A Producer Library Interface to DWARF

David Anderson

1. INTRODUCTION

This document describes an interface to `libdwarf`, a library of functions to provide creation of DWARF debugging information records, DWARF line number information, DWARF address range and pubnames information, weak names information, and DWARF frame description information.

1.1 Copyright

1.2 Purpose and Scope

The purpose of this document is to propose a library of functions to create DWARF debugging information. Reading (consuming) of such records is discussed in a separate document.

The functions in this document have mostly been implemented at Silicon Graphics and are being used by the code generator to provide debugging information. Some functions (and support for some extensions) were provided by Sun Microsystems.

Example code showing one use of the functionality may be found in the dwarfgen `dwarfgen` application (provided in the source distribution along with libdwarf).

The focus of this document is the functional interface, and as such, implementation and optimization issues are intentionally ignored.

Error handling, error codes, and certain `Libdwarf` codes are discussed in the "*A Consumer Library Interface to DWARF*", which should be read (or at least skimmed) before reading this document.

However the general style of functions here in the producer library is rather C-traditional with various types as return values (quite different from the consumer library interfaces). The style generally follows the style of the original DWARF1 reader proposed as an interface to DWARF. When the style of the reader interfaces was changed (1994) in the dwarf reader (See the "Document History" section of "A Consumer Library Interface to DWARF") the interfaces here were not changed as it seemed like too much of a change for the two applications then using the interface! So this interface remains in the traditional C style of returning various data types with various (somewhat inconsistent) means of indicating failure.

The error handling code in the library may either return a value or abort. The library user can provide a function that the producer code will call on errors (which would allow callers avoid testing for error returns

if the user function exits or aborts). See the `dwarf_producer_init_c()` description below for more details (possibly the older forms `dwarf_producer_init_b()` and `dwarf_producer_init()` may be of interest).

1.3 Document History

This document originally prominently referenced "UNIX International Programming Languages Special Interest Group " (PLSIG). Both UNIX International and the affiliated Programming Languages Special Interest Group are defunct (UNIX is a registered trademark of UNIX System Laboratories, Inc. in the United States and other countries). Nothing except the general interface style is actually related to anything shown to the PLSIG (this document was open sourced with libdwarf in the mid 1990's).

See "http://www.dwarfstd.org" for information on current DWARF standards and committee activities.

1.4 Definitions

DWARF debugging information entries (DIEs) are the segments of information placed in the `.debug_info` and related sections by compilers, assemblers, and linkage editors that, in conjunction with line number entries, are necessary for symbolic source-level debugging. Refer to the document "*DWARF Debugging Information Format*" from UI PLSIG for a more complete description of these entries.

This document adopts all the terms and definitions in "*DWARF Debugging Information Format*" version 2. and the "*A Consumer Library Interface to DWARF*".

In addition, this document refers to Elf, the ATT/USL System V Release 4 object format. This is because the library was first developed for that object format. Hopefully the functions defined here can easily be applied to other object formats.

1.5 Overview

The remaining sections of this document describe a proposed producer (compiler or assembler) interface to *Libdwarf*, first by describing the purpose of additional types defined by the interface, followed by descriptions of the available operations. This document assumes you are thoroughly familiar with the information contained in the *DWARF Debugging Information Format* document, and "*A Consumer Library Interface to DWARF*".

The interface necessarily knows a little bit about the object format (which is assumed to be Elf). We make an attempt to make this knowledge as limited as possible. For example, *Libdwarf* does not do the writing of object data to the disk. The producer program does that.

1.6 Revision History

March 1993	Work on dwarf2 sgi producer draft begins
March 1999	Adding a function to allow any number of trips through the dwarf_get_section_bytes() call.
April 10 1999	Added support for assembler text output of dwarf (as when the output must pass through an assembler). Revamped internals for better performance and simpler provision for differences in ABI.

Sep 1, 1999 Added support for little- and cross- endian debug info creation.

May 7 2007 This library interface now cleans up, deallocating all memory it uses (the application simply calls dwarf_producer_finish(dbg)).

September 20 2010 Now documents the marker feature of DIE creation.

2. Type Definitions

2.1 General Description

The *libdwarf.h* header file contains typedefs and preprocessor definitions of types and symbolic names used to reference objects of *Libdwarf* . The types defined by typedefs contained in *libdwarf.h* all use the convention of adding *Dwarf_* as a prefix to indicate that they refer to objects used by Libdwarf. The prefix *Dwarf_P_* is used for objects referenced by the *Libdwarf* Producer when there are similar but distinct objects used by the Consumer.

2.2 Namespace issues

Application programs should avoid creating names beginning with `Dwarf_` `dwarf_` or `DW_` as these are reserved to dwarf and libdwarf.

3. libdwarf and Elf and relocations

Much of the description below presumes that Elf is the object format in use. The library is probably usable with other object formats that allow arbitrary sections to be created.

3.1 binary or assembler output

With `DW_DLC_STREAM_RELOCATIONS` (see below) it is assumed that the calling app will simply write the streams and relocations directly into an Elf file, without going through an assembler.

With `DW_DLC_SYMBOLIC_RELOCATIONS` the calling app must either A) generate binary relocation streams and write the generated debug information streams and the relocation streams direct to an elf file or B) generate assembler output text for an assembler to read and produce an object file.

With case B) the libdwarf-calling application must use the relocation information to change points of each binary stream into references to symbolic names. It is necessary for the assembler to be willing to accept and generate relocations for references from arbitrary byte boundaries. For example:

```
.data 0a0bcc    #producing 3 bytes of data.
.word mylabel   #producing a reference
.word endlabel - startlable #producing absolute length
```

3.2 libdwarf relationship to Elf

When the documentation below refers to 'an elf section number' it is really only dependent on getting (via the callback function passed by the caller of `dwarf_producer_init_c()` and the older forms, `dwarf_producer_init_b()` or `dwarf_producer_init()`) a sequence of integers back (with 1 as the lowest).

When the documentation below refers to 'an Elf symbol index' it is really dependent on Elf symbol numbers only if `DW_DLC_STREAM_RELOCATIONS` are being generated (see below). With `DW_DLC_STREAM_RELOCATIONS` the library is generating Elf relocations and the section numbers in binary form so the section numbers and symbol indices must really be Elf (or elf-like) numbers.

With `DW_DLC_SYMBOLIC_RELOCATIONS` the values passed as symbol indexes can be any integer set or even pointer set. All that libdwarf assumes is that where values are unique they get unique values. Libdwarf does not generate any kind of symbol table from the numbers and does not check their uniqueness or lack thereof.

3.3 libdwarf and relocations

With `DW_DLC_SYMBOLIC_RELOCATIONS` libdwarf creates binary streams of debug information and arrays of relocation information describing the necessary relocation. The Elf section numbers and symbol numbers appear nowhere in the binary streams. Such appear only in the relocation information and the passed-back information from calls requesting the relocation information. As a consequence, the 'symbol indices' can be any pointer or integer value as the caller must arrange that the output deal with relocations.

With `DW_DLC_STREAM_RELOCATIONS` all the relocations are directly created by libdwarf as binary streams (libdwarf only creates the streams in memory, it does not write them to disk).

3.4 symbols, addresses, and offsets

The following applies to calls that pass in symbol indices, addresses, and offsets, such as `dwarf_add_AT_targ_address()` `dwarf_add_arange_b()` and `dwarf_add_frame_fde_b()`.

With `DW_DLC_STREAM_RELOCATIONS` a passed in address is one of: a) a section offset and the (non-global) symbol index of a section symbol. b) A symbol index (global symbol) and a zero offset.

With `DW_DLC_SYMBOLIC_RELOCATIONS` the same approach can be used, or, instead, a passed in address may be c) a symbol handle and an offset. In this case, since it is up to the calling app to generate binary relocations (if appropriate) or to turn the binary stream into a text stream (for input to an assembler, if appropriate) the application has complete control of the interpretation of the symbol handles.

4. Memory Management

Several of the functions that comprise the *Libdwarf* producer interface dynamically allocate values and some return pointers to those spaces. The dynamically allocated spaces can not be reclaimed (and must not be freed) except by `dwarf_producer_finish(dbg)`.

All data for a particular `Dwarf_P_Debug` descriptor is separate from the data for any other `Dwarf_P_Debug` descriptor in use in the library-calling application.

4.1 Read-only Properties

All pointers returned by or as a result of a *Libdwarf* call should be assumed to point to read-only memory. Except as defined by this document, the results are undefined for *Libdwarf* clients that attempt to write to a region pointed to by a return value from a *Libdwarf* call.

4.2 Storage Deallocation

Calling `dwarf_producer_finish(dbg)` frees all the space, and invalidates all pointers returned from `Libdwarf` functions on or descended from `dbg`).

5. Functional Interface

This section describes the functions available in the *Libdwarf* library. Each function description includes its definition, followed by a paragraph describing the function's operation.

The functions may be categorized into groups: *initialization and termination operations*, *debugging information entry creation*, *Elf section callback function*, *attribute creation*, *expression creation*, *line number creation*, *fast-access (aranges) creation*, *fast-access (pubnames) creation*, *fast-access (weak names) creation*, *macro information creation*, *low level (.debug_frame) creation*, and *location list (.debug_loc) creation*.

The following sections describe these functions.

5.1 Initialization and Termination Operations

These functions setup `Libdwarf` to accumulate debugging information for an object, usually a compilation-unit, provided by the producer. The actual addition of information is done by functions in the other sections of this document. Once all the information has been added, functions from this section are used to transform the information to appropriate byte streams, and help to write out the byte streams to disk.

Typically then, a producer application would create a `Dwarf_P_Debug` descriptor to gather debugging information for a particular compilation-unit using `dwarf_producer_init_c()`. (Older code may use `dwarf_producer_init_b()` or `dwarf_producer_init()`). The producer application would use this `Dwarf_P_Debug` descriptor to accumulate debugging information for this object using functions from other sections of this document. Once all the information had been added, it would call `dwarf_transform_to_disk_form()` to convert the accumulated information into byte streams in accordance with the `DWARF` standard. The application would then repeatedly call `dwarf_get_section_bytes()` for each of the `.debug_*` created. This gives the producer information about the data bytes to be written to disk. At this point, the producer would release all resource used by `Libdwarf` for this object by calling `dwarf_producer_finish()`.

It is also possible to create assembler-input character streams from the byte streams created by this library. This feature requires slightly different interfaces than direct binary output. The details are mentioned in the text.

5.1.1 dwarf_producer_init()

```
Dwarf_P_Debug dwarf_producer_init(
        Dwarf_Unsigned flags,
        Dwarf_Callback_Func func,
        Dwarf_Handler errhand,
        Dwarf_Ptr errarg,
        Dwarf_Error *error)
```

This is the oldest form and code should migrate to the newest form, `dwarf_producer_init_c()`.

The function `dwarf_producer_init()` returns a new `Dwarf_P_Debug` descriptor that can be used to add `Dwarf` information to the object. On error it returns `DW_DLV_BADADDR`. `flags` determine whether the target object is 64-bit or 32-bit. `func` is a pointer to a function called-back from `Libdwarf` whenever `Libdwarf` needs to create a new object section (as it will for each .debug_* section and related relocation section).

`errhand` is a pointer to a function that will be used as a default fall-back function for handling errors detected by `Libdwarf`.

`errarg` is the default error argument used by the function pointed to by `errhand`.

For historical reasons the error handling is complicated and the following three paragraphs describe the three possible scenarios when a producer function detects an error. In all cases a short error message is printed on stdout if the error number is negative (as all such should be, see libdwarf.h). Then further action is taken as follows.

First, if the Dwarf_Error argument to any specific producer function (see the functions documented below) is non-null the `errhand` argument here is ignored in that call and the specific producer function sets the Dwarf_Error and returns some specific value (for dwarf_producer_init it is DW_DLV_BADADDR as mentioned just above) indicating there is an error.

Second, if the Dwarf_Error argument to any specific producer function (see the functions documented below) is NULL and the `errarg` to `dwarf_producer_init()` is non-NULL then on an error in the producer code the Dwarf_Handler function is called and if that called function returns the producer code returns a specific value (for dwarf_producer_init it is DW_DLV_BADADDR as mentioned just above) indicating there is an error.

Third, if the Dwarf_Error argument to any specific producer function (see the functions documented below) is NULL and the `errarg` to `dwarf_producer_init()` is NULL then on an error `abort()` is called.

The `flags` values are as follows:

> `DW_DLC_WRITE` is required. The values `DW_DLC_READ` `DW_DLC_RDWR` are not supported by the producer and must not be passed.
>
> If `DW_DLC_SIZE_64` is not ORed into `flags` then `DW_DLC_SIZE_32` is assumed. Oring in both is an error.
>
> If `DW_DLC_OFFSET_SIZE_64` is not ORed into `flags` then 64 bit offsets (as defined in the 1999 DWARF3) may be used (see next paragraph) to generate DWARF (if and only if DW_DLC_SIZE_64 is also ORed into `flags`).
>
> If `HAVE_STRICT_32BIT_OFFSET` is set at configure time only 32bit DWARF offsets are generated (use configure option --enable-dwarf-format-strict-32bit) and `DW_DLC_OFFSET_SIZE_64` is ignored. If `HAVE_SGI_IRIX_OFFSETS` is set at configure time SGI IRIX offsets (standard 32bit, a special 64bit offset for 64bit address objects) are

generated (use configure option --enable-dwarf-format-sgi-irix) and DW_DLC_OFFSET_SIZE_64 is ignored. If neither HAVE_STRICT_32BIT_OFFSET nor HAVE_SGI_IRIX_OFFSETS is set at configure time then standard offset sizes are used (and HAVE_DWARF2_99_EXTENSION is set) and DW_DLC_OFFSET_SIZE_64 is honored.

If DW_DLC_ISA_IA64 is not ORed into flags then DW_DLC_ISA_MIPS is assumed. Oring in both is an error.

If DW_DLC_TARGET_BIGENDIAN is not ORed into flags then endianness the same as the host is assumed.

If DW_DLC_TARGET_LITTLEENDIAN is not ORed into flags then endianness the same as the host is assumed.

If both DW_DLC_TARGET_LITTLEENDIAN and DW_DLC_TARGET_BIGENDIAN are or-d in it is an error.

Either one of two output forms is specifiable: DW_DLC_STREAM_RELOCATIONS or DW_DLC_SYMBOLIC_RELOCATIONS .

The default is DW_DLC_STREAM_RELOCATIONS . The DW_DLC_STREAM_RELOCATIONS are relocations in a binary stream (as used in a MIPS Elf object).

The DW_DLC_SYMBOLIC_RELOCATIONS are the same relocations but expressed in an array of structures defined by libdwarf, which the caller of the relevant function (see below) must deal with appropriately. This method of expressing relocations allows the producer-application to easily produce assembler text output of debugging information.

If DW_DLC_SYMBOLIC_RELOCATIONS is ORed into flags then relocations are returned not as streams but through an array of structures.

The function func must be provided by the user of this library. Its prototype is:

```
typedef int (*Dwarf_Callback_Func)(
    char* name,
    int                 size,
    Dwarf_Unsigned      type,
    Dwarf_Unsigned      flags,
    Dwarf_Unsigned      link,
    Dwarf_Unsigned      info,
    int*                sect_name_index,
    int*                error)
```

For each section in the object file that libdwarf needs to create, it calls this function once (calling it from dwarf_transform_to_disk_form()), passing in the section name, the section type, the section flags, the link field, and the info field. For an Elf object file these values should be appropriate Elf section header values. For example, for relocation callbacks, the link field is supposed to be set (by the app) to the index of the symtab section (the link field passed through the callback must be ignored by the app). And, for relocation callbacks, the info field is passed as the elf section number of the section the relocations apply to.

On success the user function should return the Elf section number of the newly created Elf section.

On success, the function should also set the integer pointed to by `sect_name_index` to the Elf symbol number assigned in the Elf symbol table of the new Elf section. This symbol number is needed with relocations dependent on the relocation of this new section. Because "int *" is not guaranteed to work with elf 'symbols' that are really pointers, It is better to use the `dwarf_producer_init_c()` interface.

For example, the `.debug_line` section's third data element (in a compilation unit) is the offset from the beginning of the `.debug_info` section of the compilation unit entry for this `.debug_line` set. The relocation entry in `.rel.debug_line` for this offset must have the relocation symbol index of the symbol `.debug_info` returned by the callback of that section-creation through the pointer `sect_name_index`.

On failure, the function should return -1 and set the `error` integer to an error code.

Nothing in libdwarf actually depends on the section index returned being a real Elf section. The Elf section is simply useful for generating relocation records. Similarly, the Elf symbol table index returned through the `sect_name_index` must simply be an index that can be used in relocations against this section. The application will probably want to note the values passed to this function in some form, even if no Elf file is being produced.

5.1.2 dwarf_producer_init_c()