Debugging formats DWARF and STAB

Adarsh Thampan
Development Manager
IBM

Suchitra Venugopal
Advisory System Analyst
IBM

25 July 2011

Learn more about STAB and DWARF, two popular debugging formats. Find out how to debug and dissect UNIX executables constituting the DWARF and STAB formats. This material is of particular interest to programmers working on compilers and debuggers and anyone interested in reading or writing DWARF and STAB information.

Overview

The debugging information stored within a binary helps a programmer or debugger locate problem areas in a program, and find the stack trace when an error happens in an application. This information is also used by popular debugger tools, such as gdb and wdb. A debugging data format is a means of storing information about a compiled program for use by high-level debuggers. Modern debugging data formats store enough information to allow source-level debugging.

The debugger and debug information is available or present in all software components, however, not all users are aware of what the debug data consists of and how the data is stored.

If you work with debugging formats, are a C/C++ programmer with relevant experience on any UNIX platform and have basic knowledge of compilers and debuggers, or are in the process of choosing formats then this article is for you.

We cover extracting debugging information from two popular debugging formats, STAB and DWARF. We discuss the advantages of one format over the other. Additionally, we provide an example application to extract information from a binary in a DWARF format to assist in experimentation and understanding.

The DWARF and STAB formats are the most widely used executable and linking format (ELF).

Programmers can make use of this document in following situations:

- To perform the debug of an application without a debugger by extracting the debug information separately and storing it in separate files. Binaries are stripped of all debugging information when a product goes to production. If the necessary debug information can be captured before stripping, and is placed in separate files, these files can later be used to generate detailed trace information when any exception or crash occurs in the application.
- To identify a debugging format to fit a particular platform. UNIX platforms like Solaris use STAB, while HP Itanium uses DWARF.
- To ascertain if a particular binary corresponds to the correct source code version. For example, if a new function is added to a source code and binary is created, verification that the source file and the binaries are in sync may be required. This can be done easily if one is able to check the debug information in the binaries. When a source code is modified and the binaries rebuilt, the internal layout of the binary changes accordingly to accommodate the recent changes. Hence, simply checking the debug information can ensure that the binaries are in sync with the source. The sample application included provides important debug information to the user for any binary.

Debugging formats

A debugging data format is a means of storing information about a compiled computer program for use by high-level debuggers. Modern debugging data formats store enough information to allow source-level debugging.

There are many debugging formats available. STAB, COFF, PECOFF, OMF, IEEE695, and three versions of DWARF, are a few common choices. Let's compare STAB and DWARF, and discuss extracting debug information from both formats.

STAB

The traditional format for debugging information is called STAB (symbol table). STAB information is saved in the `.stab` and `.stabstr`. sections of an ELF file.

The STAB debug format is a poorly documented, semi-standard format for debugging information in COFF and ELF object files. The debugging information is stored as part of the object file's symbol table and thus is limited in complexity and scope. Despite this, STAB is a common debugging format on older UNIX and compatible systems.

For some object file formats, the debugging information is encapsulated in assembler directives known collectively as STAB directives, which are interspersed with the generated code. STABs are the native format for debugging information in the a.out and XCOFF object file formats. The GNU tools can also emit STABs in the COFF and ECOFF object file formats.

The assembler creates two custom sections:

- .stab, which contains an array of fixed length structures, one structure per stab, and
- .stabstr, that contains all the variable length strings that are referenced by stabs in the .stab section.

The byte order of the STABs binary data depends on the object file format.

Structure of a program

The elements of the program structure that STABs encode include the name of the main function, the names of the source and include files, the line numbers, procedure names and types, and the beginnings and ends of blocks of code.

Most languages allow the main program to have any name. The `N_MAIN` STAB type tells the debugger the name that is used in this program. Only the string field is significant; it is the name of a function that is the main program. Most C compilers do not use this STAB (they expect the debugger to assume that the name is main), but some C compilers emit an `N_MAIN` STAB for the main function.

Before any other STABs occur, there must be a STAB specifying the source file. This information is contained in a symbol of STAB type `N_SO`. The string field contains the name of the file. The value of the symbol is the start address of the portion of the text section corresponding to that file.

An `N_SLINE` symbol represents the start of a source line. The `desc` field contains the line number and the `value` contains the code address for the start of that source line. On most machines the address is absolute; for STABs in sections it is relative to the function in which the `N_SLINE` symbol occurs.

All of the following STABs normally use the `N_FUN` symbol type for functions.

Other common sections include:

- `N_SLINE, N_XLINE`— line numbers
- `N_LBRAC`— left bracket, start of a function
- `N_RBRAC`— right bracket, end of a function
- `N_SOL`— all the included files for this binary

DWARF

DWARF (debug with arbitrary record format) is a more recent format for ELF files. It was created to overcome shortcomings in STAB, allowing for more detailed and compact descriptions of data structures, data variable movement, and complex language structures, such as in C. The debugging information is stored in sections in the object file. 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 ownership relation of debugging information entries is achieved naturally because the debugging information is represented as a tree. The nodes of the tree are the debugging information entries themselves. The child entries of any node are exactly those debugging information entries owned by that node. The tree itself is represented by flattening it in prefix order. Each debugging information entry is defined either to have child entries or not to have child entries. If an entry is defined not to have children, the next physically succeeding entry is the sibling of the prior entry. If an entry is defined to have children, the next physically succeeding

entry is the first child of the prior entry. Additional children of the parent entry are represented as siblings of the first child. A chain of sibling entries is terminated by a null entry.

Debugging information entry

The basic descriptive entity in DWARF is the debugging information entry (DIE). A DIE has a tag that specifies what the DIE describes and a list of attributes that fills in details, and further describes the entity. A DIE (except for the top most) 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).

Compilation unit

A compilation unit (CU) is a type of DIE. 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 CU DIE. This DIE contains general information about the compilation, including: the directory and name of the source file, the programming language used, and a string that 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 CU is contiguous (that is, 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. The CU DIE is the parent of all of the DIEs that describe the compilation unit.

A CU typically represents the text and data contributed to an executable by a single relocatable object file. It may be derived from several source files, including pre-processed "include files.'"

The CU entry may have the following attributes:

- A `DW_AT_low_pc` attribute whose value is the relocated address of the first machine instruction generated for that compilation unit.
- A `DW_AT_high_pc` attribute whose value is the relocated address of the first location past the last machine instruction generated for that compilation unit.

Other types of DIE include:

- `DW_TAG_subprogram`— A global or file static subroutine or function
- `DW_TAG_inlined_subroutine`— A particular inlined instance of a subroutine or function
- `DW_TAG_entry_point`— A Fortran entry point
- `DW_TAG_variable`— A variable
- `DW_TAG_pointer_type`— A pointer variable
- `DW_TAG_formal_parameter`— Each argument within the function parameter pack is represented by a debugging information entry with this tag

The subprogram DIE owns DIEs describing the subprogram. The parameters that may be passed to a function are represented by variable DIEs that have the variable parameter attribute.

Line number table

The DWARF line table contains the mapping between the source lines (for the executable parts of a program) and the memory containing the code corresponding to the source. 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). 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 (for example, using a bad pointer) at some location in memory, you can look for the source line that is closest to the memory address.

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.

Getting information from STAB and DWARF formats

STAB and DWARF are two different ways of representing debug information for binaries. The following section describes how the debug information can be captured from STAB and DWARF sections.

The approach followed is to scan each binary file for debug information and determine which file names contribute to that binary. For each file, all function names and line numbers (first line number and last line number) for each function is captured. A linked list can be used to store this information.

Information from STAB format

The binary file is scanned through and the stab sections in the file are checked. When `N_SOL` is found, it represents the name of the file that is included in this binary and from this symbol the file name can be stored. After the file name is obtained, the next step should be to get all the functions that are present in this file. `N_FUN` represents a function in the stab section. If we get this symbol after an `N_SOL` symbol (file name), it means a function is encountered for the above file name. From `N_FUN`, the function name and details can be stored.

After this, when an `N_LBRAC` symbol is returned, it represents a left brace and the start of the function. Now we can conclude that the above function starts here. Following this symbol would be `N_SLINE` or `N_XLINE`, which denote lines in the function. The first and last line numbers, or if required, all line numbers can be stored from this symbol.

After a few entries for line numbers (`N_SLINE` or `N_XLINE`) the `N_RBRAC` symbol would be captured. When this symbol is encountered, it denotes the end of the function. At this point, one set of information is captured, that is, for a binary, one of the included file names and one function and the details of that function. Similarly, we can obtain all the functions in that included file and the details of all the included files for that binary. The above steps need to be carried out for retrieving the full information about a binary file.

The symbol table stores the details of all the symbols (functions) in the binary. Listing 1 is a diagrammatic representation of gathering information from STAB format for a simple function.

Listing 1. Simple program showing STAB types

```
abcd.c    ----> Symbol is N_SOL
int function_name(int a, int b)   ------> Symbol is N_FUN
{     ----> Symbol is N_LBRAC
    int i;       ----->Symbol is N_SLINE
    i = a+b;
    return i;
} -----> Symbol is N_RBRAC
```

A flow chart representing the algorithm for getting the information from the STAB format follows in Figures 1 and 2.

Start by getting the next stab section, then:

- After you reach the end of the section, consolidate all the information gathered in various stab sections and store them in a data structure. If you do not reach the end of the stab section:
    1. Check to see if this section = N_SOL, if it does then extract the source file name and go back to Start.
    2. If if does not = N_SOL, does it = N_UNDF/N_ENDM? If it does then reset the current function. Go back to start.
    3. If it does not = N_UNDF/N_ENDM, check if it = N_FUN, if yes, then find the subprogram name. If required, store the stab number. Go back to Start.
    4. If the section does not = N_FUN, check if it = N_LBRAC, if it does it denotes the start of the subprogram. (A flag can be set for this if the N_FUN section has already been captured.) Go back to Start.
    5. If the section does not = N_LBRAC, does it = N_SLINE/n_XLINE; these are lines after the N_LBRAC. If it does, the first and last line, and the number of lines can be stored for the subprogram. Go back to Start.
    6. If the section does not = N_SLINE/n_XLINE, does it = N_RBRAC? If it does, it represents the end of a function. Go back to Start. If the section does not = N_RBRAC ignore this stab section and go back to Start.

Figure 1. Flow chart to extract debug information from STAB

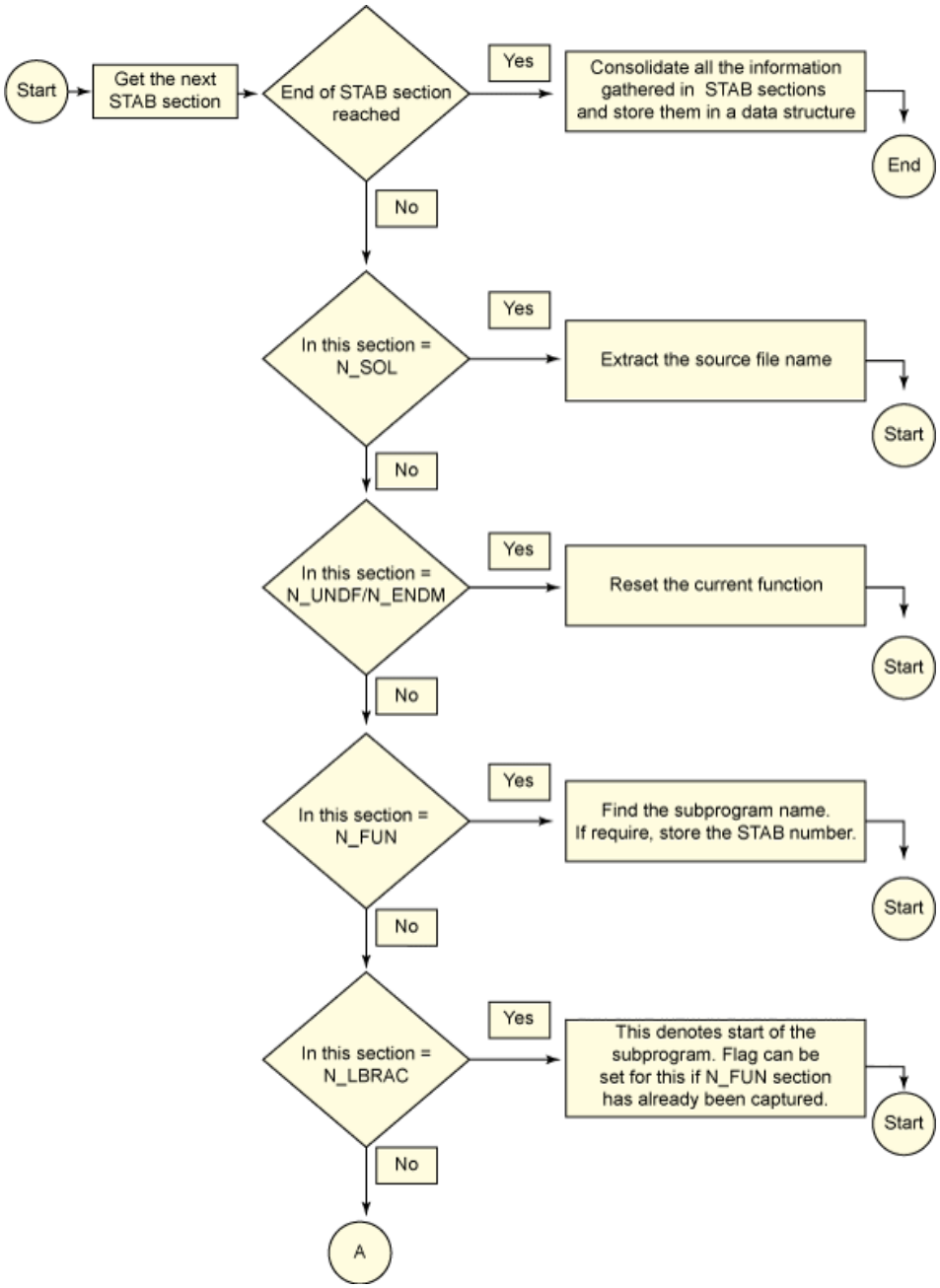
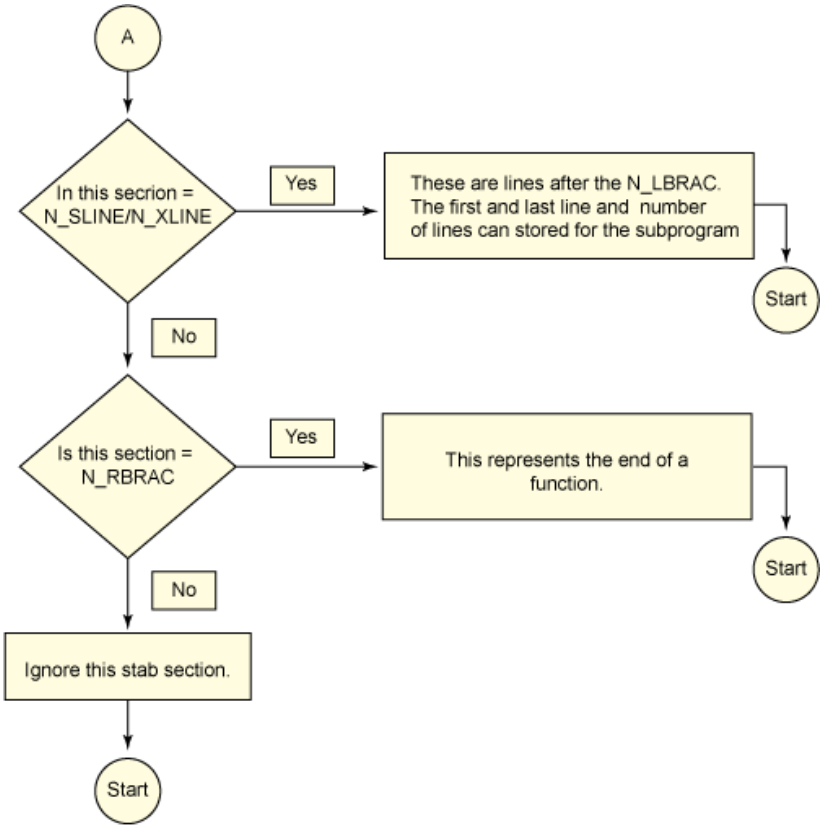


Figure 2. Continuation of flow chart to extract debug information from STAB



Information from DWARF format

In DWARF format all information is in a tree format, as shown in Listing 2.

Listing 2. DWARF Tree Format

```
COMPILE_UNIT<header overall offset = 0>:
<0><   11> dw_tag_compile_unit  0
  DW_AT_stmt_list   0
  DW_AT_HP_actuals_stmt_list 0
  DW_AT_macro_info  0
  DW_AT_name   /usres/mainsoft-v52-orig/mw/init.C
  DW_AT_producer
  DW_AT_lanugage   DW_LANG_C_plus_plus
  DW_AT_HP_proc_per_section yes(1)
  DW_AT_comp_dir   /home/yadvecha/ISLHPIA02/RTWIN/730/Source

LOCAL_SYMBOLS:
<1><  109> DW_TAG_module
  DW_AT_name   /users/mainsoft-v52-orig/mw/init.C
  DW_AT_sibling   <365>
<2><  149> DW_TAG_class_type
  DW_AT_name   setInFunc
  DW_AT_declaration  yes(1)
  DW_AT_sibling   <333>
<3><  166> DW_TAG_subprogram
  DW_AT_name   ~setInFunc
  DW_AT_HP_linkage_name  _ZN9setInFuncD1Ev
  DW_AT_declaration  no
```

```
  DW_AT_external   yes (1)
  DW_AT_accessibility  DW_ACCESS_public
  DW_AT_sibling   <229>
<4><  203> DW_TAG_formal_parameter
  DW_AT_name   this
  DW_AT_artificial  yes (1)
  DW_AT_location   DW_OP_regx 34
  DW_AT_type   <365>
```

The symbol table section in DWARF is the same as in STAB. Extracting the debug information from DWARF works as follows:

Each DIE in the binary is parsed and when a CU DIE is found, the name of the CU is stored. This represents the name of the files that are included in this binary. The name of the CU is also stored to find the line number table for locating the line numbers for all the functions in this CU. When DW_TAG_subprogram DIE is captured after a CU DIE, this denotes that this subprogram belongs to the file that is captured under the CU DIE. When DW_TAG_subprogram is encountered, the details of the function are stored. From the attributes of the subprogram, get the low address and high address corresponding to this subprogram. This represents the addresses corresponding to the start and end of that subprogram. For calculating the line numbers corresponding to these addresses, get the line buffer from the debug_line section (line buffer) for the CU DIE. For each line in the line buffer for a CU, check for the low address match to find the first line number for the function. The last line number for the function is obtained from the debug_line section where the line number address is less than or equal to the high address of the function.

With this captured information, we can create a list of source file names, function names and the line numbers for the functions. The debugger uses these basic pieces of information for debugging purposes.

Listing 3 shows the DIE for a subprogram and how the line number is obtained from the line table.

Listing 3. DWARF line number mapping

```
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
Address      File      Line     Col

0x0         0         42        0
0x9            0           44        0
0x1a           0           44        0
0x24           0           46        0
0x2c           0           47        0
0x32           0           49        0
0x41           0           50        0
0x47           0           51        0
0x50           0           53        0
0x59           0           54        0
0x6a           0           54        0
0x73           0           55        0
0x7b           0           56        0
File 0: strndup.c
```

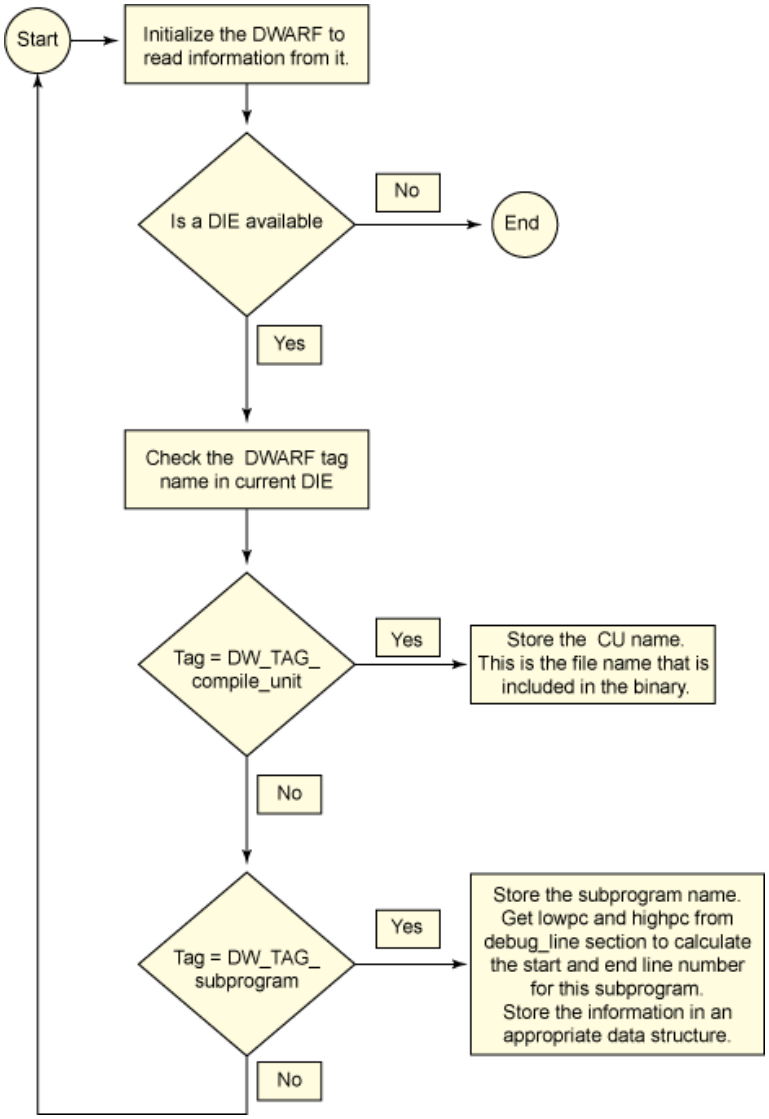
The start and end line numbers for this subprogram are 42 and 56, which correspond to addresses `0x0(DW_AT_low_pc)` and `0x7b(DW_AT_high_pc)`.

Figure 3 shows a flow chart representing the algorithm for getting the information from DWARF:

Start by initializing the DWARF to read information from it. If a DIE is not available then end. If a DIE is available:

- Check the DWARF tag name in the current DIE. If Tag = DW_TAG_compile_unit, then store the compilation unit name. This is the file name which is included in the binary.
- If it does not have Tag = DW_TAG_compile_unit, then store the subprogram name. Get lowpc and highpc from the debug_line section of this subprogram to calculate the its start and end line numbers. Store the info in an appropriate data structure.
- If you cannot store the subprogram name then start over by checking if DIE is available.

Figure 3. Flow chart to extract debug information from DWARF



Which one should I use?

DWARF is a block-structured and extensible description of a program's source, and how it is translated into executable code. It is easy to add new descriptions or extend descriptions in DWARF. Additionally, DWARF is a more advanced format and has facilities for describing a more complex execution environment, such as discontiguous scopes, stack structures, and stack unwinding, which STAB cannot. STAB, on the other hand, is more traditional, and limited in it's expressive abilities. It depends on predefined symbol and type definitions and is not easily modified or extended.

Some of the major advantages of using DWARF are:

- It is easier for platform-independent tools to read DWARF instead of parsing the more esoteric stabs.
- It is actively being developed.
- It is more easily extendible. This allows easy implementation of advanced features and optimized code debugging. Older tools could possibly ignore new data.

New applications using debug information are better served by DWARF. As mentioned, it provides more information and is easily extendible. If you are working on an application where the debug information is already stored in STAB, however, it is better to continue with STAB unless you have sufficient time to change the design and fully switch over to DWARF. As a general rule, DWARF is more desirable.

Going forward

The download example has an application (dwarfexample) that reads the important debug information in a DWARF format from a binary. A binary file (stacktrace) along with the source code (stack_tracing.c) is also provided so that the user can run the "dwarfexample" application using the binary and match the output of the application with the actual data in the source code (as in stack_tracing.c). Debug information displayed using this application includes the source files, which are included in the binary, and the functions details (function names and the line numbers) in each of the source files. You can edit the source code of the sample program (stack_tracing.c) to either add or delete functions or change the line numbers and observe that the output of the sample application when run using the new binary reflects the changes done in the sample program.

In the sample application (see Download) the output as of now displays files, functions, and line numbers present in the binary. You can extend this application to get more information such as inline subprograms, and call frame information, from the binary.

Downloads

Description	Name	Size
Program to extract debug info from binaries	DwarfExample.zip	10KB

Resources

- Participate in the discussion forum for this content.
- Dwarf debugging standard: Check out the official website for DWARF standard, and stay current with DWARF format 3.
- Introduction to the DWARF debugging format: Find more about this and other debugging formats and the information stored in them.
- Libdwarf and Dwarfdump: Access these libraries for reading the DWARF data.
- ELF and DWARF for Linux: Find more information related to ELF and DWARF for Linux for S/390 and zSeries

About the authors

Adarsh Thampan

Adarsh Thampan has ten years industry experience that includes porting IBM WebSphere Federation Server and Optim on HP Itanium platform, and IBM products to Linux on System Z and z/OS. He is a co-author of the white paper, Mixed Platform Stack Project: Linux on System z and IBM z/OS.

Suchitra Venugopal

Suchitra Venugopal has over 9 years experience in the software industry in various roles. Her primarily focus is on open system technologies. She has worked on porting IBM Optim to HP Itanium platform, and testing automation for ISW on zLinux. Currently, she is involved with porting of SPSS to zOS platform.