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

Copyright 1993-2006 Silicon Graphics, Inc.

Copyright 2007-2016 David Anderson.

Permission is hereby granted to copy or republish or use any or all of this document without restriction except that when publishing more than a small amount of the document please acknowledge Silicon Graphics, Inc and David Anderson.

This document is distributed in the hope that it would be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

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 before reading this document.

A very few functions in the Producer Library follow the error-returns as defined in "*A Consumer Library Interface to DWARF*".

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()` description below for more details.

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.

May 01 2014 The dwarf_producer_init() code has a new interface and DWARF is configured at run time by its arguments. The producer code used to be configured at configure time, but the configure time producer configure options are no longer used. The configuration was unnecessarily complicated: the run-time configuration is simpler to understand.

September 10, 2016 Beginning the process of creating new interfaces so that checking for error is consistent across all calls (as is done in the consumer library). The old interfaces are kept and supported so we have binary and source compatibility with old code.

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 - startlabel #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()`. 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 that all such libdwarf-allocated memory is freed by `dwarf_producer_finish_a(dbg)` or `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

The read-only properties specified in the consumer interface document do not generally apply to the functions described here.

4.2 Storage Deallocation

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

4.3 Error Handling

In general any error detected by the producer should be considered fatal. That is, it is impossible to produce correct output so producing anything seems questionable.

The original producer interfaces tended to return a pointer or a large integer as a result and required the caller to cast that value to determine if it was actually a -1 meaning there was an error.

Beginning in September 2016 additional interfaces are being added to eliminate the necessity for callers to do this ugly casting of results. The revised functions return `DW_DLV_OK`, or `DW_DLV_ERROR`. (which are small signed integers) and will have an additional pointer argument that will provide the value that used to be the return value. This will make the interfaces type-safe.

The function `dwarf_get_section_bytes_a()` can also return `DW_DLV_NO_ENTRY`.

The original interfaces will remain. Binary and source compatibility for old code using the older interfaces is retained.

The list of new functions added to create interfaces with the simpler return value is:

dwarf_add_die_to_debug_a()
dwarf_new_die_a()
dwarf_die_link_a()
dwarf_transform_to_disk_form_a()
dwarf_get_section_bytes_a()
dwarf_producer_finish_a()
(More to be added as time permits).

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 following sections describe these functions.

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.*

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().

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_a() 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_a().

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()

```
int dwarf_producer_init(
        Dwarf_Unsigned flags,
        Dwarf_Callback_Func func,
        Dwarf_Handler errhand,
        Dwarf_Ptr errarg,
        void *    user_data
        const char *isa_name,
        const char *dwarf_version,
        const char *extra,
        Dwarf_P_Debug *dbg_returned,
        Dwarf_Error *error)
```

The function dwarf_producer_init() returns a new Dwarf_P_Debug descriptor that can be used to add Dwarf information to the object. On success it returns DW_DLV_OK. On error it returns DW_DLV_ERROR. 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).

The flags values (to be OR'd together in the flags field in the calling code) 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.

The flag bit DW_DLC_POINTER64 (or DW_DLC_SIZE_64) Indicates the target has a 64 bit (8 byte) address size. The flag bit DW_DLC_POINTER32 (or DW_DLC_SIZE_32) Indicates the target has a 32 bit (4 byte) address size. If none of these pointer sizes is passed in DW_DLC_POINTER32 is assumed.

The flag bit DW_DLC_OFFSET32 indicates that 32bit offsets should be used in the generated DWARF. The flag bit DW_DLC_OFFSET64 DW_DLC_OFFSET_SIZE_64 indicates that 64bit offsets should be used in the generated DWARF.

The flag bit DW_DLC_IRIX_OFFSET64 indicates that the generated DWARF should use the early (pre DWARF3) IRIX method of generating 64 bit offsets. In this case DW_DLC_POINTER64 should also be passed in, and the isa_name passed in (see below) should be "irix".

If DW_DLC_TARGET_BIGENDIAN or 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/IRIX 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.

When 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,
    Dwarf_Unsigned*     sect_name_index,
    void *              user_data,
    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.

The sect_name_index field is a field you use to pass a symbol index back to libdwarf. In Elf, each section gets an elf symbol table entry so that relocations have an address to refer to (relocations rely on addresses in the Elf symbol table). You will create the Elf symbol table, so you have to tell libdwarf the index to put into relocation records for the section newly defined here.

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.

Use the dwarf_producer_init_c() interface instead of this 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 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.

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_OK 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_OK 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 user_data argument is not examined by libdwarf. It is passed to user code in all calls by libdwarf to the Dwarf_Callback_Func() function and may be used by consumer code for the consumer's own purposes. Typical uses might be to pass in a pointer to some user data structure or to pass an integer that somehow is useful to the libdwarf-using code.

The isa_name argument must be non-null and contain one of the strings defined in the isa_relocs array in pro_init.c: "irix","mips","x86", "x86_64","arm","arm64","ppc","ppc64", "sparc". The names are not strictly ISA names (nor ABI names) but a hopefully-meaningful mixing of the concepts of ISA and ABI. The intent is mainly to define relocation codes applicable to DW_DLC_STREAM_RELOCATIONS. New isa_name values will be provided as users request. In the "irix" case a special relocation is defined so a special CIE reference field can be created (if and only if the augmentation string is "z").

The dwarf_version argument should be one of "V2", "V3", "V4", "V5" to indicate which DWARF version is the overall format to be emitted. Individual section version numbers will obey the standard for that overall DWARF version. Initially only "V2" is supported.

The extra argument is intended to support a comma-separated list of as-yet-undefined options. Passing in a null pointer or an empty string is acceptable if no such options are needed or used. All-lowercase option names are reserved to the libdwarf implementation itself (specific implementations may want to use a leading upper-case letter for additional options).

The error argument is set through the pointer to return specific error if error is non-null and and there is an error. The error details will be passed back through this pointer argument.

5.1.2 dwarf_pro_set_default_string_form()

```
int dwarf_pro_set_default_string_form(
        Dwarf_P_Debug *dbg,
        int            desired_form,
        Dwarf_Error *error)
```

The function dwarf_pro_set_default_string_form() sets the Dwarf_P_Debug descriptor to favor one of the two allowed values: DW_FORM_string (the default) or DW_FORM_strp.

When DW_FORM_strp is selected very short names will still use form DW_FORM_string .

The function should be called immediately after a successful call to dwarf_producer_init().

Strings for DW_FORM_strp are not duplicated in the .debug_str section: each unique string appears exactly once.

On success it returns DW_DLV_OK. On error it returns DW_DLV_ERROR.

5.1.3 dwarf_transform_to_disk_form_a()

```
int dwarf_transform_to_disk_form_a(
        Dwarf_P_Debug dbg,
        Dwarf_Signed *chunk_count_out,
        Dwarf_Error* error)
```

New September 2016. The function dwarf_transform_to_disk_form_a() is new in September 2016. It produces the same result as dwarf_transform_to_disk_form() but returns the count through the new pointer argument chunk_count_out .

On success it returns DW_DLV_OK and sets chunk_count_out to the number of chunks of section data to be accessed by dwarf_get_section_bytes_a() .

It turns the DIE and other information specified for this Dwarf_P_Debug into a stream of bytes for each section being produced. These byte streams can be retrieved from the Dwarf_P_Debug by calls to dwarf_get_section_bytes_a() (see below).

In case of error dwarf_transform_to_disk_form() returns DW_DLV_NOCOUNT.

When successful dwarf_transform_to_disk_form() returns the number of chunks of section data to be accessed by dwarf_get_section_bytes_a() (see below) and the section data provided your code will insert into an object file or the like. Each section of the resulting object is typically many small chunks. Each chunk has a section index and a length as well as a pointer to a block of data (see dwarf_get_section_bytes_a()).

For each unique section being produced dwarf_transform_to_disk_form() calls the Dwarf_Callback_Func exactly once. The callback provides the connection between Elf sections (which we presume is the object format to be emitted) and the libdwarf() internal section numbering.

For DW_DLC_STREAM_RELOCATIONS a call to Dwarf_Callback_Func is made by libdwarf for each relocation section. Calls to dwarf_get_section_bytes_a() (see below). allow the dwarf_transform_to_disk_form() caller to get byte streams and write them to an object file as desired, just as with the other sections of the object being created.

For DW_DLC_SYMBOLIC_RELOCATIONS the user code should use dwarf_get_relocation_info_count() and dwarf_get_relocation_info() to retrieve the relocation info generated by dwarf_transform_to_disk_form() and do something with it.

On failure it returns DW_DLV_ERROR and returns an error pointer through *error .

5.1.4 dwarf_transform_to_disk_form()

```
Dwarf_Signed dwarf_transform_to_disk_form(
        Dwarf_P_Debug dbg,
        Dwarf_Error* error)
```

The function dwarf_transform_to_disk_form() is the original call to generate output and a better interface is used by dwarf_transform_to_disk_form_a() though both do the same work and have the same meaning.

5.1.5 dwarf_get_section_bytes_a()

```
int dwarf_get_section_bytes_a(
        Dwarf_P_Debug dbg,
        Dwarf_Signed dwarf_section,
        Dwarf_Signed *elf_section_index,
        Dwarf_Unsigned *length,
        Dwarf_Ptr      *section_bytes,
        Dwarf_Error* error)
```

The function dwarf_get_section_bytes_a() must be called repetitively, with the index dwarf_section starting at 0 and continuing for the number of sections returned by dwarf_transform_to_disk_form_a() .

It returns DW_DLV_NO_ENTRY to indicate that there are no more sections of Dwarf information. Normally one would index through using the sectioncount from dwarf_transform_to_disk_form_a() so DW_DLV_NO_ENTRY would never be seen.

For each successful return (return value DW_DLV_OK), *section_bytes points to *length bytes of data that are normally added to the output object in Elf section *elf_section by the producer application. It is illegal to call these in any order other than 0 through N-1 where N is the number of dwarf sections returned by dwarf_transform_to_disk_form() . The elf section number is returned through the pointer elf_section_index.

The dwarf_section number is ignored: the data is returned as if the caller passed in the correct dwarf_section numbers in the required sequence.

In case of an error, DW_DLV_ERROR is returned and the error argument is set to indicate the error.

There is no requirement that the section bytes actually be written to an elf file. For example, consider the .debug_info section and its relocation section (the call back function would resulted in assigning 'section' numbers and the link field to tie these together (.rel.debug_info would have a link to .debug_info). One could examine the relocations, split the .debug_info data at relocation boundaries, emit byte streams (in hex) as assembler output, and at each relocation point, emit an assembler directive with a symbol name for the assembler. Examining the relocations is awkward though. It is much better to use dwarf_get_section_relocation_info()

The memory space of the section byte stream is freed by the dwarf_producer_finish_a() call (or would be if the dwarf_producer_finish_a() was actually correct), along with all the other space in use with that Dwarf_P_Debug.

5.1.6 dwarf_get_section_bytes()

```
Dwarf_Ptr dwarf_get_section_bytes(
        Dwarf_P_Debug dbg,
        Dwarf_Signed dwarf_section,
        Dwarf_Signed *elf_section_index,
        Dwarf_Unsigned *length,
        Dwarf_Error* error)
```

Beginning in September 2016 one should call dwarf_get_section_bytes_a()
in preference to dwarf_get_section_bytes() as the former makes checking for errors easier.

The function dwarf_get_section_bytes() must be called repetitively, with the index dwarf_section starting at 0 and continuing for the number of sections returned by dwarf_transform_to_disk_form() .

It returns NULL to indicate that there are no more sections of Dwarf information. Normally one would index through using the sectioncount from dwarf_transform_to_disk_form_a() so NULL would never be seen.

For each non-NULL return, the return value points to *length bytes of data that are normally added to the output object in Elf section *elf_section by the producer application. The elf section number is returned through the pointer elf_section_index.

In case of an error, DW_DLV_BADADDR is returned and the error argument is set to indicate the error.

It is illegal to call these in any order other than 0 through N-1 where N is the number of dwarf sections returned by dwarf_transform_to_disk_form() . The dwarf_section number is actually ignored: the data is returned as if the caller passed in the correct dwarf_section numbers in the required sequence. The error argument is not used.

There is no requirement that the section bytes actually be written to an elf file. For example, consider the .debug_info section and its relocation section (the call back function would resulted in assigning 'section' numbers and the link field to tie these together (.rel.debug_info would have a link to .debug_info). One could examine the relocations, split the .debug_info data at relocation boundaries, emit byte streams (in hex) as assembler output, and at each relocation point, emit an assembler directive with a symbol name for the assembler. Examining the relocations is awkward though. It is much better to use dwarf_get_section_relocation_info()

The memory space of the section byte stream is freed by the dwarf_producer_finish_a() call (or would be if the dwarf_producer_finish_a() was actually correct), along with all the other space in use with that Dwarf_P_Debug.

5.1.7 dwarf_get_relocation_info_count()

```
int  dwarf_get_relocation_info_count(
        Dwarf_P_Debug dbg,
        Dwarf_Unsigned *count_of_relocation_sections ,
        int            *drd_buffer_version,
        Dwarf_Error* error)
```

The function `dwarf_get_relocation_info()` returns, through the pointer `count_of_relocation_sections`, the number of times that `dwarf_get_relocation_info()` should be called.

The function `dwarf_get_relocation_info()` returns DW_DLV_OK if the call was successful (the `count_of_relocation_sections` is therefore meaningful, though `count_of_relocation_sections` could be zero).

`*drd_buffer_version` is the value 2. If the structure pointed to by the `*reldata_buffer` changes this number will change. The application should verify that the number is the version it understands (that it matches the value of DWARF_DRD_BUFFER_VERSION (from libdwarf.h)). The value 1 version was never used in production MIPS libdwarf (version 1 did exist in source).

It returns DW_DLV_NO_ENTRY if `count_of_relocation_sections` is not meaningful because `DW_DLC_SYMBOLIC_RELOCATIONS` was not passed to the `dwarf_producer_init_c()` `dwarf_producer_init_b()` or `dwarf_producer_init()` call (whichever one was used).

It returns DW_DLV_ERROR if there was an error, in which case `count_of_relocation_sections` is not meaningful.

5.1.8 dwarf_get_relocation_info()

```
int dwarf_get_relocation_info(
        Dwarf_P_Debug dbg,
        Dwarf_Signed *elf_section_index,
        Dwarf_Signed *elf_section_index_link,
        Dwarf_Unsigned *relocation_buffer_count,
        Dwarf_Relocation_Data *reldata_buffer,
        Dwarf_Error* error)
```

The function `dwarf_get_relocation_info()` should normally be called repetitively, for the number of relocation sections that `dwarf_get_relocation_info_count()` indicated exist.

It returns `DW_DLV_OK` to indicate that valid values are returned through the pointer arguments. The `error` argument is not set.

It returns DW_DLV_NO_ENTRY if there are no entries (the count of relocation arrays is zero.). The `error` argument is not set.

It returns `DW_DLV_ERROR` if there is an error. Calling `dwarf_get_relocation_info()` more than the number of times indicated by `dwarf_get_relocation_info_count()` (without an intervening call to `dwarf_reset_section_bytes()`) results in a return of `DW_DLV_ERROR` once past the valid count. The `error` argument is set to indicate the error.

Now consider the returned-through-pointer values for DW_DLV_OK .

*elf_section_index is the 'elf section index' of the section implied by this group of relocations.

*elf_section_index_link is the section index of the section that these relocations apply to.

*relocation_buffer_count is the number of array entries of relocation information in the array pointed to by *reldata_buffer .

*reldata_buffer points to an array of 'struct Dwarf_Relocation_Data_s' structures.

The version 2 array information is as follows:

```
enum Dwarf_Rel_Type {dwarf_drt_none,
        dwarf_drt_data_reloc,
        dwarf_drt_segment_rel,
        dwarf_drt_first_of_length_pair,
        dwarf_drt_second_of_length_pair
};
typedef struct Dwarf_Relocation_Data_s  * Dwarf_Relocation_Data;
struct Dwarf_Relocation_Data_s {
    unsigned char       drd_type;   /* contains Dwarf_Rel_Type */
    unsigned char       drd_length; /* typically 4 or 8 */
    Dwarf_Unsigned      drd_offset; /* where the data to reloc is */
    Dwarf_Unsigned      drd_symbol_index;
};
```

The Dwarf_Rel_Type enum is encoded (via casts if necessary) into the single unsigned char drd_type field to control the space used for this information (keep the space to 1 byte).

The unsigned char drd_length field holds the size in bytes of the field to be relocated. So for elf32 object formats with 32 bit apps, drd_length will be 4. For objects with MIPS -64 contents, drd_length will be 8. For some dwarf 64 bit environments, such as ia64, drd_length is 4 for some relocations (file offsets, for example) and 8 for others (run time addresses, for example).

If drd_type is dwarf_drt_none, this is an unused slot and it should be ignored.

If drd_type is dwarf_drt_data_reloc this is an ordinary relocation. The relocation type means either (R_MIPS_64) or (R_MIPS_32) (or the like for the particular ABI. drd_length gives the length of the field to be relocated. drd_offset is an offset (of the value to be relocated) in the section this relocation stuff is linked to. drd_symbol_index is the symbol index (if elf symbol indices were provided) or the handle to arbitrary information (if that is what the caller passed in to the relocation-creating dwarf calls) of the symbol that the relocation is relative to.

When drd_type is dwarf_drt_first_of_length_pair the next data record will be drt_second_of_length_pair and the drd_offset of the two data records will match. The relevant 'offset' in the section this reloc applies to should contain a symbolic pair like

```
        .word    second_symbol - first_symbol
```

to generate a length. drd_length gives the length of the field to be relocated.

drt_segment_rel means (R_MIPS_SCN_DISP) is the real relocation (R_MIPS_SCN_DISP applies to exception tables and this part may need further work). drd_length gives the length of the field to be relocated.

The memory space of the section byte stream is freed by the dwarf_producer_finish_a() call (or would be if the dwarf_producer_finish_a() was actually correct), along with all the other space in use with that Dwarf_P_Debug.

5.1.9 dwarf_reset_section_bytes()

```
void dwarf_reset_section_bytes(
        Dwarf_P_Debug dbg
        )
```

The function dwarf_reset_section_bytes() is used to reset the internal information so that dwarf_get_section_bytes() will begin (on the next call) at the initial dwarf section again. It also resets so that calls to dwarf_get_relocation_info() will begin again at the initial array of relocation information.

Some dwarf producers need to be able to run through the dwarf_get_section_bytes() and/or the dwarf_get_relocation_info() calls more than once and this call makes additional passes possible. The set of Dwarf_Ptr values returned is identical to the set returned by the first pass. It is acceptable to call this before finishing a pass of dwarf_get_section_bytes() or dwarf_get_relocation_info() calls. No errors are possible as this just resets some internal pointers. It is unwise to call this before dwarf_transform_to_disk_form() has been called.

5.1.10 dwarf_pro_get_string_stats()

```
int dwarf_pro_get_string_stats(
    Dwarf_P_Debug dbg,
    Dwarf_Unsigned * str_count,
    Dwarf_Unsigned * str_total_length,
    Dwarf_Unsigned * strp_count_debug_str,
    Dwarf_Unsigned * strp_len_debug_str,
    Dwarf_Unsigned * strp_reused_count,
    Dwarf_Unsigned * strp_reused_len,
    Dwarf_Error* error)
```

If it returns DW_DLV_OK the function dwarf_pro_get_string_stats() returns information about how DW_AT_name etc strings were stored in the output object. The values suggest how much string duplication was detected in the DWARF being created.

Call it after calling dwarf_transform_to_disk_form() and before calling dwarf_producer_finish_a() . It has no effect on the object being output.

On error it returns DW_DLV_ERROR and sets error through the pointer.

5.1.11 dwarf_producer_finish_a()

```
int dwarf_producer_finish_a(
        Dwarf_P_Debug dbg,
        Dwarf_Error* error)
```

This is new in September 2016 and has the newer interface style, but is otherwise identical to dwarf_producer_finish() .

The function dwarf_producer_finish_a() should be called after all the bytes of data have been copied somewhere (normally the bytes are written to disk). It frees all dynamic space allocated for dbg, include space for the structure pointed to by dbg. This should not be called till the data have been copied or written to disk or are no longer of interest. It returns DW_DLV_OK if successful.

On error it returns DW_DLV_ERROR and sets error through the pointer.

5.1.12 dwarf_producer_finish()

```
Dwarf_Unsigned dwarf_producer_finish(
        Dwarf_P_Debug dbg,
        Dwarf_Error* error)
```

This is the original interface. It works but calling dwarf_producer_finish_a() is preferred as it matches the latest libdwarf interface standard.

The function dwarf_producer_finish() should be called after all the bytes of data have been copied somewhere (normally the bytes are written to disk). It frees all dynamic space allocated for dbg, include space for the structure pointed to by dbg. This should not be called till the data have been copied or written to disk or are no longer of interest. It returns zero if successful.

On error it returns DW_DLV_NOCOUNT and sets error through the pointer.

5.2 Debugging Information Entry Creation

The functions in this section add new DIEs to the object, and also the relationships among the DIE to be specified by linking them up as parents, children, left or right siblings of each other. In addition, there is a function that marks the root of the graph thus created.

5.2.1 dwarf_add_die_to_debug_a()

```
int dwarf_add_die_to_debug_a(
        Dwarf_P_Debug dbg,
        Dwarf_P_Die first_die,
        Dwarf_Error *error)
```

The function dwarf_add_die_to_debug_a() indicates to Libdwarf the root DIE of the DIE graph that has been built so far. It is intended to mark the compilation-unit DIE for the object represented by dbg. The root DIE is specified by first_die.

It returns DW_DLV_OK on success, and DW_DLV_error on error.

5.2.2 dwarf_add_die_to_debug()

```
Dwarf_Unsigned dwarf_add_die_to_debug(
        Dwarf_P_Debug dbg,
        Dwarf_P_Die first_die,
        Dwarf_Error *error)
```

This is the original form of the call. Use `dwarf_add_die_to_debug_a() instead.`

`It returns 0 on success, and DW_DLV_NOCOUNT on error.`

5.2.3 dwarf_new_die_a()

```
int dwarf_new_die_a(
        Dwarf_P_Debug dbg,
        Dwarf_Tag new_tag,
        Dwarf_P_Die parent,
        Dwarf_P_Die child,
        Dwarf_P_Die left_sibling,
        Dwarf_P_Die right_sibling,
        Dwarf_P_Die *die_out,
        Dwarf_Error *error)
```

New September 2016. On success `dwarf_new_die_a()` returns DW_DLV_OK and creates a new `DIE` with its parent, child, left sibling, and right sibling `DIE`s specified by `parent`, `child`, `left_sibling`, and `right_sibling`, respectively. The new die is passed to the caller via the argument `die_out()` . There is no requirement that all of these `DIE`s be specified, i.e. any of these descriptors may be `NULL`. If none is specified, this will be an isolated `DIE`. A `DIE` is transformed to disk form by `dwarf_transform_to_disk_form()` only if there is a path from the `DIE` specified by `dwarf_add_die_to_debug` to it.

The value of `new_tag` is the tag which is given to the new `DIE`. `parent`, `child`, `left_sibling`, and `right_sibling` are pointers to establish links to existing `DIE`s. Only one of `parent`, `child`, `left_sibling`, and `right_sibling` may be non-NULL. If `parent` (`child`) is given, the `DIE` is linked into the list after (before) the `DIE` pointed to. If `left_sibling` (`right_sibling`) is given, the `DIE` is linked into the list after (before) the `DIE` pointed to.

To add attributes to the new `DIE`, use the `Attribute Creation` functions defined in the next section.

On failure `dwarf_new_die_a()` returns DW_DLV_ERROR and sets `*error`.

5.2.4 dwarf_new_die()

```
Dwarf_P_Die dwarf_new_die(
        Dwarf_P_Debug dbg,
        Dwarf_Tag new_tag,
        Dwarf_P_Die parent,
        Dwarf_P_Die child,
        Dwarf_P_Die left_sibling,
        Dwarf_P_Die right_sibling,
        Dwarf_Error *error)
```

This is the original form of the function and users should switch to calling `dwarf_new_die_a()` instead to use the newer interface. See `dwarf_new_die_a()` for details (both functions do the same thing).

5.2.5 dwarf_die_link_a()

```
Dwarf_P_Die dwarf_die_link_a(
        Dwarf_P_Die die,
        Dwarf_P_Die parent,
        Dwarf_P_Die child,
        Dwarf_P_Die left-sibling,
        Dwarf_P_Die right_sibling,
        Dwarf_Error *error)
```

New September 2016. On success the function `dwarf_die_link_a()` returns `DW_DLV_OK` and links an existing `DIE` described by the given `die` to other existing `DIE`s. The given `die` can be linked to a parent `DIE`, a child `DIE`, a left sibling `DIE`, or a right sibling `DIE` by specifying non-NULL `parent`, `child`, `left_sibling`, and `right_sibling` `Dwarf_P_Die` descriptors.

Only one of `parent`, `child`, `left_sibling`, and `right_sibling` may be non-NULL. If `parent` (`child`) is given, the `DIE` is linked into the list after (before) the `DIE` pointed to. If `left_sibling` (`right_sibling`) is given, the `DIE` is linked into the list after (before) the `DIE` pointed to. Non-NULL links overwrite the corresponding links the given `die` may have had before the call to `dwarf_die_link_a()` .

If there is an error `dwarf_die_link_a()` returns `DW_DLV_ERROR` and sets `error` with the specific applicable error code.

5.2.6 dwarf_die_link()

```
Dwarf_P_Die dwarf_die_link(
        Dwarf_P_Die die,
        Dwarf_P_Die parent,
        Dwarf_P_Die child,
        Dwarf_P_Die left-sibling,
        Dwarf_P_Die right_sibling,
        Dwarf_Error *error)
```

This is the original function to link `DIEs` together. The function does the same thing as `dwarf_die_link_a()` but. the newer function is simpler to work with.

5.3 DIE Markers

DIE markers provide a way for a producer to extract DIE offsets from DIE generation. The markers do not influence the generation of DWARF, they simply allow a producer to extract .debug_info offsets for whatever purpose the producer finds useful (for example, a producer might want some unique other section unknown to libdwarf to know a particular DIE offset).

One marks one or more DIEs as desired any time before calling `dwarf_transform_to_disk_form()`.

After calling `dwarf_transform_to_disk_form()` call `dwarf_get_die_markers()` which has the offsets where the marked DIEs were written in the generated .debug_info data.

5.3.1 dwarf_add_die_marker()

```
Dwarf_Unsigned dwarf_add_die_marker(
        Dwarf_P_Debug dbg,
        Dwarf_P_Die die,
        Dwarf_Unsigned marker,
        Dwarf_Error *error)
```

The function `dwarf_add_die_marker()` writes the value `marker` to the `DIE` descriptor given by `die`. Passing in a marker of 0 means 'there is no marker' (zero is the default in DIEs).

It returns `0`, on success. On error it returns `DW_DLV_NOCOUNT`.

5.3.2 dwarf_get_die_marker()

```
Dwarf_Unsigned dwarf_get_die_marker(
        Dwarf_P_Debug dbg,
        Dwarf_P_Die die,
        Dwarf_Unsigned *marker,
        Dwarf_Error *error)
```

The function `dwarf_get_die_marker()` returns the current marker value for this DIE through the pointer `marker`. A marker value of 0 means 'no marker was set'.

It returns `0`, on success. On error it returns `DW_DLV_NOCOUNT`.

5.3.3 dwarf_get_die_markers()

```
Dwarf_Unsigned dwarf_get_die_markers(
        Dwarf_P_Debug dbg,
        Dwarf_P_Marker * marker_list,
        Dwarf_Unsigned *marker_count,
        Dwarf_Error *error)
```

The function `dwarf_get_die_marker()` returns a pointer to an array of `Dwarf_P_Marker` pointers to `struct Dwarf_P_Marker_s` structures through the pointer `marker_list`. The array length is returned through the pointer `marker_count`.

The call is only meaningful after a call to `dwarf_transform_to_disk_form()` as the transform call creates the `struct Dwarf_P_Marker_s` structures, one for each DIE generated for .debug_info (but only for DIEs that had a non-zero marker value). The field `ma_offset` in the structure is set during generation of the .debug_info byte stream. The field `ma_marker` in the structure is a copy of the DIE marker of the DIE given that offset.

It returns `0`, on success. On error it returns `DW_DLV_BADADDR` (if there are no markers it returns `DW_DLV_BADADDR`).

5.4 Attribute Creation

The functions in this section add attributes to a `DIE`. These functions return a `Dwarf_P_Attribute` descriptor that represents the attribute added to the given `DIE`. In most cases the return value is only useful

to determine if an error occurred.

Some of the attributes have values that are relocatable. They need a symbol with respect to which the linker will perform relocation. This symbol is specified by means of an index into the Elf symbol table for the object (of course, the symbol index can be more general than an index).

5.4.1 dwarf_add_AT_location_expr()

```
Dwarf_P_Attribute dwarf_add_AT_location_expr(
        Dwarf_P_Debug dbg,
        Dwarf_P_Die ownerdie,
        Dwarf_Half attr,
        Dwarf_P_Expr loc_expr,
        Dwarf_Error *error)
```

The function `dwarf_add_AT_location_expr()` adds the attribute specified by `attr` to the `DIE` descriptor given by `ownerdie`. The attribute should be one that has a location expression as its value. The location expression that is the value is represented by the `Dwarf_P_Expr` descriptor `loc_expr`. It returns the `Dwarf_P_Attribute` descriptor for the attribute given, on success. On error it returns `DW_DLV_BADADDR`.

5.4.2 dwarf_add_AT_name()

```
Dwarf_P_Attribute dwarf_add_AT_name(
        Dwarf_P_Die ownerdie,
        char *name,
        Dwarf_Error *error)
```

The function `dwarf_add_AT_name()` adds the string specified by `name` as the value of the `DW_AT_name` attribute for the given `DIE`, `ownerdie`. It returns the `Dwarf_P_attribute` descriptor for the `DW_AT_name` attribute on success. On error, it returns `DW_DLV_BADADDR`.

5.4.3 dwarf_add_AT_comp_dir()

```
Dwarf_P_Attribute dwarf_add_AT_comp_dir(
        Dwarf_P_Die ownerdie,
        char *current_working_directory,
        Dwarf_Error *error)
```

The function `dwarf_add_AT_comp_dir()` adds the string given by `current_working_directory` as the value of the `DW_AT_comp_dir` attribute for the `DIE` described by the given `ownerdie`. It returns the `Dwarf_P_Attribute` for this attribute on success. On error, it returns `DW_DLV_BADADDR`.

5.4.4 dwarf_add_AT_producer()

```
Dwarf_P_Attribute dwarf_add_AT_producer(
        Dwarf_P_Die ownerdie,
        char *producer_string,
        Dwarf_Error *error)
```

The function dwarf_add_AT_producer() adds the string given by producer_string as the value of the DW_AT_producer attribute for the DIE given by ownerdie. It returns the Dwarf_P_Attribute descriptor representing this attribute on success. On error, it returns DW_DLV_BADADDR.

5.4.5 dwarf_add_AT_any_value_sleb()

```
Dwarf_P_Attribute dwarf_add_AT_any_value_sleb(
        Dwarf_P_Die ownerdie,
        Dwarf_Half  attrnum,
        Dwarf_Signed signed_value,
        Dwarf_Error *error)
```

The function dwarf_add_AT_any_value_sleb() adds the given Dwarf_Signed value signed_value as the value of the DW_AT_const_value attribute for the DIE described by the given ownerdie.

The FORM of the output value is DW_FORM_sdata (signed leb number) and the attribute will be DW_AT_const_value.

It returns the Dwarf_P_Attribute descriptor for this attribute on success.

On error, it returns DW_DLV_BADADDR.

The function was created 13 August 2013.

5.4.6 dwarf_add_AT_const_value_signedint()

```
Dwarf_P_Attribute dwarf_add_AT_const_value_signedint(
        Dwarf_P_Die ownerdie,
        Dwarf_Signed signed_value,
        Dwarf_Error *error)
```

The function dwarf_add_AT_const_value_signedint() adds the given Dwarf_Signed value signed_value as the value of the DW_AT_const_value attribute for the DIE described by the given ownerdie.

The FORM of the output value is DW_FORM_data<n> (signed leb number) and the attribute will be DW_AT_const_value.

With this interface and output, there is no way for consumers to know from the FORM that the value is signed.

It returns the Dwarf_P_Attribute descriptor for this attribute on success.

On error, it returns DW_DLV_BADADDR.

5.4.7 dwarf_add_AT_any_value_uleb()

```
Dwarf_P_Attribute dwarf_add_AT_any_value_uleb(
        Dwarf_P_Die ownerdie,
        Dwarf_Half  attrnum,
        Dwarf_Unsigned unsigned_value,
        Dwarf_Error *error)
```

The function dwarf_add_AT_any_value_uleb() adds the given Dwarf_Unsigned value unsigned_value as the value of the attrnum attribute for the DIE described by the given ownerdie.

The FORM of the output value is DW_FORM_udata (unsigned leb number) and the attribute is attrnum.

It returns the Dwarf_P_Attribute descriptor for this attribute on success.

On error, it returns DW_DLV_BADADDR.

The function was created 13 August 2013.

5.4.8 dwarf_add_AT_const_value_unsignedint()

```
Dwarf_P_Attribute dwarf_add_AT_const_value_unsignedint(
        Dwarf_P_Die ownerdie,
        Dwarf_Unsigned unsigned_value,
        Dwarf_Error *error)
```

The function dwarf_add_AT_const_value_unsignedint() adds the given Dwarf_Unsigned value unsigned_value as the value of the DW_AT_const_value attribute for the DIE described by the given ownerdie.

The FORM of the output value is DW_FORM_data<n> and the attribute will be DW_AT_const_value.

With this interface and output, there is no way for consumers to know from the FORM that the value is signed.

It returns the Dwarf_P_Attribute descriptor for this attribute on success.

On error, it returns DW_DLV_BADADDR.

5.4.9 dwarf_add_AT_const_value_string()

```
Dwarf_P_Attribute dwarf_add_AT_const_value_string(
        Dwarf_P_Die ownerdie,
        char *string_value,
        Dwarf_Error *error)
```

The function dwarf_add_AT_const_value_string() adds the string value given by string_value as the value of the DW_AT_const_value attribute for the DIE described by the given ownerdie. It returns the Dwarf_P_Attribute descriptor for this attribute on success. On error, it returns DW_DLV_BADADDR.

5.4.10 dwarf_add_AT_targ_address()

```
Dwarf_P_Attribute dwarf_add_AT_targ_address(
        Dwarf_P_Debug dbg,
        Dwarf_P_Die ownerdie,
        Dwarf_Half attr,
        Dwarf_Unsigned pc_value,
        Dwarf_Signed sym_index,
        Dwarf_Error *error)
```

The function `dwarf_add_AT_targ_address()` adds an attribute that belongs to the "address" class to the die specified by `ownerdie`. The attribute is specified by `attr`, and the object that the `DIE` belongs to is specified by `dbg`. The relocatable address that is the value of the attribute is specified by `pc_value`. The symbol to be used for relocation is specified by the `sym_index`, which is the index of the symbol in the Elf symbol table.

It returns the `Dwarf_P_Attribute` descriptor for the attribute on success, and `DW_DLV_BADADDR` on error.

5.4.11 dwarf_add_AT_targ_address_b()

```
Dwarf_P_Attribute dwarf_add_AT_targ_address_b(
        Dwarf_P_Debug dbg,
        Dwarf_P_Die ownerdie,
        Dwarf_Half attr,
        Dwarf_Unsigned pc_value,
        Dwarf_Unsigned sym_index,
        Dwarf_Error *error)
```

The function `dwarf_add_AT_targ_address_b()` is identical to `dwarf_add_AT_targ_address()` except that `sym_index()` is guaranteed to be large enough that it can contain a pointer to arbitrary data (so the caller can pass in a real elf symbol index, an arbitrary number, or a pointer to arbitrary data). The ability to pass in a pointer through `sym_index()` is only usable with `DW_DLC_SYMBOLIC_RELOCATIONS`.

The `pc_value` is put into the section stream output and the `sym_index` is applied to the relocation information.

Do not use this function for attr `DW_AT_high_pc` if the value to be recorded is an offset (not a pc) [use `dwarf_add_AT_unsigned_const()` or `dwarf_add_AT_any_value_uleb()` instead].

5.4.12 dwarf_add_AT_dataref()

```
Dwarf_P_Attribute dwarf_add_AT_dataref(
        Dwarf_P_Debug dbg,
        Dwarf_P_Die ownerdie,
        Dwarf_Half attr,
        Dwarf_Unsigned pc_value,
        Dwarf_Unsigned sym_index,
        Dwarf_Error *error)
```

This is very similar to `dwarf_add_AT_targ_address_b()` but results in a different FORM (results in DW_FORM_data4 or DW_FORM_data8).

Useful for adding relocatable addresses in location lists.

`sym_index()` is guaranteed to be large enough that it can contain a pointer to arbitrary data (so the caller can pass in a real elf symbol index, an arbitrary number, or a pointer to arbitrary data). The ability to pass in a pointer through `sym_index()` is only usable with `DW_DLC_SYMBOLIC_RELOCATIONS`.

The `pc_value` is put into the section stream output and the `sym_index` is applied to the relocation information.

Do not use this function for `DW_AT_high_pc`, use `dwarf_add_AT_unsigned_const()` or `dwarf_add_AT_any_value_uleb()` [if the value to be recorded is an offset of `DW_AT_low_pc`] or `dwarf_add_AT_targ_address_b()` [if the value to be recorded is an address].

5.4.13 dwarf_add_AT_ref_address()

```
Dwarf_P_Attribute dwarf_add_AT_ref_address(
        Dwarf_P_Debug dbg,
        Dwarf_P_Die ownerdie,
        Dwarf_Half attr,
        Dwarf_Unsigned pc_value,
        Dwarf_Unsigned sym_index,
        Dwarf_Error *error)
```

This is very similar to `dwarf_add_AT_targ_address_b()` but results in a different FORM (results in `DW_FORM_ref_addr` being generated).

Useful for `DW_AT_type` and `DW_AT_import` attributes.

`sym_index()` is guaranteed to be large enough that it can contain a pointer to arbitrary data (so the caller can pass in a real elf symbol index, an arbitrary number, or a pointer to arbitrary data). The ability to pass in a pointer through `sym_index()` is only usable with `DW_DLC_SYMBOLIC_RELOCATIONS`.

The `pc_value` is put into the section stream output and the `sym_index` is applied to the relocation information.

Do not use this function for `DW_AT_high_pc`.

5.4.14 dwarf_add_AT_unsigned_const()

```
Dwarf_P_Attribute dwarf_add_AT_unsigned_const(
        Dwarf_P_Debug dbg,
        Dwarf_P_Die ownerdie,
        Dwarf_Half attr,
        Dwarf_Unsigned value,
        Dwarf_Error *error)
```

The function dwarf_add_AT_unsigned_const() adds an attribute with a Dwarf_Unsigned value belonging to the "constant" class, to the DIE specified by ownerdie. The object that the DIE belongs to is specified by dbg. The attribute is specified by attr, and its value is specified by value.

The FORM of the output will be one of the DW_FORM_data<n> forms.

It returns the Dwarf_P_Attribute descriptor for the attribute on success, and DW_DLV_BADADDR on error.

5.4.15 dwarf_add_AT_signed_const()

```
Dwarf_P_Attribute dwarf_add_AT_signed_const(
        Dwarf_P_Debug dbg,
        Dwarf_P_Die ownerdie,
        Dwarf_Half attr,
        Dwarf_Signed value,
        Dwarf_Error *error)
```

The function dwarf_add_AT_signed_const() adds an attribute with a Dwarf_Signed value belonging to the "constant" class, to the DIE specified by ownerdie. The object that the DIE belongs to is specified by dbg. The attribute is specified by attr, and its value is specified by value.

It returns the Dwarf_P_Attribute descriptor for the attribute on success, and DW_DLV_BADADDR on error.

5.4.16 dwarf_add_AT_reference()

```
Dwarf_P_Attribute dwarf_add_AT_reference(
        Dwarf_P_Debug dbg,
        Dwarf_P_Die ownerdie,
        Dwarf_Half attr,
        Dwarf_P_Die otherdie,
        Dwarf_Error *error)
```

The function dwarf_add_AT_reference() adds an attribute with a value that is a reference to another DIE in the same compilation-unit to the DIE specified by ownerdie. The object that the DIE belongs to is specified by dbg. The attribute is specified by attr, and the other DIE being referred to is specified by otherdie.

The FORM of the output will be one of the DW_FORM_data<n> forms.

This cannot generate DW_FORM_ref_addr references to DIEs in other compilation units.

It returns the Dwarf_P_Attribute descriptor for the attribute on success, and DW_DLV_BADADDR on error.

5.4.17 dwarf_add_AT_reference_b()

```
Dwarf_P_Attribute dwarf_add_AT_reference_b(
        Dwarf_P_Debug dbg,
        Dwarf_Half attr,
        Dwarf_P_Die ownerdie,
        Dwarf_P_Die otherdie,
        Dwarf_Error *error)
```

The function `dwarf_add_AT_reference_b()` is the same as `dwarf_add_AT_reference()` except that `dwarf_add_AT_reference_b()` accepts a NULL `otherdie` with the assumption that `dwarf_fixup_AT_reference_die()` will be called by user code to fill in the missing `otherdie` before the DIEs are transformed to disk form.

5.4.18 dwarf_fixup_AT_reference_die()

```
int dwarf_fixup_AT_reference_die(
        Dwarf_Half attrnum,
        Dwarf_P_Die ownerdie,
        Dwarf_P_Die otherdie,
        Dwarf_Error *error)
```

The function `dwarf_fixup_AT_reference_die()` is provided to set the NULL `otherdie` that `dwarf_add_AT_reference_b()` allows to the reference target DIE. This must be done before transforming to disk form. `attrnum()` should be the attribute number of the attribute of `Wownerdie` which is to be updated. For example, if a local forward reference was in a `WDW_AT_sibling` attribute in ownerdie, pass the value `WDW_AT_sibling` as attrnum.

Since no attribute number can appear more than once on a given DIE the `attrnum()` suffices to uniquely identify which attribute of `Wownerdie` to update

It returns either `DW_DLV_OK` (on success) or `DW_DLV_ERROR` (on error). Calling this on an attribute where `otherdie` was already set is an error.

5.4.19 dwarf_add_AT_flag()

```
Dwarf_P_Attribute dwarf_add_AT_flag(
        Dwarf_P_Debug dbg,
        Dwarf_P_Die ownerdie,
        Dwarf_Half attr,
        Dwarf_Small flag,
        Dwarf_Error *error)
```

The function `dwarf_add_AT_flag()` adds an attribute with a `Dwarf_Small` value belonging to the "flag" class, to the `DIE` specified by `ownerdie`. The object that the `DIE` belongs to is specified by `dbg`. The attribute is specified by `attr`, and its value is specified by `flag`.

It returns the `Dwarf_P_Attribute` descriptor for the attribute on success, and `DW_DLV_BADADDR` on error.

5.4.20 dwarf_add_AT_string()

```
Dwarf_P_Attribute dwarf_add_AT_string(
        Dwarf_P_Debug dbg,
        Dwarf_P_Die ownerdie,
        Dwarf_Half attr,
        char *string,
        Dwarf_Error *error)
```

The function `dwarf_add_AT_string()` adds an attribute with a value that is a character string to the `DIE` specified by `ownerdie`. The object that the `DIE` belongs to is specified by `dbg`. The attribute is specified by `attr`, and its value is pointed to by `string`.

It returns the `Dwarf_P_Attribute` descriptor for the attribute on success, and `DW_DLV_BADADDR` on error.

5.5 Expression Creation

The following functions are used to convert location expressions into blocks so that attributes with values that are location expressions can store their values as a `DW_FORM_blockn` value. This is for both .debug_info and .debug_loc expression blocks.

To create an expression, first call `dwarf_new_expr()` to get a `Dwarf_P_Expr` descriptor that can be used to build up the block containing the location expression. Then insert the parts of the expression in prefix order (exactly the order they would be interpreted in in an expression interpreter). The bytes of the expression are then built-up as specified by the user.

5.5.1 dwarf_new_expr()

```
Dwarf_Expr dwarf_new_expr(
        Dwarf_P_Debug dbg,
        Dwarf_Error *error)
```

The function `dwarf_new_expr()` creates a new expression area in which a location expression stream can be created. It returns a `Dwarf_P_Expr` descriptor that can be used to add operators to build up a location expression. It returns `NULL` on error.

5.5.2 dwarf_add_expr_gen()

```
Dwarf_Unsigned dwarf_add_expr_gen(
        Dwarf_P_Expr expr,
        Dwarf_Small opcode,
        Dwarf_Unsigned val1,
        Dwarf_Unsigned val2,
        Dwarf_Error *error)
```

The function `dwarf_add_expr_gen()` takes an operator specified by `opcode`, along with up to 2 operands specified by `val1`, and `val2`, converts it into the `Dwarf` representation and appends the bytes to the byte stream being assembled for the location expression represented by `expr`. The first operand, if present, to `opcode` is in `val1`, and the second operand, if present, is in `val2`. Both the operands may actually be signed or unsigned depending on `opcode`. It returns the number of bytes in the byte stream for

expr currently generated, i.e. after the addition of opcode. It returns DW_DLV_NOCOUNT on error.

The function dwarf_add_expr_gen() works for all opcodes except those that have a target address as an operand. This is because it does not set up a relocation record that is needed when target addresses are involved.

5.5.3 dwarf_add_expr_addr()

```
Dwarf_Unsigned dwarf_add_expr_addr(
        Dwarf_P_Expr expr,
        Dwarf_Unsigned address,
        Dwarf_Signed sym_index,
        Dwarf_Error *error)
```

The function dwarf_add_expr_addr() is used to add the DW_OP_addr opcode to the location expression represented by the given Dwarf_P_Expr descriptor, expr. The value of the relocatable address is given by address. The symbol to be used for relocation is given by sym_index, which is the index of the symbol in the Elf symbol table. It returns the number of bytes in the byte stream for expr currently generated, i.e. after the addition of the DW_OP_addr operator. It returns DW_DLV_NOCOUNT on error.

5.5.4 dwarf_add_expr_addr_b()

```
Dwarf_Unsigned dwarf_add_expr_addr_b(
        Dwarf_P_Expr expr,
        Dwarf_Unsigned address,
        Dwarf_Unsigned sym_index,
        Dwarf_Error *error)
```

The function dwarf_add_expr_addr_f() is identical to dwarf_add_expr_addr() except that sym_index() is guaranteed to be large enough that it can contain a pointer to arbitrary data (so the caller can pass in a real elf symbol index, an arbitrary number, or a pointer to arbitrary data). The ability to pass in a pointer through sym_index() is only usable with DW_DLC_SYMBOLIC_RELOCATIONS.

5.5.5 dwarf_expr_current_offset()

```
Dwarf_Unsigned dwarf_expr_current_offset(
        Dwarf_P_Expr expr,
        Dwarf_Error *error)
```

The function dwarf_expr_current_offset() returns the number of bytes currently in the byte stream for the location expression represented by the given W(Dwarf_P_Expr descriptor, expr. It returns DW_DLV_NOCOUNT on error.

5.5.6 dwarf_expr_into_block()

```
Dwarf_Addr dwarf_expr_into_block(
        Dwarf_P_Expr expr,
        Dwarf_Unsigned *length,
        Dwarf_Error *error)
```

The function `dwarf_expr_into_block()` returns the address of the start of the byte stream generated for the location expression represented by the given `Dwarf_P_Expr` descriptor, `expr`. The length of the byte stream is returned in the location pointed to by `length`. It returns `DW_DLV_BADADDR` on error.

5.6 Line Number Operations

These are operations on the .debug_line section. They provide information about instructions in the program and the source lines the instruction come from. Typically, code is generated in contiguous blocks, which may then be relocated as contiguous blocks. To make the provision of relocation information more efficient, the information is recorded in such a manner that only the address of the start of the block needs to be relocated. This is done by providing the address of the first instruction in a block using the function `dwarf_lne_set_address()`. Information about the instructions in the block are then added using the function `dwarf_add_line_entry()`, which specifies offsets from the address of the first instruction. The end of a contiguous block is indicated by calling the function `dwarf_lne_end_sequence()`.

Line number operations do not support `DW_DLC_SYMBOLIC_RELOCATIONS`.

5.6.1 dwarf_add_line_entry_b()

```
Dwarf_Unsigned dwarf_add_line_entry_b(
        Dwarf_P_Debug dbg,
        Dwarf_Unsigned file_index,
        Dwarf_Addr code_offset,
        Dwarf_Unsigned lineno,
        Dwarf_Signed column_number,
        Dwarf_Bool is_source_stmt_begin,
        Dwarf_Bool is_basic_block_begin,
        Dwarf_Bool is_epilogue_begin,
        Dwarf_Bool is_prologue_end,
        Dwarf_Unsigned  isa,
        Dwarf_Unsigned  discriminator,
        Dwarf_Error *error)
```

The function `dwarf_add_line_entry()` adds an entry to the section containing information about source lines. It specifies in `code_offset`, the address of this line. The function subtracts `code_offset` from the value given as the address of a previous line call to compute an offset, and the offset is what is recorded in the line instructions so no relocation will be needed on the line instruction generated.

The source file that gave rise to the instruction is specified by `file_index`, the source line number is specified by `lineno`, and the source column number is specified by `column_number` (column numbers begin at 1) (if the source column is unknown, specify 0). `file_index` is the index of the source file in a list of source files which is built up using the function `dwarf_add_file_decl()`.

`is_source_stmt_begin` is a boolean flag that is true only if the instruction at `code_address` is the first instruction in the sequence generated for the source line at `lineno`. Similarly, `is_basic_block_begin` is a boolean flag that is true only if the instruction at `code_address` is the first instruction of a basic block.

is_epilogue_begin is a boolean flag that is true only if the instruction at code_address is the first instruction in the sequence generated for the function epilogue code.

Similarly, is_prolgue_end is a boolean flag that is true only if the instruction at code_address is the last instruction of the sequence generated for the function prologue.

isa should be zero unless the code at code_address is generated in a non-standard isa. The values assigned to non-standard isas are defined by the compiler implementation.

discriminator should be zero unless the line table needs to distinguish among multiple blocks associated with the same source file, line, and column. The values assigned to discriminator are defined by the compiler implementation.

It returns 0 on success, and DW_DLV_NOCOUNT on error.

This function is defined as of December 2011.

5.6.2 dwarf_add_line_entry()

```
Dwarf_Unsigned dwarf_add_line_entry(
        Dwarf_P_Debug dbg,
        Dwarf_Unsigned file_index,
        Dwarf_Addr code_offset,
        Dwarf_Unsigned lineno,
        Dwarf_Signed column_number,
        Dwarf_Bool is_source_stmt_begin,
        Dwarf_Bool is_basic_block_begin,
        Dwarf_Error *error)
```

This function is the same as dwarf_add_line_entry_b() except this older version is missing the new DWARF3/4 line table fields.

5.6.3 dwarf_lne_set_address()

```
Dwarf_Unsigned dwarf_lne_set_address(
        Dwarf_P_Debug dbg,
        Dwarf_Addr offs,
        Dwarf_Unsigned symidx,
        Dwarf_Error *error)
```

The function dwarf_lne_set_address() sets the target address at which a contiguous block of instructions begin. Information about the instructions in the block is added to .debug_line using calls to dwarfdwarf_add_line_entry() which specifies the offset of each instruction in the block relative to the start of the block. This is done so that a single relocation record can be used to obtain the final target address of every instruction in the block.

The relocatable address of the start of the block of instructions is specified by offs. The symbol used to relocate the address is given by symidx, which is normally the index of the symbol in the Elf symbol table.

It returns 0 on success, and DW_DLV_NOCOUNT on error.

5.6.4 dwarf_lne_end_sequence()

```
Dwarf_Unsigned dwarf_lne_end_sequence(
        Dwarf_P_Debug dbg,
        Dwarf_Addr    address;
        Dwarf_Error *error)
```

The function dwarf_lne_end_sequence() indicates the end of a contiguous block of instructions. address() should be just higher than the end of the last address in the sequence of instructions. Before the next block of instructions (if any) a call to dwarf_lne_set_address() will have to be made to set the address of the start of the target address of the block, followed by calls to dwarf_add_line_entry() for each of the instructions in the block.

It returns 0 on success, and DW_DLV_NOCOUNT on error.

5.6.5 dwarf_add_directory_decl()

```
Dwarf_Unsigned dwarf_add_directory_decl(
        Dwarf_P_Debug dbg,
        char *name,
        Dwarf_Error *error)
```

The function dwarf_add_directory_decl() adds the string specified by name to the list of include directories in the statement program prologue of the .debug_line section. The string should therefore name a directory from which source files have been used to create the present object.

It returns the index of the string just added, in the list of include directories for the object. This index is then used to refer to this string. The first successful call of this function returns one, not zero, to be consistent with the directory indices that dwarf_add_file_decl() (below) expects..

dwarf_add_directory_decl() returns DW_DLV_NOCOUNT on error.

5.6.6 dwarf_add_file_decl()

```
Dwarf_Unsigned dwarf_add_file_decl(
        Dwarf_P_Debug dbg,
        char *name,
        Dwarf_Unsigned dir_idx,
        Dwarf_Unsigned time_mod,
        Dwarf_Unsigned length,
        Dwarf_Error *error)
```

The function dwarf_add_file_decl() adds the name of a source file that contributed to the present object. The name of the file is specified by name (which must not be the empty string or a null pointer, it must point to a string with length greater than 0).

In case the name is not a fully-qualified pathname, it is considered prefixed with the name of the directory specified by dir_idx (which does not mean the name is changed or physically prefixed by this producer function, we simply describe the meaning here). dir_idx is the index of the directory to be prefixed in the list builtup using dwarf_add_directory_decl(). As specified by the DWARF spec, a dir_idx of zero will be interpreted as meaning the directory of the compilation and another index must

refer to a valid directory as FIXME

time_mod gives the time at which the file was last modified, and length gives the length of the file in bytes.

It returns the index of the source file in the list built up so far using this function, on success. This index can then be used to refer to this source file in calls to dwarf_add_line_entry(). On error, it returns DW_DLV_NOCOUNT.

5.7 Fast Access (aranges) Operations

These functions operate on the .debug_aranges section.

5.7.1 dwarf_add_arange()

```
Dwarf_Unsigned dwarf_add_arange(
        Dwarf_P_Debug dbg,
        Dwarf_Addr begin_address,
        Dwarf_Unsigned length,
        Dwarf_Signed symbol_index,
        Dwarf_Error *error)
```

The function dwarf_add_arange() adds another address range to be added to the section containing address range information, .debug_aranges. The relocatable start address of the range is specified by begin_address, and the length of the address range is specified by length. The relocatable symbol to be used to relocate the start of the address range is specified by symbol_index, which is normally the index of the symbol in the Elf symbol table.

It returns a non-zero value on success, and 0 on error.

5.7.2 dwarf_add_arange_b()

```
Dwarf_Unsigned dwarf_add_arange_b(
        Dwarf_P_Debug dbg,
        Dwarf_Addr begin_address,
        Dwarf_Unsigned length,
        Dwarf_Unsigned symbol_index,
        Dwarf_Unsigned end_symbol_index,
        Dwarf_Addr     offset_from_end_symbol,
        Dwarf_Error *error)
```

The function dwarf_add_arange_b() adds another address range to be added to the section containing address range information, .debug_aranges.

If end_symbol_index is not zero we are using two symbols to create a length (must be DW_DLC_SYMBOLIC_RELOCATIONS to be useful)

> begin_address is the offset from the symbol specified by symbol_index . offset_from_end_symbol is the offset from the symbol specified by end_symbol_index. length is ignored. This begin-end pair will be show up in the relocation array returned by dwarf_get_relocation_info() as a dwarf_drt_first_of_length_pair and dwarf_drt_second_of_length_pair pair of relocation records. The consuming application

will turn that pair into something conceptually identical to

```
.word end_symbol + offset_from_end - \
 ( start_symbol + begin_address)
```

The reason offsets are allowed on the begin and end symbols is to allow the caller to re-use existing labels when the labels are available and the corresponding offset is known (economizing on the number of labels in use). The 'offset_from_end - begin_address' will actually be in the binary stream, not the relocation record, so the app processing the relocation array must read that stream value into (for example) net_offset and actually emit something like

```
.word end_symbol - start_symbol + net_offset
```

If `end_symbol_index` is zero we must be given a length (either `DW_DLC_STREAM_RELOCATIONS` or `DW_DLC_SYMBOLIC_RELOCATIONS`):

The relocatable start address of the range is specified by `begin_address`, and the length of the address range is specified by `length`. The relocatable symbol to be used to relocate the start of the address range is specified by `symbol_index`, which is normally the index of the symbol in the Elf symbol table. The `offset_from_end_symbol` is ignored.

It returns a non-zero value on success, and `0` on error.

5.8 Fast Access (pubnames) Operations

These functions operate on the .debug_pubnames section.

5.8.1 dwarf_add_pubname()

```
Dwarf_Unsigned dwarf_add_pubname(
        Dwarf_P_Debug dbg,
        Dwarf_P_Die die,
        char *pubname_name,
        Dwarf_Error *error)
```

The function `dwarf_add_pubname()` adds the pubname specified by `pubname_name` to the section containing pubnames, i.e.
.debug_pubnames. The `DIE` that represents the function being named is specified by `die`.

It returns a non-zero value on success, and `0` on error.

5.9 Fast Access (weak names) Operations

These functions operate on the .debug_weaknames section.

5.9.1 dwarf_add_weakname()

```
Dwarf_Unsigned dwarf_add_weakname(
        Dwarf_P_Debug dbg,
        Dwarf_P_Die die,
        char *weak_name,
        Dwarf_Error *error)
```

The function `dwarf_add_weakname()` adds the weak name specified by `weak_name` to the section containing weak names, i.e.
.debug_weaknames. The `DIE` that represents the function being named is specified by `die`.

It returns a non-zero value on success, and `0` on error.

5.10 Static Function Names Operations

The .debug_funcnames section contains the names of static function names defined in the object, and also the offsets of the `DIE`s that represent the definitions of the functions in the .debug_info section.

5.10.1 dwarf_add_funcname()

```
Dwarf_Unsigned dwarf_add_funcname(
        Dwarf_P_Debug dbg,
        Dwarf_P_Die die,
        char *func_name,
        Dwarf_Error *error)
```

The function `dwarf_add_funcname()` adds the name of a static function specified by `func_name` to the section containing the names of static functions defined in the object represented by `dbg`. The `DIE` that represents the definition of the function is specified by `die`.

It returns a non-zero value on success, and `0` on error.

5.11 File-scope User-defined Type Names Operations

The .debug_typenames section contains the names of file-scope user-defined types in the given object, and also the offsets of the `DIE`s that represent the definitions of the types in the .debug_info section.

5.11.1 dwarf_add_typename()

```
Dwarf_Unsigned dwarf_add_typename(
        Dwarf_P_Debug dbg,
        Dwarf_P_Die die,
        char *type_name,
        Dwarf_Error *error)
```

The function `dwarf_add_typename()` adds the name of a file-scope user-defined type specified by `type_name` to the section that contains the names of file-scope user-defined type. The object that this section belongs to is specified by `dbg`. The `DIE` that represents the definition of the type is specified by `die`.

It returns a non-zero value on success, and `0` on error.

5.12 File-scope Static Variable Names Operations

The .debug_varnames section contains the names of file-scope static variables in the given object, and also the offsets of the DIEs that represent the definition of the variables in the .debug_info section.

5.12.1 dwarf_add_varname()

```
Dwarf_Unsigned dwarf_add_varname(
        Dwarf_P_Debug dbg,
        Dwarf_P_Die die,
        char *var_name,
        Dwarf_Error *error)
```

The function dwarf_add_varname() adds the name of a file-scope static variable specified by var_name to the section that contains the names of file-scope static variables defined by the object represented by dbg. The DIE that represents the definition of the static variable is specified by die.

It returns a non-zero value on success, and 0 on error.

5.13 Macro Information Creation

All strings passed in by the caller are copied by these functions, so the space in which the caller provides the strings may be ephemeral (on the stack, or immediately reused or whatever) without this causing any difficulty.

5.13.1 dwarf_def_macro()

```
int dwarf_def_macro(Dwarf_P_Debug dbg,
        Dwarf_Unsigned lineno,
        char *name
        char *value,
        Dwarf_Error *error);
```

Adds a macro definition. The name argument should include the parentheses and parameter names if this is a function-like macro. Neither string should contain extraneous whitespace. dwarf_def_macro() adds the mandated space after the name and before the value in the output DWARF section(but does not change the strings pointed to by the arguments). If this is a definition before any files are read, lineno should be 0. Returns DW_DLV_ERROR and sets error if there is an error. Returns DW_DLV_OK if the call was successful.

5.13.2 dwarf_undef_macro()

```
int dwarf_undef_macro(Dwarf_P_Debug dbg,
        Dwarf_Unsigned lineno,
        char *name,
        Dwarf_Error *error);
```

Adds a macro un-definition note. If this is a definition before any files are read, lineno should be 0. Returns DW_DLV_ERROR and sets error if there is an error. Returns DW_DLV_OK if the call was successful.

5.13.3 dwarf_start_macro_file()

```
int dwarf_start_macro_file(Dwarf_P_Debug dbg,
        Dwarf_Unsigned lineno,
        Dwarf_Unsigned fileindex,
        Dwarf_Error *error);
```

`fileindex` is an index in the .debug_line header: the index of the file name. See the function `dwarf_add_file_decl()`. The `lineno` should be 0 if this file is the file of the compilation unit source itself (which, of course, is not a #include in any file). Returns `DW_DLV_ERROR` and sets `error` if there is an error. Returns `DW_DLV_OK` if the call was successful.

5.13.4 dwarf_end_macro_file()

```
int dwarf_end_macro_file(Dwarf_P_Debug dbg,
        Dwarf_Error *error);
```

Returns `DW_DLV_ERROR` and sets `error` if there is an error. Returns `DW_DLV_OK` if the call was successful.

5.13.5 dwarf_vendor_ext()

```
int dwarf_vendor_ext(Dwarf_P_Debug dbg,
    Dwarf_Unsigned constant,
    char *         string,
    Dwarf_Error*   error);
```

The meaning of the `constant` and the`string` in the macro info section are undefined by DWARF itself, but the string must be an ordinary null terminated string. This call is not an extension to DWARF. It simply enables storing macro information as specified in the DWARF document. Returns `DW_DLV_ERROR` and sets `error` if there is an error. Returns `DW_DLV_OK` if the call was successful.

5.14 Low Level (.debug_frame) operations

These functions operate on the .debug_frame section. Refer to `libdwarf.h` for the register names and register assignment mapping. Both of these are necessarily machine dependent.

5.14.1 dwarf_new_fde()

```
Dwarf_P_Fde dwarf_new_fde(
        Dwarf_P_Debug dbg,
        Dwarf_Error *error)
```

The function `dwarf_new_fde()` returns a new `Dwarf_P_Fde` descriptor that should be used to build a complete `FDE`. Subsequent calls to routines that build up the `FDE` should use the same `Dwarf_P_Fde` descriptor.

It returns a valid `Dwarf_P_Fde` descriptor on success, and `DW_DLV_BADADDR` on error.

5.14.2 dwarf_add_frame_cie()

```
Dwarf_Unsigned dwarf_add_frame_cie(
        Dwarf_P_Debug dbg,
        char *augmenter,
        Dwarf_Small code_align,
        Dwarf_Small data_align,
        Dwarf_Small ret_addr_reg,
        Dwarf_Ptr init_bytes,
        Dwarf_Unsigned init_bytes_len,
        Dwarf_Error *error);
```

The function `dwarf_add_frame_cie()` creates a `CIE`, and returns an index to it, that should be used to refer to this `CIE`. `CIE`s are used by `FDE`s to setup initial values for frames. The augmentation string for the `CIE` is specified by `augmenter`. The code alignment factor, data alignment factor, and the return address register for the `CIE` are specified by `code_align`, `data_align`, and `ret_addr_reg` respectively. `init_bytes` points to the bytes that represent the instructions for the `CIE` being created, and `init_bytes_len` specifies the number of bytes of instructions.

There is no convenient way to generate the `init_bytes` stream. One just has to calculate it by hand or separately generate something with the correct sequence and use dwarfdump -v and readelf (or objdump) and some kind of hex dumper to see the bytes. This is a serious inconvenience!

It returns an index to the `CIE` just created on success. On error it returns `DW_DLV_NOCOUNT`.

5.14.3 dwarf_add_frame_fde()

```
Dwarf_Unsigned dwarf_add_frame_fde(
        Dwarf_P_Debug dbg,
        Dwarf_P_Fde fde,
        Dwarf_P_Die die,
        Dwarf_Unsigned cie,
        Dwarf_Addr virt_addr,
        Dwarf_Unsigned  code_len,
        Dwarf_Unsigned sym_idx,
        Dwarf_Error* error)
```

The function `dwarf_add_frame_fde()` adds the `FDE` specified by `fde` to the list of `FDE`s for the object represented by the given `dbg`. `die` specifies the `DIE` that represents the function whose frame information is specified by the given `fde`. `cie` specifies the index of the `CIE` that should be used to setup the initial conditions for the given frame.

If the MIPS/IRIX specific DW_AT_MIPS_fde attribute is not needed in .debug_info pass in 0 as the `die` argument.

It returns an index to the given `fde`.

5.14.4 dwarf_add_frame_fde_b()

```
Dwarf_Unsigned dwarf_add_frame_fde_b(
        Dwarf_P_Debug dbg,
        Dwarf_P_Fde fde,
        Dwarf_P_Die die,
        Dwarf_Unsigned cie,
        Dwarf_Addr virt_addr,
        Dwarf_Unsigned  code_len,
        Dwarf_Unsigned sym_idx,
        Dwarf_Unsigned sym_idx_of_end,
        Dwarf_Addr     offset_from_end_sym,
        Dwarf_Error* error)
```

This function is like dwarf_add_frame_fde() except that dwarf_add_frame_fde_b() has new arguments to allow use with DW_DLC_SYMBOLIC_RELOCATIONS.

The function dwarf_add_frame_fde_b() adds the FDE specified by fde to the list of FDEs for the object represented by the given dbg.

die specifies the DIE that represents the function whose frame information is specified by the given fde. If the MIPS/IRIX specific DW_AT_MIPS_fde attribute is not needed in .debug_info pass in 0 as the die argument.

cie specifies the index of the CIE that should be used to setup the initial conditions for the given frame. virt_addr represents the relocatable address at which the code for the given function begins, and sym_idx gives the index of the relocatable symbol to be used to relocate this address (virt_addr that is). code_len specifies the size in bytes of the machine instructions for the given function.

If sym_idx_of_end is zero (may be DW_DLC_STREAM_RELOCATIONS or DW_DLC_SYMBOLIC_RELOCATIONS):

> virt_addr represents the relocatable address at which the code for the given function begins, and sym_idx gives the index of the relocatable symbol to be used to relocate this address (virt_addr that is). code_len specifies the size in bytes of the machine instructions for the given function. sym_idx_of_end and offset_from_end_sym are unused.

If sym_idx_of_end is non-zero (must be DW_DLC_SYMBOLIC_RELOCATIONS to be useful):

> virt_addr is the offset from the symbol specified by sym_idx . offset_from_end_sym is the offset from the symbol specified by sym_idx_of_end. code_len is ignored. This begin-end pair will be show up in the relocation array returned by dwarf_get_relocation_info() as a dwarf_drt_first_of_length_pair and dwarf_drt_second_of_length_pair pair of relocation records. The consuming application will turn that pair into something conceptually identical to
>
> ```
> .word end_symbol + begin - \
>  ( start_symbol + offset_from_end)
> ```
>
> The reason offsets are allowed on the begin and end symbols is to allow the caller to re-use existing labels when the labels are available and the corresponding offset is known (economizing on the number of labels in use). The 'offset_from_end - begin_address' will actually be in the binary stream, not the relocation record, so the app processing the relocation array must read that stream value into (for example) net_offset and actually emit something like

.word end_symbol - start_symbol + net_offset

It returns an index to the given fde.

On error, it returns DW_DLV_NOCOUNT.

5.14.5 dwarf_add_frame_info_b()

```
Dwarf_Unsigned dwarf_add_frame_info_b(
        Dwarf_P_Debug   dbg,
        Dwarf_P_Fde     fde,
        Dwarf_P_Die     die,
        Dwarf_Unsigned  cie,
        Dwarf_Addr      virt_addr,
        Dwarf_Unsigned  code_len,
        Dwarf_Unsigned  sym_idx,
        Dwarf_Unsigned  end_symbol_index,
        Dwarf_Addr      offset_from_end_symbol,
        Dwarf_Signed    offset_into_exception_tables,
        Dwarf_Unsigned  exception_table_symbol,
        Dwarf_Error*    error)
```

The function dwarf_add_frame_fde() adds the FDE specified by fde to the list of FDEs for the object represented by the given dbg.

This function refers to MIPS/IRIX specific exception tables and is not a function other targets need.

die specifies the DIE that represents the function whose frame information is specified by the given fde. If the MIPS/IRIX specific DW_AT_MIPS_fde attribute is not needed in .debug_info pass in 0 as the die argument.

cie specifies the index of the CIE that should be used to setup the initial conditions for the given frame.

offset_into_exception_tables specifies the MIPS/IRIX specific offset into .MIPS.eh_region elf section where the exception tables for this function begins. exception_table_symbol is also MIPS/IRIX specific and it specifies the index of the relocatable symbol to be used to relocate this offset.

If end_symbol_index is not zero we are using two symbols to create a length (must be DW_DLC_SYMBOLIC_RELOCATIONS to be useful)

> virt_addr is the offset from the symbol specified by sym_idx . offset_from_end_symbol is the offset from the symbol specified by end_symbol_index. code_len is ignored. This begin-end pair will be show up in the relocation array returned by dwarf_get_relocation_info() as a dwarf_drt_first_of_length_pair and dwarf_drt_second_of_length_pair pair of relocation records. The consuming application will turn that pair into something conceptually identical to
>
> .word end_symbol + offset_from_end_symbol - \
> (start_symbol + virt_addr)

The reason offsets are allowed on the begin and end symbols is to allow the caller to re-use existing labels when the labels are available and the corresponding offset is known (economizing on the number of labels in use). The 'offset_from_end - begin_address' will actually be in the binary stream, not the relocation record, so the app processing the relocation array must read that stream value into (for example) net_offset and actually emit something like

.word end_symbol - start_symbol + net_offset

If `end_symbol_index` is zero we must be given a code_len value (either `DW_DLC_STREAM_RELOCATIONS` or `DW_DLC_SYMBOLIC_RELOCATIONS`):

The relocatable start address of the range is specified by `virt_addr`, and the length of the address range is specified by `code_len`. The relocatable symbol to be used to relocate the start of the address range is specified by `symbol_index`, which is normally the index of the symbol in the Elf symbol table. The `offset_from_end_symbol` is ignored.

It returns an index to the given `fde`.

On error, it returns `DW_DLV_NOCOUNT`.

5.14.6 dwarf_add_frame_info()

```
Dwarf_Unsigned dwarf_add_frame_info(
        Dwarf_P_Debug dbg,
        Dwarf_P_Fde fde,
        Dwarf_P_Die die,
        Dwarf_Unsigned cie,
        Dwarf_Addr virt_addr,
        Dwarf_Unsigned  code_len,
        Dwarf_Unsigned sym_idx,
        Dwarf_Signed offset_into_exception_tables,
        Dwarf_Unsigned exception_table_symbol,
        Dwarf_Error* error)
```

The function `dwarf_add_frame_fde()` adds the `FDE` specified by `fde` to the list of `FDE`s for the object represented by the given `dbg`.

`die` specifies the `DIE` that represents the function whose frame information is specified by the given `fde`. If the MIPS/IRIX specific DW_AT_MIPS_fde attribute is not needed in .debug_info pass in 0 as the `die` argument.

`cie` specifies the index of the `CIE` that should be used to setup the initial conditions for the given frame. `virt_addr` represents the relocatable address at which the code for the given function begins, and `sym_idx` gives the index of the relocatable symbol to be used to relocate this address (`virt_addr` that is). `code_len` specifies the size in bytes of the machine instructions for the given function.

`offset_into_exception_tables` specifies the offset into `.MIPS.eh_region` elf section where the exception tables for this function begins. `exception_table_symbol` gives the index of the relocatable symbol to be used to relocate this offset. These arguments are MIPS/IRIX specific, pass in 0 for other targets.

It returns an index to the given fde.

5.14.7 dwarf_fde_cfa_offset()

```
Dwarf_P_Fde dwarf_fde_cfa_offset(
        Dwarf_P_Fde fde,
        Dwarf_Unsigned reg,
        Dwarf_Signed offset,
        Dwarf_Error *error)
```

The function dwarf_fde_cfa_offset() appends a DW_CFA_offset operation to the FDE, specified by fde, being constructed. The first operand of the DW_CFA_offset operation is specified by regP. The register specified should not exceed 6 bits. The second operand of the DW_CFA_offset operation is specified by offset.

It returns the given fde on success.

It returns DW_DLV_BADADDR on error.

5.14.8 dwarf_add_fde_inst()

```
Dwarf_P_Fde dwarf_add_fde_inst(
        Dwarf_P_Fde fde,
        Dwarf_Small op,
        Dwarf_Unsigned val1,
        Dwarf_Unsigned val2,
        Dwarf_Error *error)
```

The function dwarf_add_fde_inst() adds the operation specified by op to the FDE specified by fde. Up to two operands can be specified in val1, and val2. Based on the operand specified Libdwarf decides how many operands are meaningful for the operand. It also converts the operands to the appropriate datatypes (they are passed to dwarf_add_fde_inst as Dwarf_Unsigned).

It returns the given fde on success, and DW_DLV_BADADDR on error.

5.14.9 dwarf_insert_fde_inst_bytes()

```
int dwarf_insert_fde_inst_bytes(
        Dwarf_P_Debug dbg,
        Dwarf_P_Fde  fde,
        Dwarf_Unsigned len,
        Dwarf_Ptr ibytes,
        Dwarf_Error *error)
```

The function dwarf_insert_fde_inst_bytes() inserts the byte array (pointed at by ibytes and of length len) of frame instructions into the fde fde. It is incompatible with dwarf_add_fde_inst(), do not use both functions on any given Dwarf_P_Debug. At present it may only be called once on a given fde. The len bytes ibytes may be constructed in any way, but the assumption is they were copied from an object file such as is returned by the libdwarf consumer function dwarf_get_fde_instr_bytes().

It returns DW_DLV_OK on success, and DW_DLV_ERROR on error.

CONTENTS

A Producer Library Interface to DWARF

David Anderson

ABSTRACT

This document describes an interface to a library of functions to create DWARF debugging information entries and DWARF line number information. It does not make recommendations as to how the functions described in this document should be implemented nor does it suggest possible optimizations.

The document is oriented to creating DWARF version 2. Support for creating DWARF3 is intended but such support is not yet fully present. DWARF4 support is also intended.

Rev 1.46, 7 October 2016