF to have better support for the HP/Intel IA-64 architecture as well as better documentation of the ABI used by C++ programs. These two efforts separated, and the author took over as Chair for the revived DWARF Committee.

Following more than 18 months of development work and creation of a draft of the DWARF 3 specification, the standardization effort hit what might be called a soft patch. The committee (and this author, in particular) wanted to insure that the DWARF standard was readily available and to avoid the possible divergence caused by multiple sources for the standard. The DWARF Committee became the DWARF Workgroup of the Free Standards Group in 2003. Active development and clarification of the DWARF 3 Standard resumed early in 2005 with the goal to resolve any open issues in the standard. A public review draft was released to solicit public comments in October and the final version of the DWARF 3 Standard was released in December, 2005.

DWARF 4 — DWARF Debugging Format Committee

After the Free Standards Group merged with Open Source Development Labs (OSDL) in 2007 to form the Linux Foundation, the DWARF Committee returned to independent status and created its own web site at dwarfstd.org. Work began on Version 4 of the DWARF in 2007. This version clarified DWARF expressions, added support for VLIW architectures, improved language support, generalized support for packed data, added a new technique for compressing the debug data by eliminating duplicate type descriptions, and added support for profile-based compiler optimizations, as well as extensive editing of the documentation. The DWARF Version 4 Standard was released in June, 2010, following a public review.

Work on DWARF Version 5 started in February, 2012. This version is expected to be completed in 2014.

DWARF Overview[4]

Most modern programming languages are block structured: each entity (a class definition or a function, for example) is contained within another entity. Each file in a C program may contain multiple data definitions, multiple variable definitions, and multiple functions. Within each C function there may be several data definitions followed by executable statements. A statement may be a compound statement that in turn can contain data definitions and executable statements. This creates lexical scopes, where names are known only within the scope in which they are defined. To find the definition of a particular symbol in a program, you first look in the current scope, then in successive enclosing scopes until you find the symbol. There may be multiple definitions of the same name in different scopes. Compilers very naturally represent a program internally as a tree.

DWARF follows this model in that it is also block structured. Each descriptive entity in DWARF (except for the topmost entry which describes the source file) is contained within a parent entry and may contain children entities. If a node contains multiple entities, they are all siblings, related to each other. The DWARF description of a program is a tree structure, similar to the compiler's internal tree, where each node can have children or siblings. The nodes may represent types, variables, or functions. This is a compact format where only the information that is needed to describe an aspect of a program is provided. The format is extensible in a uniform fashion, so that a debugger can recognize and ignore an extension, even if it might not understand its meaning. (This is much better than the situation with most other debugging formats where the debugger gets fatally confused attempting to read unrecognized data.) DWARF is also designed to be extensible to describe virtually any procedural programming language on any machine architecture, rather than being bound to only describing one language or one version of a language on a limited range of architectures.

[4] In the remainder of this paper, we will be discussing DWARF Version 2 and later versions. Unless otherwise noted, all descriptions apply to DWARF Versions 2 through 4.

While DWARF is most commonly associated with the ELF object file format, it is independent of the object file format. It can and has been used with other object file formats. All that is necessary is that the different data sections that make up the DWARF data be identifiable in the object file or executable. DWARF does not duplicate information that is contained in the object file, such as identifying the processor architecture or whether the file is written in big-endian or little-endian format.

Debugging Information Entry (DIE)

Tags and Attributes

The basic descriptive entity in DWARF is the Debugging Information Entry (DIE). A DIE has a *tag*, which specifies what the DIE describes and a list of *attributes* which fill in details and further describes the entity. A DIE (except for the topmost) is contained in or owned by a parent DIE and may have sibling DIEs or children DIEs. Attributes may contain a variety of values: constants (such as a function name), variables (such as the start address for a function), or references to another DIE (such as for the type of a function's return value).

Figure 1 shows C's classic `hello.c` program with a simplified graphical representation of its DWARF description. The topmost DIE represents the compilation unit. It has two "children", the first is the DIE describing *main* and the second describing the base type *int* which is the type of the value returned by *main*. The subprogram DIE is a child of the compilation unit DIE, while the base type DIE is referenced by the Type attribute in the subprogram DIE. We also talk about a DIE "owning" or "containing" the children DIEs.

Types of DIEs

DIEs can be split into two general types. Those that describe data including data types and those that describe functions and other executable code.

Describing Data and Types

Most programming languages have sophisticated descriptions of data. There are a number of built-in data types, pointers, various data structures, and usually ways of creating new data types. Since DWARF is intended to be used with a vari-

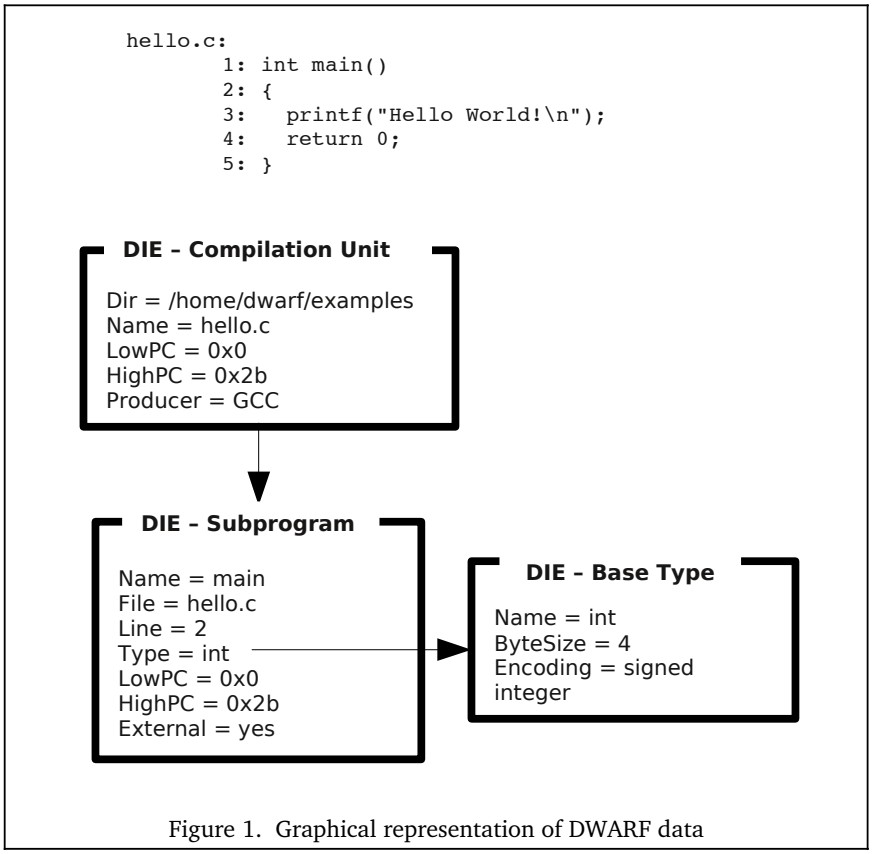

Figure 1. Graphical representation of DWARF data

ety of languages, it abstracts out the basics and provides a representation that can be used for all supported language. The primary types, built directly on the hardware, are the *base types*. Other data types are constructed as collections or compositions of these base types.

Base Types

Every programming language defines several basic scalar data types. For example, both C and Java define *int* and *double*. While Java provides a complete definition for these types, C only specifies some general characteristics, allowing the complier to pick the actual specifications that best fit the target processor. Some languages, like Pascal, allow new base types to be defined, for example, an integer type which can hold integer values between 0 and 100. Pascal doesn't specify how this should be implemented. One compiler might implement this as a single byte, another might use a 16-bit integer, a third might implement all integer types as 32-bit values no matter how they are defined.

With DWARF Version 1 and other debugging formats, the compiler and debugger are supposed to share a common understanding about whether an *int* is 16, 32, or even 64 bits. This becomes awkward when the same hardware can support different size integers or when different compilers make different implementation decisions for the same target processor. These assumptions, often undocumented, make it difficult to have compatibility between different compilers or debuggers, or even between different versions of the same tools.

```
DW_TAG_base_type
        DW_AT_name = int
        DW_AT_byte_size = 4
        DW_AT_encoding = signed
```

Figure 2a. *int* base type on 32-bit processor.

```
DW_TAG_base_type
        DW_AT_name = int
        DW_AT_byte_size = 2
        DW_AT_encoding = signed
```

Figure 2b. *int* base type on 16-bit processor

DWARF base types provide the lowest level mapping between the simple data types and how they are implemented on the target machine's hardware. This makes the definition of *int* explicit for both Java and C and allows different definitions to be used, possibly even within the same program. Figure 2a shows the DIE which describes *int* on a typical 32-bit processor. The attributes specify the name (*int*), an encoding (signed binary integer), and the size in bytes (4). Figure 2b shows a similar definition of *int* on a 16-bit processor. (In Figure 2, we use the tag and attribute names defined in the DWARF standard, rather than the more informal names used in Figure 1. The names of tags are all prefixed with *DW_TAG* and the names of attributes are prefixed with *DW_AT*.)

The base types allow the compiler to describe almost any mapping between a programming language scalar type and how it is actually implemented on the processor. Figure 3 describes a 16-bit integer value that is stored in the upper 16 bits of a four byte word. In this base type, there is a bit size attribute that specifies that the value is 16 bits wide and an offset from the high-order bit of zero[5].

The DWARF base types allow a number of different encodings to be described, including address, character, fixed point, floating point, and packed decimal, in addition to binary integers. There is still a little ambiguity remaining: for example, the actual encoding for a floating point number is not specified; this is determined by the encoding that the hardware actually supports. In a processor which supports both 32-bit and 64-bit floating point values following the IEEE-754 standard, the encodings represented by “float” are different depending on the size of the value.

```
DW_TAG_base_type
        DW_AT_name = word
        DW_AT_byte_size = 4
        DW_AT_bit_size = 16
        DW_AT_bit_offset = 0
        DW_AT_encoding = signed
```

Figure 3. 16-bit *word* type stored in the top 16-bits of a 32-bit word.

Type Composition

A named variable is described by a DIE which has a variety of attributes, one of which is a reference to a type definition. Figure 4 describes an integer variable named *x*. (For the moment we will ignore the other information that is usually contained in a DIE describing a variable.)

The base type for *int* describes it as a signed binary integer occupying four bytes.

[5] This is a real-life example taken from an implementation of Pascal that passed 16-bit integers in the top half of a word on the stack.

The DW_TAG_variable DIE for *x* gives its name and a type attribute, which refers to the base type DIE. For clarity, the DIEs are labeled sequentially in the this and following examples; in the actual DWARF data, a reference to a DIE is the offset from the start of the compilation unit where the DIE can be found. References can be to previously defined DIEs, as in Figure 4, or to DIEs which are defined later. Once we have created a base type DIE for *int*, any variable in the same compilation can reference the same DIE[6].

DWARF uses the base types to construct other data type definitions by composition. A new type is created as a modification of another type. For example, Figure 5 shows a pointer to an *int* on our typical 32-bit machine. This DIE defines a pointer type, specifies that its size is four bytes, and in turn references the *int* base type. Other DIEs describe the *const* or *volatile* attributes, C++ reference type, or C *restrict* types. These type DIEs can be chained together to describe more complex data types, such as "`const char **argv`" which is described in Figure 6.

```
<1>: DW_TAG_base_type
       DW_AT_name = int
       DW_AT_byte_size = 4
       DW_AT_encoding = signed

<2>: DW_TAG_variable
       DW_AT_name = x
       DW_AT_type = <1>
```

Figure 4. DWARF description of "`int x`".

```
<1>: DW_TAG_variable
       DW_AT_name = px
       DW_AT_type = <2>

<2>: DW_TAG_pointer_type
       DW_AT_byte_size = 4
       DW_AT_type = <3>

<3>: DW_TAG_base_type
       DW_AT_name = int
       DW_AT_byte_size = 4
       DW_AT_encoding = signed
```

Figure 5. DWARF description of "`int *px`".

Array

Array types are described by a DIE which defines whether the data is stored in *column major* order (as in Fortan) or in *row major* order (as in C or C++). The index for the array is represented by a *subrange type* that gives the lower and upper bounds of each dimension. This allows DWARF to describe both C style arrays, which always have zero as the lowest index, as well as arrays in Pascal or Ada, which can have any value for the low and high bounds.

Structures, Classes, Unions, and Interfaces

Most languages allow the programmer to group data together into structures (called *struct* in C and C++, *class* in C++, and *record* in Pascal). Each of the components of the structure generally has a unique name and may have a different type, and each occupies its own space. C and C++ have the *union* and Pascal has the *variant record* that are similar to a structure except that the component occupy the same memory locations. The Java *interface* has a subset of the properties of a C++ *class*, since it may only have abstract methods and constant data members.

Although each language has its own terminology (C++ calls the components of a class *members* while Pascal calls them *fields*) the underlying organization can be described in DWARF. True to its heritage, DWARF uses the C/C++/Java terminology and has DIEs which describe *struct*, *union*, *class*, and *interface*. We'll describe the *class* DIE here, but the others have essentially the same organization.

The DIE for a *class* is the parent of the DIEs which describe each of the class's *members*. Each *class* has a name and possibly other attributes. If the size of an instance is known at compile time, then it will have a byte size attribute. Each of these descriptions looks very much like the description of a simple variable, although there may be some additional attributes. For example, C++ allows the programmer to specify whether a member is *public*, *private*, or *protected*. These are described with the *accessibility* attribute.

C and C++ allow bit fields as class members that are not simple variables. These are described with *bit offset* from the start of the class instance to the left-most bit of the bit field and *bit size* that says how many bits the member occupies.

Variables

Variables are generally pretty simple. They have a name which represents a chunk of memory (or register) that can contain some kind of a value. The kind of values that the variable can contain, as well as restrictions on how it can be modified (e.g., whether it is *const*) are described by the type of the variable.

What distinguishes a variable is where its value is stored and its scope. The scope of a variable defines where the variable known within the program and is, to some degree, determined by where the variable is declared. In C, variables declared within a function or block have function or block scope. Those declared outside a function have either global or file scope. This allows different variables with the same name to be defined in different files without conflicting. It also allows different functions or compilations to reference the same variable. DWARF documents where the variable is declared in the source file with a (*file*, *line*, *column*) triplet.

```
<1>: DW_TAG_variable
       DW_AT_name = argv
       DW_AT_type = <2>

<2>: DW_TAG_pointer_type
       DW_AT_byte_size = 4
       DW_AT_type = <3>

<3>: DW_TAG_pointer_type
       DW_AT_byte_size = 4
       DW_AT_type = <4>

<4>: DW_TAG_const_type
       DW_AT_type = <5>

<5>: DW_TAG_base_type
       DW_AT_name = char
       DW_AT_byte_size = 1
       DW_AT_encoding = unsigned
```

Figure 6. DWARF description of "`const char **argv`".

DWARF splits variables into three categories: constants, formal parameters, and variables. A constant is used with languages that have true named constants as part of the language, such as Ada parameters. (C

[6] Some compilers define a common set of type definitions at the start of every compilation unit. Others only generate the definitions for the types which are actually referenced in the program. Either is valid.

does not have constants as part of the language. Declaring a variable *const* just says that you cannot modify the variable without using an explicit cast.) A formal parameter represents values passed to a function. We'll come back to that a bit later.

Some languages, like C or C++ (but not Pascal), allow a variable to be declared without defining it. This implies that there should be a real definition of the variable somewhere else, hopefully somewhere that the compiler or debugger can find. A DIE describing a variable declaration provides a description of the variable without actually telling the debugger where it is.

Most variables have a location attribute that describes where the variable is stored. In the simplest of cases, a variable is stored in memory and has a fixed address[7]. But many variables, such as those declared within a C function, are dynamically allocated and locating them requires some (usually simple) computation. For example, a local variable may be allocated on the stack, and locating it may be as simple as adding a fixed offset to a frame pointer. In other cases, the variable may be stored in a register. Other variables may require somewhat more complicated computations to locate the data. A variable that is a member of a C++ class may require more complex computations to determine the location of the base class within a derived class.

```
fig7.c:
        1:  int a;
        2:  void foo()
        3:  {
        4:      register int b;
        5:      int c;
        6:  }

<1>:    DW_TAG_subprogram
          DW_AT_name = foo
<2>:    DW_TAG_variable
          DW_AT_name = b
          DW_AT_type = <4>
          DW_AT_location = (DW_OP_reg0)
<3>:    DW_TAG_variable
          DW_AT_name = c
          DW_AT_type = <4>
          DW_AT_location =
                (DW_OP_fbreg: -12)
<4>:    DW_TAG_base_type
          DW_AT_name = int
          DW_AT_byte_size = 4
          DW_AT_encoding = signed
<5>:    DW_TAG_variable
          DW_AT_name = a
          DW_AT_type = <4>
          DW_AT_external = 1
          DW_AT_location = (DW_OP_addr: 0)
```

Figure 7. DWARF description of variables *a*, *b*, and *c*.

Location Expressions

DWARF provides a very general scheme to describe how to locate the data represented by a variable. A DWARF location expression contains a sequence of operations which tell a debugger how to locate the data. Figure 7 shows DIEs for three variables named *a*, *b*, and *c*. Variable *a* has a fixed location in memory, variable *b* is in register 0, and variable *c* is at offset –12 within the current function's stack frame. Although *a* was declared first, the DIE to describe it is generated later, after all functions. The actual location of *a* will be filled in by the linker.

The DWARF location expression can contain a sequence of operators and values that are evaluated by a simple stack machine. This can be an arbitrarily complex computation, with a wide range of arithmetic operations, tests and branches within the expression, calls to evaluate other location expressions, and accesses to the processor's memory or registers. There are even operations used to describe data which is split up and stored in different locations, such as a structure where some data are stored in memory and some are stored in registers.

Although this great flexibility is seldom used in practice, the location expression should allow the location of a variable's data to be described no matter how complex the language definition or how clever the compiler's optimizations.

Describing Executable Code

Functions and Subprograms

DWARF treats functions that return values and subroutines that do not as variations of the same thing. Drifting slightly away from its roots in C terminology, DWARF describes both with a subprogram DIE. This DIE has a name, a source location triplet, and an attribute which indicates whether the subprogram is *external*, that is, visible outside the current compilation.

A subprogram DIE has attributes that give the low and high memory addresses that the subprogram occupies, if it is contiguous, or a list of memory ranges if the function does not occupy a contiguous set of memory addresses. The low PC address is assumed to be the entry point for the routine unless another one is explicitly specified.

The value that a function returns is given by the *type* attribute. Subroutines that do not return values (like C *void* functions) do not have this attribute. DWARF doesn't describe the calling conventions for a function; that is defined in the Application Binary Interface (ABI) for the particular architecture. There may be attributes that help a debugger to locate the subprogram's data or to find the current subprogram's caller. The *return address* attribute is a location expression that specifies where the address of the caller is stored. The *frame base* attribute is a location expression that computes the address of the stack frame for the function. These are useful since some of the most common optimizations that a compiler might do are to eliminate instructions that explicitly save the return address or frame pointer.

The subprogram DIE owns DIEs that describe the subprogram. The parameters that may be passed to a function are represented by variable DIEs which have the *variable parameter* attribute. If the parameter is optional or has a default value, these are represented by attributes. The DIEs for the parameters are in the same order as the argument list for the function, but there may be additional DIEs interspersed, for example, to define types used by the parameters.

A function may define variables that may be local or global. The DIEs for these variables follow the parameter DIEs. Many languages allow nesting of lexical blocks. These are represented by *lexical block* DIEs which in turn, may own variable DIEs or nested lexical block DIEs.

Here is a somewhat longer example. Figure 8a shows the source for `strndup.c`, a function in *gcc* that duplicates a string. Figure 8b lists the DWARF generated for this file. As in previous examples, the source line information and the location attributes are not shown.

In Figure 8b, DIE <2> shows the definition of `size_t` which is a *typdef* of `unsigned int`. This allows a debugger to

[7] Well, maybe not a fixed address, but one that is a fixed offset from where the executable is loaded. The loader relocates references to addresses within an executable so that at run-time the location attribute contains the actual memory address. In an object file, the location attribute is the offset, along with an appropriate relocation table entry.

display the type of formal argument *n* as a `size_t`, while displaying its value as an unsigned integer. DIE <5> describes the function `strndup`. This has a pointer to its sibling, DIE <10>; all of the following DIEs are children of the Subprogram DIE. The function returns a pointer to *char*, described in DIE <10>. DIE <5> also describes the subroutine as *external* and *prototyped* and gives the low and high PC values for the routine. The formal parameters and local variables of the routine are described in DIEs <6> to <9>.

```
strndup.c:
  1:  #include "ansidecl.h"
  2:  #include <stddef.h>
  3:
  4:  extern size_t strlen (const char*);
  5:  extern PTR malloc (size_t);
  6:  extern PTR memcpy (PTR, const PTR, size_t);
  7:
  8:  char *
  9:  strndup (const char *s, size_t n)
 10:  {
 11:    char *result;
 12:    size_t len = strlen (s);
 13:
 14:    if (n < len)
 15:      len = n;
 16:
 17:    result = (char *) malloc (len + 1);
 18:    if (!result)
 19:      return 0;
 20:
 21:    result[len] = '\0';
 22:    return (char *) memcpy (result, s, len);
 23:  }
```

Figure 8a. Source for `strndup.c`.

```
<1>: DW_TAG_base_type
       DW_AT_name = int
       DW_AT_byte_size = 4
       DW_AT_encoding = signed
<2>: DW_TAG_typedef
       DW_AT_name = size_t
       DW_AT_type = <3>
<3>: DW_TAG_base_type
       DW_AT_name = unsigned int
       DW_AT_byte_size = 4
       DW_AT_encoding = unsigned
<4>: DW_TAG_base_type
       DW_AT_name = long int
       DW_AT_byte_size = 4
       DW_AT_encoding = signed
<5>: DW_TAG_subprogram
       DW_AT_sibling = <10>
       DW_AT_external = 1
       DW_AT_name = strndup
       DW_AT_prototyped = 1
       DW_AT_type = <10>
       DW_AT_low_pc = 0
       DW_AT_high_pc = 0x7b
<6>: DW_TAG_formal_parameter
       DW_AT_name = s
       DW_AT_type = <12>
       DW_AT_location =
          (DW_OP_fbreg: 0)
<7>: DW_TAG_formal_parameter
       DW_AT_name = n
       DW_AT_type = <2>
       DW_AT_location =
          (DW_OP_fbreg: 4)
<8>: DW_TAG_variable
       DW_AT_name = result
       DW_AT_type = <10>
       DW_AT_location =
          (DW_OP_fbreg: -28)
<9>: DW_TAG_variable
       DW_AT_name = len
       DW_AT_type = <2>
       DW_AT_location =
          (DW_OP_fbreg: -24)
<10>: DW_TAG_pointer_type
       DW_AT_byte_size = 4
       DW_AT_type = <11>
<11>: DW_TAG_base_type
       DW_AT_name = char
       DW_AT_byte_size = 1
       DW_AT_encoding =
          signed char
<12>: DW_TAG_pointer_type
       DW_AT_byte_size = 4
       DW_AT_type = <13>
<13>: DW_TAG_const_type
       DW_AT_type = <11>
```

Figure 8b. DWARF description for `strndup.c`.

Compilation Unit

Most interesting programs consists of more than a single file. Each source file that makes up a program is compiled independently and then linked together with system libraries to make up the program. DWARF calls each separately compiled source file a compilation unit.

The DWARF data for each compilation unit starts with a Compilation Unit DIE. This DIE contains general information about the compilation, including the directory and name of the source file, the programming language used, a string which identifies the producer of the DWARF data, and offsets into the DWARF data sections to help locate the line number and macro information.

If the compilation unit is contiguous (i.e., it is loaded into memory in one piece) then there are values for the low and high memory addresses for the unit. This makes it easier for a debugger to identify which compilation unit created the code at a particular memory address. If the compilation unit is not contiguous, then a list of the memory addresses that the code occupies is provided by the compiler and linker.

The Compilation Unit DIE is the parent of all of the DIEs that describe the compilation unit. Generally, the first DIEs will describe data types, followed by global data, then the functions that make up the source file. The DIEs for variables and functions are in the same order in which they appear in the source file.

Data encoding

Conceptually, the DWARF data that describes a program is a tree. Each DIE may have a sibling and maybe several children DIEs. Each of the DIEs has a type (called its TAG) and a number of attributes. Each attributes is represented by a attribute type and a value. Unfortunately, this is not a very dense encoding. Without compression, the DWARF data is unwieldy.

DWARF offers several ways to reduce the size of the data which needs to be saved with the object file. The first is to "flatten" the tree by saving it in prefix order. Each type of DIE is defined to either have children or not. If the DIE cannot have children, the next DIE is its sibling. If the DIE can have children, then the next DIE is its first child. The remaining children are represented as the siblings of this first child. This way, links to the sibling or child DIEs can be eliminated. If the compiler writer thinks that it might be useful to be able to jump from one DIE to its sibling without stepping through each of its children DIEs (for example, to jump to the next function in a compilation) then a *sibling* attribute can be added to the DIE.

A second scheme to compress the data is to use abbreviations. Although DWARF allows great flexibility in which DIEs and attributes it may generate, most compilers only generate a limited set of DIEs, all of which have the same set of attributes. Instead of storing the value of the TAG and the attribute-value pairs, only an index into a table of abbreviations is stored, followed by the attribute codes. Each abbreviation gives the TAG value, a flag indicating whether the DIE has children, and a list of attributes with the type of value it expects. Figure 9 shows the abbreviation for the formal parameter DIE used in Figure 8b. DIE <6> in Figure 8 is actually encoded as shown[8]. This is a significant reduction in the amount of data that needs to be saved at some expense in added complexity.

[8]The encoded entry also includes the file and line values which are not shown in Fig. 8b.

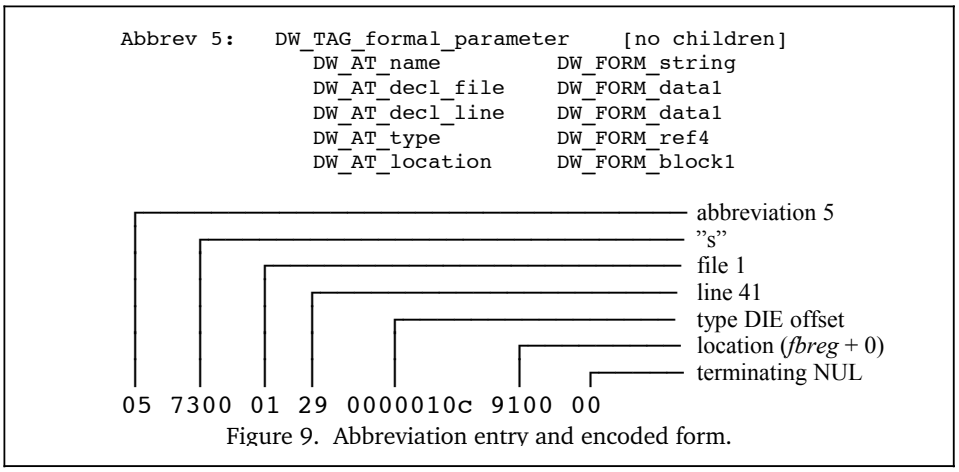

Figure 9. Abbreviation entry and encoded form.

Less commonly used are features of DWARF Version 3 and 4 which allow references from one compilation unit to the DWARF data stored in another compilation unit or in a shared library. Many compilers generate the same abbreviation table and base types for every compilation, independent of whether the compilation actually uses all of the abbreviations or types. These can be saved in a shared library and referenced by each compilation unit, rather than being duplicated in each.

Other DWARF Data

Line Number Table

The DWARF line table contains the mapping between memory addresses that contain the executable code of a program and the source lines that correspond to these addresses. In the simplest form, this can be looked at as a matrix with one column containing the memory addresses and another column containing the source triplet (file, line, and column) for that address. If you want to set a breakpoint at a particular line, the table gives you the memory address to store the breakpoint instruction. Conversely, if your program has a fault (say, using a bad pointer) at some location in memory, you can look for the source line that is closest to the memory address.

DWARF has extended this with added columns to convey additional information about a program. As a compiler optimizes the program, it may move instructions around or remove them. The code for a given source statement may not be stored as a sequence of machine instructions, but may be scattered and interleaved with the instructions for other nearby source statements. It may be useful to identify the end of the code which represents the prolog of a function or the beginning of the epilog, so that the debugger can stop after all of the arguments to a function have been loaded or before the function returns. Some processors can execute more than one instruction set, so there is another column that indicates which set is stored at the specified machine location.

As you might imagine, if this table were stored with one row for each machine instruction, it would be huge. DWARF compresses this data by encoding it as sequence of instructions called a *line number program*[9]. These instructions are interpreted by a simple finite state machine to recreate the complete line number table.

The finite state machine is initialized with a set of default values. Each row in the line number table is generated by executing one or more of the opcodes of the line number program. The opcodes are generally quite simple: for example, add a value to either the machine address or to the line number, set the column number, or set a flag which indicates that the memory address represents the start of an source statement, the end of the function prolog, or the start of the function epilog. A set of special opcodes combine the most common operations (incrementing the memory address and either incrementing or decrementing the source line number) into a single opcode.

Finally, if a row of the line number table has the same source triplet as the previous row, then no instructions are generated for this row in the line number program. Figure 10 lists the line number program for `strndup.c`. Notice that only the machine addresses that represent the beginning instruction of a statement are stored. The compiler did not identify the basic blocks in this code, the end of the prolog or the start of the epilog to the function. This table is encoded in just 31 bytes in the line number program.

Address	File	Line	Col	Stmt	Block	End	Prolog	Epilog	ISA
0x0	0	42	0	yes	no	no	no	no	0
0x9	0	44	0	yes	no	no	no	no	0
0x1a	0	44	0	yes	no	no	no	no	0
0x24	0	46	0	yes	no	no	no	no	0
0x2c	0	47	0	yes	no	no	no	no	0
0x32	0	49	0	yes	no	no	no	no	0
0x41	0	50	0	yes	no	no	no	no	0
0x47	0	51	0	yes	no	no	no	no	0
0x50	0	53	0	yes	no	no	no	no	0
0x59	0	54	0	yes	no	no	no	no	0
0x6a	0	54	0	yes	no	no	no	no	0
0x73	0	55	0	yes	no	no	no	no	0
0x7b	0	56	0	yes	no	yes	no	no	0

File 0: `strndup.c`
File 1: `stddef.h`

Figure 10. Line Number Table for `strndup.c`.

Macro Information

Most debuggers have a very difficult time displaying and debugging code which has macros. The user sees the original source file, with the macros, while the code corresponds to whatever the macros generated.

DWARF includes the description of the macros defined in the program. This is quite rudimentary information, but can be used by a debugger to display the values for a macro or possibly translate the macro into the corresponding source language.

Call Frame Information

Every processor has a certain way of calling functions and passing argu-

[9] Calling this a line number program is something of a misnomer. The program describes much more than just line numbers, such as instruction set, beginning of basic blocks, end of function prolog, etc.

ments, usually defined in the ABI. In the simplest case, this is the same for each function and the debugger knows exactly how to find the argument values and the return address for the function.

For some processors, there may be different calling sequences depending on how the function is written, for example, if there are more than a certain number of arguments. There may be different calling sequences depending on operating systems. Compilers will try to optimize the calling sequence to make code both smaller and faster. One common optimization is having a simple function which doesn't call any others (a leaf function) use its caller stack frame instead of creating its own. Another optimization may be to eliminate a register which points to the current call frame. Some registers may be preserved across the call while others are not. While it may be possible for the debugger to puzzle out all the possible permutations in calling sequence or optimizations, it is both tedious and error-prone. A small change in the optimizations and the debugger may no longer be able to walk the stack to the calling function.

The DWARF Call Frame Information (CFI) provides the debugger with enough information about how a function is called so that it can locate each of the arguments to the function, locate the current call frame, and locate the call frame for the calling function. This information is used by the debugger to "unwind the stack," locating the previous function, the location where the function was called, and the values passed.

Like the line number table, the CFI is encoded as a sequence of instructions that are interpreted to generate a table. There is one row in this table for each address that contains code. The first column contains the machine address while the subsequent columns contain the values of the machine registers when the instruction at that address is executed. Like the line number table, if this table were actually created it would be huge. Luckily, very little changes between two machine instructions, so the CFI encoding is quite compact.

Variable length data

Integer values are used throughout DWARF to represent everything from offsets into data sections to sizes of arrays or structures. In most cases, it isn't possible to place a bound on the size of these values. In a classic data structure each of these values would be represented using the default integer size. Since most values can be represented in only a few bits, this means that the data consists mostly of zeros[10].

DWARF defines a variable length integer, called Little Endian Base 128 (LEB128 or more commonly ULEB for unsigned values and SLEB for signed values), which compresses these integer values. Since the low-order bits contain the data and high-order bits consist of all zeros or ones, LEB values chop off the low-order seven bits of the value. If the remaining bits are all zero or one (sign-extension bits), this is the encoded value. Otherwise, set the high-order bit to one, output this byte, and go on to the next seven low-order bits.

Shrinking DWARF data

The encoding schemes used by DWARF significantly reduce the size of the debugging information compared to an unencoded format like DWARF Version 1. Unfortunately, with many programs the amount of debugging data generated by the compiler can become quite large, frequently much larger than the executable code and data.

DWARF offers ways to further reduce the size of the debugging data. Most strings in the DWARF debugging data are actually references into a separate `.debug_str` section. Duplicate strings can be eliminated when generating this section. Potentially, a linker can merge the `.debug_str` sections from several compilations into a single, smaller string section.

Many programs contain declarations which are duplicated in each compilation unit. For example, debugging data describing many (perhaps thousands) declarations of C++ template functions may be repeated in each compilation. These repeated descriptions can be saved in separate compilation units in uniquely named sections. The linker can use COMDAT (common data) techniques to eliminate the duplicate sections.

Many programs reference a large number of include files which contain many type definitions, resulting in DWARF data which contains thousands of DIEs for these types. A compiler can reduce the size of this data by only generating DWARF for the types which are actually used in the compilation. With DWARF Version 4, type definitions can be saved into a separate `.debug_types` section. The compilation unit contains a DIE which references this separate type unit and a unique 64-bit signature for these types. A linker can recognize compilations which define the same type units and eliminate the duplicates.

[10] An example of this may be seen in the relocation directory in an object file, where file offset and relocation values are represented by integers. Most values are have leading zeros.

ELF sections

While DWARF is defined in a way that allows it to be used with any object file format, it's most often used with ELF. Each of the different kinds of DWARF data are stored in their own section. The names of these sections all start with "`.debug_`". For improved efficiency, most references to DWARF data use an offset from the start of the data for the current compilation. This avoids the need to relocate the debugging data, which speeds up program loading and debugging.

The ELF sections and their contents are

`.debug_abbrev`	Abbreviations used in the `.debug_info` section
`.debug_aranges`	A mapping between memory address and compilation
`.debug_frame`	Call Frame Information
`.debug_info`	The core DWARF data containing DIEs
`.debug_line`	Line Number Program
`.debug_loc`	Location descriptions
`.debug_macinfo`	Macro descriptions
`.debug_pubnames`	A lookup table for global objects and functions
`.debug_pubtypes`	A lookup table for global types
`.debug_ranges`	Address ranges referenced by DIEs
`.debug_str`	String table used by `.debug_info`
`.debug_types`	Type descriptions

Summary

So there you have it – DWARF in a nutshell. Well, not quite a nutshell. The basic concepts for the DWARF debug information are straight-forward. A program is described as a tree with nodes representing the various functions, data and types in the source in a compact language- and machine-independent fashion. The line table provides the mapping between the executable instructions and the source that generated them. The CFI describes how to unwind the stack.

There is quite a bit of subtlety in DWARF as well, given that it needs to express the many different nuances for a wide range of programming languages and different machine architectures. Future directions for DWARF are to improve the description of optimized code so that debuggers can better navigate the code which advanced compiler optimizations generate.

The complete DWARF Version 4 Standard is available for download without charge at the DWARF website (dwarfstd.org). There is also a mailing list for questions and discussion about DWARF. Instructions on registering for the mailing list are also on the website.

Acknowledgements

I want to thank Chris Quenelle of Sun Microsystems and Ron Brender, formerly of HP, for their comments and advice about a previous version of this paper. Thanks also to Susan Heimlich for her many editorial comments.

Generating DWARF with GCC

It's very simple to generate DWARF with *gcc*. Simply specify the *–g* option to generate debugging information. The ELF sections can be displayed using *objump* with the *–h* option.

```
$ gcc –g –c strndup.c
$ objdump –h strndup.o
strndup.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000007b  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  000000b0  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  000000b0  2**2
                  ALLOC
  3 .debug_abbrev 00000073  00000000  00000000  000000b0  2**0
                  CONTENTS, READONLY, DEBUGGING
  4 .debug_info   00000118  00000000  00000000  00000123  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  5 .debug_line   00000080  00000000  00000000  0000023b  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  6 .debug_frame  00000034  00000000  00000000  000002bc  2**2
                  CONTENTS, RELOC, READONLY, DEBUGGING
  7 .debug_loc    0000002c  00000000  00000000  000002f0  2**0
                  CONTENTS, READONLY, DEBUGGING
  8 .debug_pubnames 0000001e  00000000  00000000  0000031c  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  9 .debug_aranges 00000020  00000000  00000000  0000033a  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
 10 .comment      0000002a  00000000  00000000  0000035a  2**0
                  CONTENTS, READONLY
 11 .note.GNU-stack 00000000  00000000  00000000  00000384  2**0
                  CONTENTS, READONLY
```

Printing DWARF using Readelf

Readelf can display and decode the DWARF data in an object or executable file. The options are

```
-w             – display all DWARF sections
-w[liaprmfFso] – display specific sections
       l       – line table
       i       – debug info
       a       – abbreviation table
       p       – public names
       r       – ranges
       m       – macro table
       f       – debug frame (encoded)
       F       – debug frame (decoded)
       s       – string table
       o       – location lists
```

The DWARF listing for all but the smallest programs is quite voluminous, so it would be a good idea to direct *readelf*'s output to a file and then browse the file with *less* or an editor such as *vi*.


DWARF DEBUGGING FORMAT