

# **A Consumer Library Interface to DWARF**

*David Anderson*

## **1. INTRODUCTION**

This document describes an interface to *libdwarf*, a library of functions to provide access to DWARF debugging information records, DWARF line number information, DWARF address range and global names information, weak names information, DWARF frame description information, DWARF static function names, DWARF static variables, and DWARF type information.

The document has long mentioned the "Unix International Programming Languages Special Interest Group" (PLSIG), under whose auspices the DWARF committee was formed around 1991. "Unix International" was disbanded in the 1990s and no longer exists.

The DWARF committee published DWARF2 July 27, 1993.

In the mid 1990s this document and the library it describes (which the committee never endorsed, having decided not to endorse or approve any particular library interface) was made available on the internet by Silicon Graphics, Inc.

In 2005 the DWARF committee began an affiliation with FreeStandards.org. In 2007 FreeStandards.org merged with The Linux Foundation. The DWARF committee dropped its affiliation with FreeStandards.org in 2007 and established the dwarfstd.org website. See "<http://www.dwarfstd.org>" for current information on standardization activities and a copy of the standard.

### **1.1 Copyright**

Copyright 1993-2006 Silicon Graphics, Inc.

Copyright 2007-2019 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 document a library of functions to access DWARF debugging information. There is no effort made in this document to address the creation

of these records as those issues are addressed separately (see "A Producer Library Interface to DWARF").

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

### 1.3 Document History

A document was written about 1991 which had similar layout and interfaces. Written by people from Hal Corporation, That document described a library for reading DWARF1. The authors distributed paper copies to the committee with the clearly expressed intent to propose the document as a supported interface definition. The committee decided not to pursue a library definition.

SGI wrote the document you are now reading in 1993 with a similar layout and content and organization, but it was complete document rewrite with the intent to read DWARF2 (the DWARF version then in existence). The intent was (and is) to also cover future revisions of DWARF. All the function interfaces were changed in 1994 to uniformly return a simple integer success-code (see `DW_DLV_OK` etc), generally following the recommendations in the chapter titled "Candy Machine Interfaces" of "Writing Solid Code", a book by Steve Maguire (published by Microsoft Press).

### 1.4 Definitions

DWARF debugging information entries (DIEs) are the segments of information placed in the `.debug_*` sections by compilers, assemblers, and linkage editors that, in conjunction with line number entries, are necessary for symbolic source-level debugging. Refer to the latest "*DWARF Debugging Information Format*" from [www.dwarfstd.org](http://www.dwarfstd.org) for a more complete description of these entries.

This document adopts all the terms and definitions in "*DWARF Debugging Information Format*" versions 2,3,4, and 5. It originally focused on the implementation at Silicon Graphics, Inc., but now attempts to be more generally useful.

### 1.5 Overview

The remaining sections of this document describe the proposed 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.

We separate the functions into several categories to emphasize that not all consumers want to use all the functions. We call the categories Debugger, Internal-level, High-level, and Miscellaneous not because one is more important than another but as a way of

making the rather large set of function calls easier to understand.

Unless otherwise specified, all functions and structures should be taken as being designed for Debugger consumers.

The Debugger Interface of this library is intended to be used by debuggers. The interface is low-level (close to dwarf) but suppresses irrelevant detail. A debugger will want to absorb all of some sections at startup and will want to see little or nothing of some sections except at need. And even then will probably want to absorb only the information in a single compilation unit at a time. A debugger does not care about implementation details of the library.

The Internal-level Interface is for a DWARF prettyprinter and checker. A thorough prettyprinter will want to know all kinds of internal things (like actual FORM numbers and actual offsets) so it can check for appropriate structure in the DWARF data and print (on request) all that internal information for human users and libdwarf authors and compiler-writers. Calls in this interface provide data a debugger does not normally care about.

The High-level Interface is for higher level access (it is not really a high level interface!). Programs such as disassemblers will want to be able to display relevant information about functions and line numbers without having to invest too much effort in looking at DWARF.

The miscellaneous interface is just what is left over: the error handler functions.

The following is a brief mention of the changes in this libdwarf from the libdwarf draft for DWARF Version 1 and recent changes.

## 1.6 Items Changed

Added new functions for full `.debug_loclists` access: `dwarf_get_locdesc_entry_d()`, `dwarf_get_loclist_head_basics()`, `dwarf_get_loclist_head_kind()`, and `dwarf_loc_head_c_dealloc()`. For accessing certain DWARF5 new location operators (for example `DW_OP_const_type`) as well as all other operators we add `dwarf_get_location_op_value_d()`. Added functions allowing simple reporting of `.debug_loclists` without involving other sections: `dwarf_load_loclists()`, `dwarf_get_loclist_context_basics()`, `dwarf_get_loclist_lle()`, `dwarf_get_loclist_offset_index_value()`, and `dwarf_get_loclist_raw_entry_detail()`. (June 10, 2020);

Added new functions for full `.debug_rnglists` support and fixed issues with DWARF5 `.debug_addr_index` FORMs. New functions for general use: `dwarf_addr_form_is_indexed()`, `dwarf_get_rnglists_entry_fields()`, `dwarf_rnglists_get_rle_head()`, `dwarf_dealloc_rnglists_head()`. New functions for a complete listing of the `.debug_rnglists` section. `dwarf_load_rnglists()`, `dwarf_get_rnglist_offset_index_value()`, `dwarf_get_rnglist_context()`,

`dwarf_get_rnglist_head_basics()`, `dwarf_get_rnglist_context_basics()`, `dwarf_get_rnglist_rle()`. Also added new functions `dwarf_dealloc_die()`, `dwarf_dealloc_error()`, and `dwarf_dealloc_attribute()` to provide type-safe calls for deallocation of the specific data types. (May 20, 2020)

What was historically called 'length\_size' in libdwarf and dwarfdump is actually the size of an offset (4 or 8 in DWARF2,3,4 and 5). For readability all instances of 'length\_size' are being converted, as time permits, to 'offset\_size'. (May 1, 2020)

Added a new function `dwarf_set_de_alloc_flag()` which allows turning-off of libdwarf-internal allocation tracking to improve libdwarf performance a few percent (which only really matters with giant DWARF sections). The downside of turning off the flag is consumer code must do all the `dwarf_dealloc()` calls itself to avoid memory leaks. (March 14, 2020)

Corrected the documentation of `dwarf_diename`: It was never appropriate to use `dwarf_dealloc` on the string pointer returned but Up till now this document said such a call was required. (March 14, 2020)

Now we document here that if one uses `dwarf_init()` or `dwarf_init_b()` or `dwarf_init_path()` that the function `dwarf_get_elf()` cannot succeed as there is no longer any Elf pointer (from `libelf`) to return. (November 26, 2019)

New function `dwarf_gnu_debuglink()` allow callers to access fields that GNU compilers create and use to link an executable to its separate DWARF debugging content object file. (September 9, 2019, updated October 2019)

`dwarf_next_cu_header_d()` (and the other earlier versions of this) now allow a null in place of a pointer for `next_cu_offset`. `dwarf_hipcb()` now allows a null in place of the `return_form` and/or `return_class` arguments. Unless you know a sufficiently recent libdwarf is to be used it is not safe to pass those arguments as null pointers. This allowance of null is because we've become aware that the relevant NetBSD man pages on these functions incorrectly specified that null was allowed. (April 22, 2019)

The `dwarf_elf_init()` and `dwarf_elf_init_b()` are now deprecated as they require the use of `elf.h` and `libelf.h` and `libelf`. Use `dwarf_init_path()` or `dwarf_init_b()` instead. The new non-libelf reader code checks elf header values more thoroughly than `libelf` and detects corrupted Elf earlier and in more cases than `libelf`. Since the reports of elf corruption from libdwarf/dwarfdump are not detailed we suggest one use an object dumper to check the object file in question. Two useful object dumpers are GNU `readelf` (part of GNU binutils) and `readelfobj` (part of the `readelfobj` project on sourceforge.net). `readelfobj` uses essentially the same algorithms as libdwarf does and should report something meaningful. (April 20, 2019)

Added support for MacOS dSYM objects and PE object files as well as an initialization function allowing a path instead of a Posix/Unix fd or a `libelf Elf*`. (January 2019)

Added a libdwarf interface `dwarf_errmsg_by_number()` so that places in the code that can have errors but do not want the Dwarf\_Error complexities can report more details than just an error number. (December 19, 2018)

Now Mach-o dSYM files containing dwarf are readable by libdwarf and their DWARF dumped by dwarfdump. There are no new options or choices, libdwarf and dwarfdump notice which kind of object they are processing. New functions added to libdwarf.h.in: dwarf\_init\_path(), dwarf\_object\_detector\_path(), and dwarf\_object\_detector\_fd(). (October 24, 2018)

All references to Dwarf\_Frame\_Op3 have been removed as that struct was never created or available. The new function dwarf\_get\_fde\_info\_for\_reg3\_b() is documented. (May 12, 2018)

With DWARF5 it became harder to use dwarf\_srclines\_data\_b() as DWARF5 changed each line table header file table to zero-based indexing from one-based (and made the primary file index zero). So a new function dwarf\_srclines\_file\_indexes() returns values that make it easy to step through and call dwarf\_srclines\_data\_b() sensibly whether the line table is DWARF2,3,4, or 5. (March 23, 2018)

Added COMDAT support. Recent compilers generate COMDAT sections (for some DWARF information) routinely so this became important recently. The new libdwarf COMDAT support extends the groupnumber idea as suggested just below. (May 17, 2017)

Adding dwarf\_init\_b() and dwarf\_elf\_init\_b() and dwarf\_object\_init\_b() with a groupnumber option added. DWARF5 adds split-dwarf and we call original sections like .debug\_info group one and new sections like .debug\_info.dwo group two. It has not escaped our attention that this numbering can be extended to deal with Elf COMDAT section groups of DWARF information, though COMDAT groups are not currently supported. (April 02, 2017)

Adding support for DWARF5 .debug\_loc.dwo and split dwarf range tables. Added dwarf\_get\_offset\_size(). (November 08, 2015)

Adding support for reading DWARF5 line tables and GNU two-level line tables. The function dwarf\_srclines() still works but those using DWARF4 or DWARF5 are advised to switch to dwarf\_srclines\_b(). dwarf\_srclines() cannot handle skeleton line tables sensibly and a new interface was needed for two-level line tables so the new approach satisfies both. (October 5, 2015)

Adding support for Package Files (DWARF5) to enable access of address data using DW\_FORM\_addrx. See dwarf\_set\_tied\_dbg(). (September 13, 2015)

Adding some DWARF5 support and improved DWP Package File support, using dwarf\_next\_cu\_header\_d().

Added a note about dwarf\_errmsg(): the string pointer returned should be considered ephemeral, not a string which remains valid permanently. User code should print it or copy it before calling other libdwarf functions on the specific Dwarf\_Debug instance. (May 15, 2014)

Added a printf-callback so libdwarf will not actually print to stdout. Added dwarf\_highpc\_b() so return of a DWARF4 DW\_AT\_high\_pc of class constant can be returned properly. (August 15 2013)

Defined how the new operator DW\_OP\_GNU\_const\_type is handled. (January 26 2013)

Added dwarf\_loclist\_from\_expr\_b() function which adds arguments of the DWARF version (2 for DWARF2, etc) and the offset size to the dwarf\_loclist\_from\_expr\_a() function. Because the DW\_OP\_GNU\_implicit\_pointer opcode is defined differently for DWARF2 than for later versions. (November 2012)

Added new functions (some for libdwarf client code) and internal logic support for the DWARF4 .debug\_types section. The new functions are dwarf\_next\_cu\_header\_c(), dwarf\_siblingof\_b(), dwarf\_offdie\_b(), dwarf\_get\_cu\_die\_offset\_given\_cu\_header\_offset\_b(), dwarf\_get\_die\_infotypes\_flag(), dwarf\_get\_section\_max\_offsets\_b().

New functions and logic support additional detailed error reporting so that more compiler bugs can be reported sensibly by consumer code (as opposed to having libdwarf just assume things are ok and blindly continuing on with erroneous data). November 20, 2010

It seems impossible to default to both DW\_FRAME\_CFA\_COL and DW\_FRAME\_CFA\_COL3 in a single build of libdwarf, so the default is now unambiguously DW\_FRAME\_CFA\_COL3 unless the configure option --enable-oldframecol is specified at configure time. The function dwarf\_set\_frame\_cfa\_value() may be used to override the default : using that function gives consumer applications full control (its use is highly recommended). (January 17, 2010)

Added dwarf\_set\_reloc\_application() and the default automatic application of Elf 'rela' relocations to DWARF sections (such rela sections appear in .o files, not in executables or shared objects, in general). The dwarf\_set\_reloc\_application() routine lets a consumer turn off the automatic application of 'rela' relocations if desired (it is not clear why anyone would really want to do that, but possibly a consumer could write its own relocation application). An example application that traverses a set of DIEs was added to the new dwarfexample directory (not in this libdwarf directory, but in parallel to it). (July 10, 2009)

Added dwarf\_get\_TAG\_name() (and the FORM AT and so on) interface functions so applications can get the string of the TAG, Attribute, etc as needed. (June 2009)

Added dwarf\_get\_ranges\_a() and dwarf\_loclist\_from\_expr\_a() functions which add arguments allowing a correct address\_size when the address\_size varies by compilation unit (a varying address\_size is quite rare as of May 2009). (May 2009)

Added dwarf\_set\_frame\_same\_value(), and dwarf\_set\_frame\_undefined\_value() to complete the set of frame-information functions needed to allow an application get all frame information returned correctly (meaning that it can be correctly interpreted) for all ABIs. Documented dwarf\_set\_frame\_cfa\_value(). Corrected spelling to dwarf\_set\_frame\_rule\_initial\_value(). (April 2009).

Added support for various DWARF3 features, but primarily a new frame-information interface tailorable at run-time to more than a single ABI. See dwarf\_set\_frame\_rule\_initial\_value(), dwarf\_set\_frame\_rule\_table\_size(), dwarf\_set\_frame\_cfa\_value(). See also dwarf\_get\_fde\_info\_for\_reg3() and

dwarf\_get\_fde\_info\_for\_cfa\_reg3(). (April 2006)

Added support for DWARF3 .debug\_pubtypes section. Corrected various leaks (revising dealloc() calls, adding new functions) and corrected dwarf\_formstring() documentation.

Added dwarf\_srclines\_dealloc() as the previous deallocation method documented for data returned by dwarf\_srclines() was incapable of freeing all the allocated storage (14 July 2005).

dwarf\_nextglob(), dwarf\_globname(), and dwarf\_globdie() were all changed to operate on the items in the .debug\_pubnames section.

All functions were modified to return solely an error code. Data is returned through pointer arguments. This makes writing safe and correct library-using-code far easier. For justification for this approach, see the chapter titled "Candy Machine Interfaces" in the book "Writing Solid Code" by Steve Maguire.

## 1.7 Items Removed

Dwarf\_Type was removed since types are no longer special.

dwarf\_typeof() was removed since types are no longer special.

Dwarf\_Ellist was removed since element lists no longer are a special format.

Dwarf\_Bounds was removed since bounds have been generalized.

dwarf\_nextdie() was replaced by dwarf\_next\_cu\_header() to reflect the real way DWARF is organized. The dwarf\_nextdie() was only useful for getting to compilation unit beginnings, so it does not seem harmful to remove it in favor of a more direct function.

dwarf\_childcnt() is removed on grounds that no good use was apparent.

dwarf\_prevline() and dwarf\_nextline() were removed on grounds this is better left to a debugger to do. Similarly, dwarf\_dieline() was removed.

dwarf\_is1stline() was removed as it was not meaningful for the revised DWARF line operations.

Any libdwarf implementation might well decide to support all the removed functionality and to retain the DWARF Version 1 meanings of that functionality. This would be difficult because the original libdwarf draft specification used traditional C library interfaces which confuse the values returned by successful calls with exceptional conditions like failures and 'no more data' indications.

## 1.8 Revision History

May 2020	Adding support for DWARF5 sections .debug_rnglists and .debug_loclists.
----------	---

March 2020	Added <code>dwarf_set_de_alloc_flag()</code> so consumers get a little better performance from <code>libdwarf</code> . At a price. See the description a bit later here.
January 2019	Added support for reading DWARF in PE object files.
October 2018	Added support for reading MacOS dSYM object files.
2017	Added support for nearly all of DWARF5.
July 2014	Added support for the <code>.gdb_index</code> section and started support for the <code>.debug_cu_index</code> and <code>.debug_tu_index</code> sections.
October 2011	DWARF4 support for reading <code>.debug_types</code> added.
March 93	Work on DWARF2 SGI draft begins
June 94	The function returns are changed to return an error/success code only.
April 2006:	Support for DWARF3 consumer operations is close to completion.
November 2010:	Added various new functions and improved error checking.
March 2017:	Adding support for DWARF5 split dwarf.

## 2. Types 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 and can be placed in three categories:

- Scalar types : The scalar types defined in *libdwarf.h* are defined primarily for notational convenience and identification. Depending on the individual definition, they are interpreted as a value, a pointer, or as a flag.
- Aggregate types : Some values can not be represented by a single scalar type; they must be represented by a collection of, or as a union of, scalar and/or aggregate types.
- Opaque types : The complete definition of these types is intentionally omitted; their use is as handles for query operations, which will yield either an instance of another opaque type to be used in another query, or an instance of a scalar or aggregate type, which is the actual result.

### 2.2 Scalar Types

The following are the defined by *libdwarf.h*:



```
typedef int Dwarf_Bool;
typedef unsigned long long Dwarf_Off;
typedef unsigned long long Dwarf_Unsigned;
typedef unsigned short Dwarf_Half;
typedef unsigned char Dwarf_Small;
typedef signed long long Dwarf_Signed;
typedef unsigned long long Dwarf_Addr;
typedef void *Dwarf_Ptr;
typedef void (*Dwarf_Handler) (Dwarf_Error error, Dwarf_Ptr errarg);
```

Dwarf\_Ptr is an address for use by the host program calling the library, not for representing pc-values/addresses within the target object file. Dwarf\_Addr is for pc-values within the target object file. The sample scalar type assignments above are for a *libdwarf.h* that can read and write 32-bit or 64-bit binaries on a 32-bit or 64-bit host machine. The types must be defined appropriately for each implementation of libdwarf. A description of these scalar types in the SGI/MIPS environment is given in Figure 1.

NAME	SIZE	ALIGNMENT	PURPOSE
Dwarf_Bool	4	4	Boolean states
Dwarf_Off	8	8	Unsigned file offset
Dwarf_Unsigned	8	8	Unsigned large integer
Dwarf_Half	2	2	Unsigned medium integer
Dwarf_Small	1	1	Unsigned small integer
Dwarf_Signed	8	8	Signed large integer
Dwarf_Addr	8	8	Program address (target program)
Dwarf_Ptr	4 8	4 8	Dwarf section pointer (host program)
Dwarf_Handler	4 8	4 8	Pointer to error handler function

**Figure 1.** Scalar Types

## 2.3 Aggregate Types

The following aggregate types are defined by *libdwarf.h*: Dwarf\_Loc, Dwarf\_Locdesc, Dwarf\_Block, Dwarf\_Frame\_Op, Dwarf\_Regtable, Dwarf\_Regtable3. While most of libdwarf acts on or returns simple values or opaque pointer types, this small set of structures seems useful. Yet, at the same time, these public structures are inflexible as any change in format or content breaks binary (and possibly source in some cases) compatibility.

### 2.3.1 Location Record

The `Dwarf_Loc` type identifies a single atom of a location description or a location expression. This is obsolete and should not be used, though it works adequately for DWARF2.

```
typedef struct {
    Dwarf_Small      lr_atom;
    Dwarf_Unsigned    lr_number;
    Dwarf_Unsigned    lr_number2;
    Dwarf_Unsigned    lr_offset;
} Dwarf_Loc;
```

The `lr_atom` identifies the atom corresponding to the `DW_OP_*` definition in *dwarf.h* and it represents the operation to be performed in order to locate the item in question.

The `lr_number` field is the operand to be used in the calculation specified by the `lr_atom` field; not all atoms use this field. Some atom operations imply signed numbers so it is necessary to cast this to a `Dwarf_Signed` type for those operations.

The `lr_number2` field is the second operand specified by the `lr_atom` field; only `DW_OP_BREGX` has this field. Some atom operations imply signed numbers so it may be necessary to cast this to a `Dwarf_Signed` type for those operations.

For a `DW_OP_implicit_value` operator the `lr_number2` field is a pointer to the bytes of the value. The field pointed to is `lr_number` bytes long. There is no explicit terminator. Do not attempt to free the bytes which `lr_number2` points at and do not alter those bytes. The pointer value remains valid till the open `Dwarf_Debug` is closed. This is a rather ugly use of a host integer to hold a pointer. You will normally have to do a 'cast' operation to use the value.

For a `DW_OP_GNU_const_type` operator the `lr_number2` field is a pointer to a block with an initial unsigned byte giving the number of bytes following, followed immediately that number of const value bytes. There is no explicit terminator. Do not attempt to free the bytes which `lr_number2` points at and do not alter those bytes. The pointer value remains valid till the open `Dwarf_Debug` is closed. This is a rather ugly use of a host integer to hold a pointer. You will normally have to do a 'cast' operation to use the value.

The `lr_offset` field is the byte offset (within the block the location record came from) of the atom specified by the `lr_atom` field. This is set on all atoms. This is useful for operations `DW_OP_SKIP` and `DW_OP_BRA`.

### 2.3.2 Location Description

This is obsolete and should not be used, though it works ok for DWARF2.. The `Dwarf_Locdesc` type represents an ordered list of `Dwarf_Loc` records used in the

calculation to locate an item. Note that in many cases, the location can only be calculated at runtime of the associated program.

```
typedef struct {
    Dwarf_Addr      ld_lopc;
    Dwarf_Addr      ld_hipc;
    Dwarf_Unsigned   ld_cents;
    Dwarf_Loc*       ld_s;
} Dwarf_Locdesc;
```

The `ld_lopc` and `ld_hipc` fields provide an address range for which this location descriptor is valid. Both of these fields are set to *zero* if the location descriptor is valid throughout the scope of the item it is associated with. These addresses are virtual memory addresses, not offsets-from-something. The virtual memory addresses do not account for dso movement (none of the pc values from libdwarf do that, it is up to the consumer to do that).

The `ld_cents` field contains a count of the number of `Dwarf_Loc` entries pointed to by the `ld_s` field.

The `ld_s` field points to an array of `Dwarf_Loc` records.

### 2.3.3 Data Block

This is obsolete and should not be used, though it works ok for DWARF2. The `Dwarf_Block` type is used to contain the value of an attribute whose form is either `DW_FORM_block1`, `DW_FORM_block2`, `DW_FORM_block4`, `DW_FORM_block8`, or `DW_FORM_block`. Its intended use is to deliver the value for an attribute of any of these forms.

```
typedef struct {
    Dwarf_Unsigned   bl_len;
    Dwarf_Ptr        bl_data;
    Dwarf_Small      bl_from_loclist;
    Dwarf_Unsigned   bl_section_offset;
} Dwarf_Block;
```

The `bl_len` field contains the length in bytes of the data pointed to by the `bl_data` field.

The `bl_data` field contains a pointer to the uninterpreted data. Since we use a `Dwarf_Ptr` here one must copy the pointer to some other type (typically an unsigned `char *`) so one can add increments to index through the data. The data pointed to by `bl_data` is not necessarily at any useful alignment.

### 2.3.4 Frame Operation Codes: DWARF 2

This interface is adequate for DWARF2 but not entirely suitable for DWARF3 or later. A new (functional) interface is needed. This DWARF2 interface is not sufficient but at present is the only available interface.

See also the section "Low Level Frame Operations" below.

The DWARF2 `Dwarf_Frame_Op` type is used to contain the data of a single instruction of an instruction-sequence of low-level information from the section containing frame information. This is ordinarily used by Internal-level Consumers trying to print everything in detail.

```
typedef struct {
    Dwarf_Small  fp_base_op;
    Dwarf_Small  fp_extended_op;
    Dwarf_Half   fp_register;
    Dwarf_Signed fp_offset;
    Dwarf_Offset fp_instr_offset;
} Dwarf_Frame_Op;
```

`fp_base_op` is the 2-bit basic op code. `fp_extended_op` is the 6-bit extended opcode (if `fp_base_op` indicated there was an extended op code) and is zero otherwise.

`fp_register` is any (or the first) register value as defined in the `Call` frame instruction encodings in the dwarf document (in DWARF3 see Figure 40, in DWARF5 see table 7.29). If not used with the operation it is 0.

`fp_offset` is the address, delta, offset, or second register as defined in the `Call` frame instruction encodings documentation. If this is an address then the value should be cast to (`Dwarf_Addr`) before being used.

In any implementation this field *must* be as large as the largest of `Dwarf_Ptr`, `Dwarf_Signed`, and `Dwarf_Addr` for this to work properly. If not used with the op it is 0. If the `fp_extended_op` is `DW_CFA_def_cfa` or `DW_CFA_val_expression` or `DW_CFA_expression` then `fp_offset` is a pointer to an expression block in the in-memory copy of the frame section.

`fp_instr_offset` is the `byte_offset` (within the instruction stream of the frame instructions) of this operation. It starts at 0 for a given frame descriptor.

### 2.3.5 Frame Regtable: DWARF 2

This interface is adequate for DWARF2 and MIPS but not for DWARF3 or later. A separate and preferred interface usable for DWARF3 and for DWARF2 is described below. See also the section "Low Level Frame Operations" below.

The Dwarf\_Regtable type is used to contain the register-restore information for all registers at a given PC value. Normally used by debuggers. If you wish to default to this interface and to the use of DW\_FRAME\_CFA\_COL, specify --enable\_oldframecol at libdwarf configure time. Or add a call dwarf\_set\_frame\_cfa\_value(dbg,DW\_FRAME\_CFA\_COL) after your dwarf\_init\_b() call, this call replaces the default libdwarf-compile-time value with DW\_FRAME\_CFA\_COL.

```
/* DW_REG_TABLE_SIZE must reflect the number of registers
*(DW_FRAME_LAST_REG_NUM) as defined in dwarf.h
*/
#define DW_REG_TABLE_SIZE <fill in size here, 66 for MIPS/IRIX>
typedef struct {
    struct {
        Dwarf_Small    dw_offset_relevant;
        Dwarf_Half     dw_regnum;
        Dwarf_Addr     dw_offset;
    } rules[DW_REG_TABLE_SIZE];
} Dwarf_Regtable;
```

The array is indexed by register number. The field values for each index are described next. For clarity we describe the field values for index rules[M] (M being any legal array element index).

dw\_offset\_relevant is non-zero to indicate the dw\_offset field is meaningful. If zero then the dw\_offset is zero and should be ignored.

dw\_regnum is the register number applicable. If dw\_offset\_relevant is zero, then this is the register number of the register containing the value for register M. If dw\_offset\_relevant is non-zero, then this is the register number of the register to use as a base (M may be DW\_FRAME\_CFA\_COL, for example) and the dw\_offset value applies. The value of register M is therefore the value of register dw\_regnum.

dw\_offset should be ignored if dw\_offset\_relevant is zero. If dw\_offset\_relevant is non-zero, then the consumer code should add the value to the value of the register dw\_regnum to produce the value.

### 2.3.6 Frame Operation Codes: DWARF 3 (for DWARF2 and later )

This interface was intended to be adequate for DWARF3 and for DWARF2 (and DWARF4) but was never implemented.

### 2.3.7 Frame Regtable: DWARF 3 (for DWARF2 and later)

This interface is adequate for DWARF2 and later versions. It is new in libdwarf as of April 2006. The default configure of libdwarf inserts DW\_FRAME\_CFA\_COL3 as the default CFA column. Or add a call dwarf\_set\_frame\_cfa\_value(dbg,DW\_FRAME\_CFA\_COL3) after your dwarf\_init\_b() call, this call replaces the default libdwarf-compile-time value with DW\_FRAME\_CFA\_COL3.

The Dwarf\_Regtable3 type is used to contain the register-restore information for all registers at a given PC value. Normally used by debuggers.

```
typedef struct Dwarf_Regtable_Entry3_s {
    Dwarf_Small      dw_offset_relevant;
    Dwarf_Small      dw_value_type;
    Dwarf_Half       dw_regnum;
    Dwarf_Unsigned    dw_offset_or_block_len;
    Dwarf_Ptr        dw_block_ptr;
} Dwarf_Regtable_Entry3;

typedef struct Dwarf_Regtable3_s {
    struct Dwarf_Regtable_Entry3_s  rt3_cfa_rule;
    Dwarf_Half                      rt3_reg_table_size;
    struct Dwarf_Regtable_Entry3_s * rt3_rules;
} Dwarf_Regtable3;
```

The array is indexed by register number. The field values for each index are described next. For clarity we describe the field values for index rules[M] (M being any legal array element index). (DW\_FRAME\_CFA\_COL3 DW\_FRAME\_SAME\_VAL, DW\_FRAME\_UNDEFINED\_VAL are not legal array indexes, nor is any index < 0 or >= rt3\_reg\_table\_size); The caller of routines using this struct must create data space for rt3\_reg\_table\_size entries of struct Dwarf\_Regtable\_Entry3\_s and arrange that rt3\_rules points to that space and that rt3\_reg\_table\_size is set correctly. The caller need not (but may) initialize the contents of the rt3\_cfa\_rule or the rt3\_rules array. The following applies to each rt3\_rules rule M:

dw\_regnum is the register number applicable. If dw\_regnum is DW\_FRAME\_UNDEFINED\_VAL, then the register I has undefined value. If dw\_regnum is DW\_FRAME\_SAME\_VAL, then the register I has the same value as in the previous frame.

If dw\_regnum is neither of these two, then the following apply:

dw\_value\_type determines the meaning of the other fields. It is one of DW\_EXPR\_OFFSET (0), DW\_EXPR\_VAL\_OFFSET(1), DW\_EXPR\_EXPRESSION(2) or DW\_EXPR\_VAL\_EXPRESSION(3).

If dw\_value\_type is DW\_EXPR\_OFFSET (0) then this is as in DWARF2

and the offset(N) rule or the register(R) rule of the DWARF3 and DWARF2 document applies. The value is either:

If `dw_offset_relevant` is non-zero, then `dw_regnum` is effectively ignored but must be identical to `DW_FRAME_CFA_COL3` (and the `dw_offset` value applies. The value of register M is therefore the value of CFA plus the value of `dw_offset`. The result of the calculation is the address in memory where the value of register M resides. This is the offset(N) rule of the DWARF2 and DWARF3 documents.

`dw_offset_relevant` is zero it indicates the `dw_offset` field is not meaningful. The value of register M is the value currently in register `dw_regnum` (the value `DW_FRAME_CFA_COL3` must not appear, only real registers). This is the register(R) rule of the DWARF3 spec.

If `dw_value_type` is `DW_EXPR_OFFSET` (1) then this is the the `val_offset(N)` rule of the DWARF3 spec applies. The calculation is identical to that of `DW_EXPR_OFFSET` (0) but the value is interpreted as the value of register M (rather than the address where register M's value is stored).

If `dw_value_type` is `DW_EXPR_EXPRESSION` (2) then this is the the `expression(E)` rule of the DWARF3 document.

`dw_offset_or_block_len` is the length in bytes of the in-memory block pointed at by `dw_block_ptr`. `dw_block_ptr` is a DWARF expression. Evaluate that expression and the result is the address where the previous value of register M is found.

If `dw_value_type` is `DW_EXPR_VAL_EXPRESSION` (3) then this is the the `val_expression(E)` rule of the DWARF3 spec.

`dw_offset_or_block_len` is the length in bytes of the in-memory block pointed at by `dw_block_ptr`. `dw_block_ptr` is a DWARF expression. Evaluate that expression and the result is the previous value of register M.

The rule `rt3_cfa_rule` is the current value of the CFA. It is interpreted exactly like any register M rule (as described just above) except that `dw_regnum` cannot be `CW_FRAME_CFA_REG3` or `DW_FRAME_UNDEFINED_VAL` or `DW_FRAME_SAME_VAL` but must be a real register number.

### 2.3.8 Macro Details Record

The `Dwarf_Macro_Details` type gives information about a single entry in the `.debug.macro` section (DWARF2, DWARF3, and DWARF4). It is not useful for

DWARF 5 .debug\_macro section data.

```
struct Dwarf_Macro_Details_s {
    Dwarf_Off      dmd_offset;
    Dwarf_Small    dmd_type;
    Dwarf_Signed   dmd_lineno;
    Dwarf_Signed   dmd_fileindex;
    char *         dmd_macro;
};
typedef struct Dwarf_Macro_Details_s Dwarf_Macro_Details;
```

`dmd_offset` is the byte offset, within the .debug\_macro section, of this macro information.

`dmd_type` is the type code of this macro info entry (or 0, the type code indicating that this is the end of macro information entries for a compilation unit. See DW\_MACRO\_define, etc in the DWARF document.

`dmd_lineno` is the line number where this entry was found, or 0 if there is no applicable line number.

`dmd_fileindex` is the file index of the file involved. This is only guaranteed meaningful on a DW\_MACRO\_start\_file `dmd_type`. Set to -1 if unknown (see the functional interface for more details).

`dmd_macro` is the applicable string. For a DW\_MACRO\_define this is the macro name and value. For a DW\_MACRO\_undef, or this is the macro name. For a DW\_MACRO\_vendor\_ext this is the vendor-defined string value. For other `dmd_types` this is 0.

## 2.4 Opaque Types

The opaque types declared in *libdwarf.h* are used as descriptors for queries against DWARF information stored in various debugging sections. Each time an instance of an opaque type is returned as a result of a *libdwarf* operation (Dwarf\_Debug excepted), it should be freed, using `dwarf_dealloc()` when it is no longer of use (read the following documentation for details, as in at least one case there is a special routine provided for deallocation and `dwarf_dealloc()` is not directly called: see `dwarf_srclines()`). Some functions return a number of instances of an opaque type in a block, by means of a pointer to the block and a count of the number of opaque descriptors in the block: see the function description for deallocation rules for such functions. The list of opaque types defined in *libdwarf.h* that are pertinent to the Consumer Library, and their intended use is described below. This is not a full list of the opaque types, see *libdwarf.h* for the full list.

```
typedef struct Dwarf_Debug_s* Dwarf_Debug;
```

An instance of the Dwarf\_Debug type is created as a result of a successful call to `dwarf_init_b()`, or `dwarf_elf_init_b()`, and is used as a descriptor for



subsequent access to most `libdwarf` functions on that object. The storage pointed to by this descriptor should be not be freed, using the `dwarf_dealloc()` function. Instead free it with `dwarf_finish()`.

```
typedef struct Dwarf_Die_s* Dwarf_Die;
```

An instance of a `Dwarf_Die` type is returned from a successful call to the `dwarf_siblingof()`, `dwarf_child`, or `dwarf_offdie_b()` function, and is used as a descriptor for queries about information related to that DIE. The storage pointed to by this descriptor should be freed, using `dwarf_dealloc()` with the allocation type `DW_DLA_DIE` when no longer needed, or, preferably, call `dwarf_dealloc_die()` instead.

```
typedef struct Dwarf_Line_s* Dwarf_Line;
```

Instances of `Dwarf_Line` type are returned from a successful call to the `dwarf_srclines()` function, and are used as descriptors for queries about source lines. The storage pointed to by these descriptors should be individually freed, using `dwarf_dealloc()` with the allocation type `DW_DLA_LINE` when no longer needed.

```
typedef struct Dwarf_Global_s* Dwarf_Global;
```

Instances of `Dwarf_Global` type are returned from a successful call to the `dwarf_get_globals()` function, and are used as descriptors for queries about global names (pubnames).

```
typedef struct Dwarf_Weak_s* Dwarf_Weak;
```

Instances of `Dwarf_Weak` type are returned from a successful call to the SGI-specific `dwarf_get_weakes()` function, and are used as descriptors for queries about weak names. The storage pointed to by these descriptors should be individually freed, using `dwarf_dealloc()` with the allocation type `DW_DLA_WEAK_CONTEXT` (or `DW_DLA_WEAK`, an older name, supported for compatibility) when no longer needed.

```
typedef struct Dwarf_Func_s* Dwarf_Func;
```

Instances of `Dwarf_Func` type are returned from a successful call to the SGI-specific `dwarf_get_funcs()` function, and are used as descriptors for queries about static function names.

```
typedef struct Dwarf_Type_s* Dwarf_Type;
```

Instances of `Dwarf_Type` type are returned from a successful call to the SGI-specific `dwarf_get_types()` function, and are used as descriptors for queries about user defined types.

```
typedef struct Dwarf_Var_s* Dwarf_Var;
```

Instances of `Dwarf_Var` type are returned from a successful call to the SGI-specific `dwarf_get_vars()` function, and are used as descriptors for queries about static variables.

```
typedef struct Dwarf_Error_s* Dwarf_Error;
```

This descriptor points to a structure that provides detailed information about errors detected by `libdwarf`. Users typically provide a location for `libdwarf` to store this descriptor for the user to obtain more information about the error. The storage pointed to by this descriptor should be freed, using `dwarf_dealloc()` with the allocation type `DW_DLA_ERROR` when no longer needed or, preferably, call `dwarf_dealloc_error()` instead.

```
typedef struct Dwarf_Attribute_s* Dwarf_Attribute;
```

Instances of `Dwarf_Attribute` type are returned from a successful call to the `dwarf_attrlist()`, or `dwarf_attr()` functions, and are used as descriptors for queries about attribute values. The storage pointed to by this descriptor should be individually freed, using `dwarf_dealloc()` with the allocation type `DW_DLA_ATTR` when no longer needed, or call `dwarf_dealloc_attribute()` instead.

```
typedef struct Dwarf_Abbrev_s* Dwarf_Abbrev;
```

An instance of a `Dwarf_Abbrev` type is returned from a successful call to `dwarf_get_abbrev()`, and is used as a descriptor for queries about abbreviations in the `.debug_abbrev` section. The storage pointed to by this descriptor should be freed, using `dwarf_dealloc()` with the allocation type `DW_DLA_ABBREV` when no longer needed.

```
typedef struct Dwarf_Fde_s* Dwarf_Fde;
```

Instances of `Dwarf_Fde` type are returned from a successful call to the `dwarf_get_fde_list()`, `dwarf_get_fde_for_die()`, or `dwarf_get_fde_at_pc()` functions, and are used as descriptors for queries about frames descriptors.

```
typedef struct Dwarf_Cie_s* Dwarf_Cie;
```

Instances of `Dwarf_Cie` type are returned from a successful call to the `dwarf_get_fde_list()` function, and are used as descriptors for queries about information that is common to several frames.

```
typedef struct Dwarf_Arange_s* Dwarf_Arange;
```

Instances of `Dwarf_Arange` type are returned from successful calls to the

`dwarf_get_aranges()`, or `dwarf_get_arange()` functions, and are used as descriptors for queries about address ranges. The storage pointed to by this descriptor should be individually freed, using `dwarf_dealloc()` with the allocation type `DW_DLA_ARANGE` when no longer needed.

```
typedef struct Dwarf_Gdbindex_s* Dwarf_Gdbindex;
```

Instances of `Dwarf_Gdbindex` type are returned from successful calls to the `dwarf_gdbindex_header()` function and are used to extract information from a `.gdb_index` section. This section is a gcc/gdb extension and is designed to allow a debugger fast access to data in `.debug_info`. The storage pointed to by this descriptor should be freed using a call to `dwarf_gdbindex_free()` with a valid `Dwarf_Gdbindex` pointer as the argument.

```
typedef struct Dwarf_Xu_Index_Header_s* Dwarf_Xu_Index_header;
```

Instances of `Dwarf_Xu_Index_Header_s` type are returned from successful calls to the `dwarf_get_xu_index_header()` function and are used to extract information from a `.debug_cu_index` or `.debug_tu_index` section. These sections are used to make possible access to `.dwo` sections gathered into a `.dwp` object as part of the DebugFission (ie Split Dwarf) project allowing separation of an executable from most of its DWARF debugging information. As of May 2015 these sections are accepted into DWARF5 but the standard has not been released. The storage pointed to by this descriptor should be freed using a call to `dwarf_xh_header_free()` with a valid `Dwarf_XuIndexHeader` pointer as the argument.

```
typedef struct Dwarf_Line_Context_s * Dwarf_Line_Context;
```

`dwarf_srclines_b()` returns a `Dwarf_Line_Context` through an argument and the new structure pointer lets us access line header information conveniently.

```
typedef struct Dwarf_Locdesc_c_s * Dwarf_Locdesc_c;
```

```
typedef struct Dwarf_Loc_Head_c_s * Dwarf_Loc_Head_c;
```

`Dwarf_Loc*` are involved in the DWARF5 interfaces to location lists. The new interfaces are all functional and contents of the above types are not exposed.

```
typedef struct Dwarf_Macro_Context_s * Dwarf_Macro_Context;
```

`dwarf_get_macro_context()` and `dwarf_get_macro_context_by_offset()` return a `Dwarf_Line_Context` through an argument and the new structure pointer lets us access macro data from the `.debug_macro` section.

```
typedef struct Dwarf_Dsc_Head_s * Dwarf_Dsc_Head;
```

`dwarf_discr_list()` returns a `Dwarf_Dsc_Head` through an argument and the new structure pointer lets us access macro data from a `DW_AT_discr_list` attribute.

### 3. UTF-8 strings

*libdwarf* is defined, at various points, to return string pointers or to copy strings into string areas you define. DWARF allows the use of `DW_AT_use_UTF8` (DWARF3 and later) `DW_ATE_UTF` (DWARF4 and later) to specify that the strings returned are actually in UTF-8 format. What this means is that if UTF-8 is specified on a particular object it is up to callers that wish to print all the characters properly to use language-appropriate functions to print Unicode strings appropriately. All ASCII characters in the strings will print properly whether printed as wide characters or not. The methods to convert UTF-8 strings so they will print correctly for all such strings is beyond the scope of this document.

If UTF-8 is not specified then one is probably safe in assuming the strings are iso\_8859-15 and normal C `printf()` will work fine..

In either case one should be wary of corrupted (accidentally or intentionally) strings with ASCII control characters in the text. Such can cause bad effects if simply printed to a device (such as a terminal).

### 4. Error Handling

The method for detection and disposition of error conditions that arise during access of debugging information via *libdwarf* is consistent across all *libdwarf* functions that are capable of producing an error. This section describes the method used by *libdwarf* in notifying client programs of error conditions.

Most functions within *libdwarf* accept as an argument a pointer to a `Dwarf_Error` descriptor where a `Dwarf_Error` descriptor is stored if an error is detected by the function. Routines in the client program that provide this argument can query the `Dwarf_Error` descriptor to determine the nature of the error and perform appropriate processing. The intent is that clients do the appropriate processing immediately on encountering an error and then the client calls `dwarf_dealloc_error` to free the `Dwarf_Error` descriptor (at which point the client should zero that descriptor as the non-zero value is stale).

For example we think the following is appropriate as a general outline. See `dwarfdump` source for many examples of both of the following incomplete examples.

```
int example_codea(Dwarf_Debug dbg, Dwarf_Die indie,
int is_info, Dwarf_Die *sibdie, Dwarf_Error *err)
{
    int res = 0;
    const char *secname = 0;
    res = dwarf_siblingof_b(dbg, indie, is_info, sibdie,
        err);
    if (res == DW_DLV_ERROR) {
        return res; /*let higher level decide what to do
            and it will eventually need to do
            dwarf_dealloc_error() appropriately */
    } else if (res == DW_DLV_NO_ENTRY) {
        /* No sibdie created. Nothing done. */
        return res;
    }
    /* sibdie created, caller will have to eventually
        do dwarf_dealloc_die() appropriately. */
    ...
    return DW_DLV_OK;
}
```

In a case where it is ok to suppress the error as being unimportant, this is an outline, not a useful function.

```
int example_codeb(Dwarf_Debug dbg, const char **sec_name,
int is_info)
{
    Dwarf_Error e = 0;
    int res = 0;
    res = dwarf_get_die_section_name(dbg, is_info,
        sec_name, &e);
    if (res == DW_DLV_ERROR) {
        dwarf_dealloc_error(e);
        e = 0;
        return res; /*let higher level decide what to do,
            Nothing allocated in the call still exists. */
    } if (res == DW_DLV_NO_ENTRY) {
        ....
    }
    ...
}
```

In the rare case where the malloc arena is exhausted when trying to create a Dwarf\_Error descriptor a pointer to a statically allocated descriptor will be returned. This static descriptor is new in December 2014. A call to dwarf\_dealloc() to free the

statically allocated descriptor is harmless (it sets the error value in the descriptor to `DW_DLE_FAILSAFE_ERRVAL`). The possible conflation of errors when the arena is exhausted (and a dwarf\_error descriptor is saved past the next reader call in any thread) is considered better than having *libdwarf* call `abort()` (as earlier *libdwarf* did).

We strongly suggest most applications calling *libdwarf* follow the suggestion above (passing a valid Dwarf\_Error address in the last argument when calling *libdwarf* where there are such Dwarf\_Error arguments) there are other approaches described just below that might be worth considering in small simple applications as they reduce the Dwarf\_Error argument to just passing 0 (null pointer)..

As an alternative to the above, a client program can also specify a function to be invoked upon detection of an error at the time the library is initialized (see `dwarf_init_b()`). When a *libdwarf* routine detects an error, this function is called with two arguments: a code indicating the nature of the error and a pointer provided by the client at initialization (again see `dwarf_init_b()`). This pointer argument can be used to relay information between the error handler and other routines of the client program. A client program can specify or change both the error handling function and the pointer argument after initialization using `dwarf_seterrhand()` and `dwarf_seterrarg()`.

In the final case, where *libdwarf* functions are not provided a pointer to a Dwarf\_Error descriptor, and no error handling function was provided at initialization, *libdwarf* functions print a short message to stdout and terminate execution with `abort()`.

Before March 2016 *libdwarf* gave up when there was no error handling by emitting a short message on stderr calling `abort(3C)`.

The following lists the processing steps taken upon detection of an error:

1. Check the error argument; if not a *NULL* pointer, allocate and initialize a Dwarf\_Error descriptor with information describing the error, place this descriptor in the area pointed to by error, and return a value indicating an error condition.
2. If an errhand argument was provided to `dwarf_init_b()` at initialization, call `errhand()` passing it the error descriptor and the value of the errarg argument provided to `dwarf_init_b()`. If the error handling function returns, return `DW_DLV_ERROR` indicating an error condition.
3. If neither the error argument nor an errhand argument was provided Terminate program execution by calling `abort(3C)`.

In all cases, it is clear from the value returned from a function that an error occurred in executing the function, since `DW_DLV_ERROR` is returned.

As can be seen from the above steps, the client program can provide an error handler at initialization, and still provide an error argument to *libdwarf* functions when it is not

desired to have the error handler invoked.

If a `libdwarf` function is called with invalid arguments, the behavior is undefined. In particular, supplying a `NULL` pointer to a `libdwarf` function (except where explicitly permitted), or pointers to invalid addresses or uninitialized data causes undefined behavior; the return value in such cases is undefined, and the function may fail to invoke the caller supplied error handler or to return a meaningful error number. Implementations also may abort execution for such cases.

Some errors are so inconsequential that it does not warrant rejecting an object or returning an error. An example would be a frame length not being a multiple of an address-size (right now this is the only such inconsequential error). To make it possible for a client to report such errors the function `dwarf_get_harmless_error_list` returns strings with error text in them. This function may be ignored if client code does not want to bother with such error reporting. See `DW_DLE_DEBUG_FRAME_LENGTH_NOT_MULTIPLE` in the `libdwarf` source code.

## 4.1 Returned values in the functional interface

Values returned by `libdwarf` functions to indicate success and errors are enumerated in Figure 2. The `DW_DLV_NO_ENTRY` case is useful for functions need to indicate that while there was no data to return there was no error either. For example, `dwarf_siblingof()` may return `DW_DLV_NO_ENTRY` to indicate that that there was no sibling to return.

SYMBOLIC NAME	VALUE	MEANING
<code>DW_DLV_ERROR</code>	1	Error
<code>DW_DLV_OK</code>	0	Successful call
<code>DW_DLV_NO_ENTRY</code>	-1	No applicable value

**Figure 2.** Error Indications

Each function in the interface that returns a value returns one of the integers in the above figure.

If `DW_DLV_ERROR` is returned and a pointer to a `Dwarf_Error` pointer is passed to the function, then a `Dwarf_Error` handle is returned through the pointer. No other pointer value in the interface returns a value. After the `Dwarf_Error` is no longer of interest, a `dwarf_dealloc(dbg, dw_err, DW_DLA_ERROR)` on the error pointer is appropriate to free any space used by the error information.

If `DW_DLV_NO_ENTRY` is returned no pointer value in the interface returns a value.

If `DW_DLV_OK` is returned, the `Dwarf_Error` pointer, if supplied, is not touched, but any other values to be returned through pointers are returned. In this case calls (depending on the exact function returning the error) to `dwarf_dealloc()` may be appropriate once the particular pointer returned is no longer of interest.

Pointers passed to allow values to be returned through them are uniformly the last pointers in each argument list.

All the interface functions are defined from the point of view of the writer-of-the-library (as is traditional for UN\*X library documentation), not from the point of view of the user of the library. The caller might code:

```
Dwarf_Line line;
Dwarf_Signed ret_loff;
Dwarf_Error err;
int retval = dwarf_lineoff(line, &ret_loff, &err);
```

for the function defined as

```
int dwarf_lineoff(Dwarf_Line line, Dwarf_Signed *return_lineoff,
    Dwarf_Error* err);
```

and this document refers to the function as returning the value through \*err or \*return\_lineoff or uses the phrase "returns in the location pointed to by err". Sometimes other similar phrases are used.

## 5. Memory Management

Several of the functions that comprise *libdwarf* return pointers (opaque descriptors) to structures that have been dynamically allocated by the library. To manage dynamic memory the function `dwarf_dealloc()` is provided to free storage allocated as a result of a call to a *libdwarf* function. Some additional functions (described later) are provided to free storage in particular circumstances. This section describes the general strategy that should be taken by a client program in managing dynamic storage.

By default *libdwarf* tracks its allocations and `dwarf_finish()` cleans up allocations where `dwarf_dealloc()` was not called. See `dwarf_set_de_alloc_flag()` below.

### 5.1 Read-only Properties

All pointers (opaque descriptors) returned by or as a result of a *libdwarf Consumer Library* call should be assumed to point to read-only memory. The results are undefined for *libdwarf* clients that attempt to write to a region pointed to by a value returned by a *libdwarf Consumer Library* call.

### 5.2 Storage Deallocation

See the section "Returned values in the functional interface", above, for the general rules where calls to `dwarf_dealloc()` are appropriate.

As of May 2020 there are additional dealloc calls which enable type-checking the calls: `dwarf_dealloc_error()`, `dwarf_dealloc_die()`, and `dwarf_dealloc_attribute()`.



### 5.2.1 dwarf\_dealloc()

The first prototype is the generic one that can dealloc any of the libdwarf types, such as DW\_DLA\_DIE etc.. This has the disadvantages that the space\_to\_dealloc argument cannot be type checked and the appropriate\_dla\_name is a simple integer, hence not meaningfully checkable either.

Whenever possible, use a type-safe deallocation call (for several types that is the only documented deallocation call) and for Dwarf\_Die Dwarf\_Attribute or Dwarf\_Error use the following dealloc functions instead of this one. The use of this form remains fully supported,

```
void dwarf_dealloc(Dwarf_Debug dbg,
                  void *space_to_dealloc,
                  int appropriate_dla_name);
```

#### Figure 3. Example\_dwarf\_dealloc()

```
void exampledealloc(Dwarf_Debug dbg,Dwarf_Die somedie)
{
    dwarf_dealloc(dbg,somedie,DW_DLA_DIE);
}
```

### 5.2.2 dwarf\_dealloc\_die()

The second prototype is only to dealloc a Dwarf\_Die. Any call to this is typechecked.

```
void dwarf_dealloc_die(Dwarf_Die mydie);
```

#### Figure 4. Example\_dwarf\_dealloc\_die()

```
void exampledeallocdie(Dwarf_Die somedie)
{
    dwarf_dealloc_die(somedie);
}
```

### 5.2.3 dwarf\_dealloc\_error()

The second prototype is only to dealloc a Dwarf\_Error. These arise when some libdwarf call returns DW\_DLV\_ERROR. Any call to this is typechecked.

```
void dwarf_dealloc_error(Dwarf_Debug dbg,
                        Dwarf_Die mydie);
```

**Figure 5.** Example\_dwarf\_dealloc\_error()

```
void exampledeallocerror(Dwarf_Debug dbg,Dwarf_Error err)
{
    dwarf_dealloc_error(dbg,err);'
}
```

#### 5.2.4 dwarf\_dealloc\_attribute()

The second prototype is only to dealloc a Dwarf\_Attribute. These arise from calls from dwarf\_attrlist(). Any call to this is typechecked.

```
void dwarf_dealloc_error(Dwarf_Debug dbg,
    Dwarf_Die mydie);
```

**Figure 6.** Example\_dwarf\_dealloc\_attribute()

```
void exampledeallocerror(Dwarf_Attribute attr)
{
    dwarf_dealloc_attribute(attr);'
}
```

In some cases the pointers returned by a *libdwarf* call are pointers to data which is not freeable. The library knows from the allocation type provided to it whether the space is freeable or not and will not free inappropriately when dwarf\_dealloc() is called. So it is vital that dwarf\_dealloc() be called with the proper allocation type.

For most storage allocated by *libdwarf*, the client can free the storage for reuse by calling dwarf\_dealloc(), providing it with the Dwarf\_Debug descriptor specifying the object for which the storage was allocated, a pointer to the area to be free-ed, and an identifier that specifies what the pointer points to (the allocation type). For example, to free a Dwarf\_Die die belonging to the object represented by Dwarf\_Debug dbg, allocated by a call to dwarf\_siblingof(), the call to dwarf\_dealloc() would be:

```
dwarf_dealloc(dbg, die, DW_DLA_DIE);
or, preferably,
dwarf_dealloc_die(die);
```

To free storage allocated in the form of a list of pointers (opaque descriptors), each member of the list should be deallocated, followed by deallocation of the actual list itself. The following code fragment uses an invocation of dwarf\_attrlist() as an example to illustrate a technique that can be used to free storage from any *libdwarf* routine that returns a list:

**Figure 7.** Example1 dwarf\_attrlist()

```
void example1(Dwarf_Debug dbg,Dwarf_Die somedie)
{
    Dwarf_Signed atcount = 0;
    Dwarf_Attribute *atlist = 0;
    Dwarf_Error error = 0;
    Dwarf_Signed i = 0;
    int errv = 0;

    errv = dwarf_attrlist(somedie, &atlist,&atcount, &error);
    if (errv == DW_DLV_OK) {
        for (i = 0; i < atcount; ++i) {
            /* use atlist[i] */
            dwarf_dealloc_attribute(atlist[i]);
            /* This original form still works.
            dwarf_dealloc(dbg, atlist[i], DW_DLA_ATTR);
            */
        }
        dwarf_dealloc(dbg, atlist, DW_DLA_LIST);
    }
}
```

The Dwarf\_Debug returned from dwarf\_init\_b() or dwarf\_elf\_init\_b() cannot be freed using dwarf\_dealloc(). The function dwarf\_finish() will deallocate all dynamic storage associated with an instance of a Dwarf\_Debug type. In particular, it will deallocate all dynamically allocated space associated with the Dwarf\_Debug descriptor, and finally make the descriptor invalid.

An Dwarf\_Error returned from dwarf\_init\_b() or dwarf\_elf\_init\_b() in case of a failure cannot be freed using dwarf\_dealloc(). The only way to free the Dwarf\_Error from either of those calls is to use *free(3)* directly. Every Dwarf\_Error must be freed by dwarf\_dealloc() except those returned by dwarf\_init\_b() or dwarf\_elf\_init\_b().

The codes that identify the storage pointed to in calls to dwarf\_dealloc() are described in figure 8.

IDENTIFIER	USED TO FREE
DW_DLA_STRING	char*
DW_DLA_LOC	Dwarf_Loc
DW_DLA_LOCDISC	Dwarf_Locdesc
DW_DLA_ELLIST	Dwarf_Ellist (not used)
DW_DLA_BOUNDS	Dwarf_Bounds (not used)
DW_DLA_BLOCK	Dwarf_Block
DW_DLA_DEBUG	Dwarf_Debug (do not use)
DW_DLA_DIE	Dwarf_Die
DW_DLA_LINE	Dwarf_Line
DW_DLA_ATTR	Dwarf_Attribute
DW_DLA_TYPE	Dwarf_Type (not used)
DW_DLA_SUBSCR	Dwarf_Subscr (not used)
DW_DLA_GLOBAL_CONTEXT	Dwarf_Global
DW_DLA_ERROR	Dwarf_Error
DW_DLA_LIST	a list of opaque descriptors
DW_DLA_LINEBUF	Dwarf_Line* (not used)
DW_DLA_ARANGE	Dwarf_Arange
DW_DLA_ABBREV	Dwarf_Abbrev
DW_DLA_FRAME_OP	Dwarf_Frame_Op
DW_DLA_CIE	Dwarf_Cie
DW_DLA_FDE	Dwarf_Fde
DW_DLA_LOC_BLOCK	Dwarf_Loc Block
DW_DLA_FRAME_BLOCK	Dwarf_Frame Block (not used)
DW_DLA_FUNC_CONTEXT	Dwarf_Func
DW_DLA_TYPENAME_CONTEXT	Dwarf_Type
DW_DLA_VAR_CONTEXT	Dwarf_Var
DW_DLA_WEAK_CONTEXT	Dwarf_Weak
DW_DLA_PUBTYPES_CONTEXT	Dwarf_Type

**Figure 8.** Allocation/Deallocation Identifiers

## 6. Functional Interface

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

The following sections describe these functions.

### 6.1 Initialization Operations

These functions are concerned with preparing an object file for subsequent access by the functions in *libdwarf* and with releasing allocated resources when access is complete.

### 6.1.1 dwarf\_init\_path()

```
int dwarf_init_path(
    const char *    path,
    char *          true_path_out_buffer,
    unsigned        true_path_bufferlen,
    Dwarf_Unsigned  access,
    unsigned        groupnumber,
    Dwarf_Handler   errhand,
    Dwarf_Ptr       errarg,
    Dwarf_Debug*    dbg,
    const char *    reserved1,
    Dwarf_Unsigned * reserved2,
    Dwarf_Unsigned * reserved3,
    Dwarf_Error*    * error);
```

On success the function returns DW\_DLV\_OK, and returns a pointer to an initialized Dwarf\_Debug through the dbg argument. All this work identically across all supported object file types.

If DW\_DLV\_NO\_ENTRY is returned there is no such file and nothing else is done or returned.

If DW\_DLV\_ERROR is returned a Dwarf\_Error is returned through the error pointer. and nothing else is done or returned.

Now we turn to the arguments.

Pass in the name of the object file via the path argument.

For MacOS pass in a pointer to true\_path\_out\_buffer big pointing to a buffer large enough to hold the passed-in path if that were doubled plus adding 100 characters. Then pass that length in the true\_path\_bufferlen argument. If a file is found (the dSYM path or if not that the original path) the final path is copied into true\_path\_out\_buffer. In any case, This is harmless with non-MacOS executables, but for non-MacOS true\_path\_out\_buffer will just match path.

When you know you won't be reading MacOS executables you could skip the MacOS special treatment by passing 0 as arguments to true\_path\_out\_buffer and true\_path\_bufferlen. Or use dwarf\_init\_b() instead of dwarf\_init\_path()

Pass in the usual DW\_DLC\_READ (which only ever applied to libelf) to access. Currently no other value is allowed. Non-elf objects currently ignore this field.

The groupnumber argument indicates which group is to be accessed Group one is normal dwarf sections such as .debug\_info. Group two is DWARF5 dwo split-dwarf dwarf sections such as .debug\_info.dwo.

Groups three and higher are for COMDAT groups. If an object file has only sections

from one of the groups then passing zero will access that group. Otherwise passing zero will access only group one.

See `dwarf_sec_group_sizes()` and `dwarf_sec_group_map()` for more group information.

Typically pass in `DW_GROUPNUMBER_ANY` to `groupnumber`. Non-elf objects do not use this field.

The `errhand` argument is a pointer to a function that will be invoked whenever an error is detected as a result of a *libdwarf* operation. The `errarg` argument is passed as an argument to the `errhand` function.

`dbg`.

`reserved1`, `reserved2`, and `reserved3` are currently unused, pass 0 in to all three.

Pass in 0 (null) or a pointer to a `Dwarf_Error` to the `error` argument if you wish *libdwarf* to return an error code.

### 6.1.2 `dwarf_init_b()`

```
int dwarf_init_b(  
int fd,  
    Dwarf_Unsigned access,  
    unsigned group_number,  
    Dwarf_Handler errhand,  
    Dwarf_Ptr errarg,  
    Dwarf_Debug * dbg,  
    Dwarf_Error *error)
```

When it returns `DW_DLV_OK`, the function `dwarf_init_b()` returns through `dbg` a `Dwarf_Debug` descriptor that represents a handle for accessing debugging records associated with the open file descriptor `fd`. `DW_DLV_NO_ENTRY` is returned if the object does not contain DWARF debugging information. `DW_DLV_ERROR` is returned if an error occurred.

The `access` argument indicates what access is allowed for the section. The `DW_DLC_READ` parameter is valid for read access (only read access is defined or discussed in this document).

The `groupnumber` argument indicates which group is to be accessed Group one is normal dwarf sections such as `.debug_info`. Group two is DWARF5 dwo split-dwarf dwarf sections such as `.debug_info.dwo`.

Groups three and higher are for COMDAT groups. If an object file has only sections from one of the groups then passing zero will access that group. Otherwise passing zero

will access only group one.

See `dwarf_sec_group_sizes()` and `dwarf_sec_group_map()` for more group information.

The `errhand` argument is a pointer to a function that will be invoked whenever an error is detected as a result of a *libdwarf* operation. The `errarg` argument is passed as an argument to the `errhand` function.

The file descriptor associated with the `fd` argument must refer to an ordinary file (i.e. not a pipe, socket, device, `/proc` entry, etc.), be opened with the at least as much permission as specified by the `access` argument, and cannot be closed or used as an argument to any system calls by the client until after `dwarf_finish()` is called. The seek position of the file associated with `fd` is undefined upon return of `dwarf_init()`.

Historical Note: With SGI IRIX, by default it was allowed that the `app close()` `fd` immediately after calling `dwarf_init()`, but that is not a portable approach (that it worked was an accidental side effect of the fact that SGI IRIX used `ELF_C_READ_MMAP` in its hidden internal call to `elf_begin()`). The portable approach is to consider that `fd` must be left open till after the corresponding `dwarf_finish()` call has returned.

Since `dwarf_init()` uses the same error handling processing as other *libdwarf* functions (see *Error Handling* above), client programs will generally supply an `error` parameter to bypass the default actions during initialization unless the default actions are appropriate.

### 6.1.3 dwarf\_init()

```
int dwarf_init(  
    int fd,  
    Dwarf_Unsigned access,  
    Dwarf_Handler errhand,  
    Dwarf_Ptr errarg,  
    Dwarf_Debug * dbg,  
    Dwarf_Error *error)
```

`dwarf_init()` is identical to `dwarf_init_b()` except that `dwarf_init()` is missing the `groupnumber` argument so access to an object file containing both dwo and non-dwo DWARF5 object sections will access only group one (and will ignore the dwo sections).

The `dwarf_get_elf()` function cannot succeed when using `dwarf_init()` or `dwarf_init_b()` or `dwarf_init_path()` to open an object file.

### 6.1.4 dwarf\_set\_de\_alloc\_flag()

```
int dwarf_set_de_alloc_flag(  
    int v)
```

`dwarf_set_de_alloc_flag()` sets and returns a flag value applying to the current running instance of `libdwarf`. It's action sets an internal value, and that value should be set/changed (if you wish to do that) before any other `libdwarf` calls.

By default `libdwarf` keeps track of all its internal allocations. So if the documentation here says you should do `dwarf_dealloc()` calls (or other calls documented here for specific functions) and you omit some or all of them then calling `dwarf_finish()` will clean up all those allocations left undone.

If you call `dwarf_set_de_alloc_flag(0)` then `libdwarf` will not keep track of allocations so your code must do all `dwarf_dealloc()` calls as defined below.

If you call `dwarf_set_de_alloc_flag(1)` that sets/restores the setting to its default value so from that point all new internal allocations will be tracked and `dwarf_finish()` can clean the new ones up.

The return value of `dwarf_set_de_alloc_flag()` is the previous value of the internal flag: One (1) is the default, meaning record allocations.. Zero (0) is the other possible value, meaning do not record `libdwarf` allocations.

It is best to ignore this call unless you have gigantic DWARF sections and you need whatever percent speed improvement from `libdwarf` that you can get. If you do use it then by all means use tools such as `cc --fsanitize...` or `valgrind` to ensure there are no leaks in your application (at least given your test cases).

The function name echos the spelling of a `libdwarf`-internal field in struct `Dwarf_Debug_s` named `de_alloc_tree`.

### 6.1.5 Dwarf\_Handler function

This is an example of a valid error handler function. A pointer to this (or another like it) may be passed to `dwarf_elf_init_b()` or `dwarf_init_b()`.

```
static void  
simple_error_handler(Dwarf_Error error, Dwarf_Ptr errarg)  
{  
    printf("libdwarf error: %d %s\n",  
        dwarf_errno(error), dwarf_errmsg(error));  
    exit(1);  
}
```

This will only be called if an error is detected inside `libdwarf` and the `Dwarf_Error` argument passed to `libdwarf` is `NULL`. A `Dwarf_Error` will be created with the error number assigned by the library and passed to the error handler.



The second argument is a copy of the value passed in to `dwarf_elf_init_b()` or `dwarf_init()` as the `errarg()` argument. Typically the init function would be passed a pointer to an application-created struct containing the data the application needs to do what it wants to do in the error handler.

In a language with exceptions or exception-like features an exception could be thrown here. Or the application could simply give up and call `exit()` as in the sample given above.

### 6.1.6 `dwarf_elf_init_b()` [deprecated 2019]

```
int dwarf_elf_init_b(
    Elf * elf_file_pointer,
    Dwarf_Unsigned access,
    unsigned groupnumber,
    Dwarf_Handler errhand,
    Dwarf_Ptr errarg,
    Dwarf_Debug * dbg,
    Dwarf_Error *error)
```

We recommend you change to calling `dwarf_init_b()` or `dwarf_init_path()` instead. The `dwarf_elf_init()` and `dwarf_elf_init_b()` interfaces give no benefit over the other interfaces (other than allowing `dwarf_get_elf()` to succeed).

The function `dwarf_elf_init_b()` is similar to `dwarf_init_b()` but an open `Elf *` pointer is passed instead of a file descriptor so `dwarf_get_elf()` can succeed.

The client is allowed to use the `Elf *` pointer for its own purposes without restriction during the time the `Dwarf_Debug` is open, except that the client should not `elf_end()` the pointer till after `dwarf_finish` is called.

### 6.1.7 `dwarf_elf_init()` [deprecated 2019]

```
int dwarf_elf_init(
    Elf * elf_file_pointer,
    Dwarf_Unsigned access,
    Dwarf_Handler errhand,
    Dwarf_Ptr errarg,
    Dwarf_Debug * dbg,
    Dwarf_Error *error)
```

The function `dwarf_elf_init()` is identical to `dwarf_init[_b]()` except an open `Elf *` pointer is passed instead of a file descriptor so `dwarf_get_elf()` can succeed.

Code using `dwarf_elf_init[_b]()` should be switched to calling `dwarf_init_b()`.

### 6.1.8 dwarf\_get\_elf()

```
int dwarf_get_elf(  
    Dwarf_Debug dbg,  
    Elf **      elf,  
    Dwarf_Error *error)
```

The function `dwarf_get_elf()` is only meaningful if a `dwarf_elf_init[_b]()` function was used to initialize the pointer `dbg`. None of the other `dwarf*init*()` functions here ever use `libelf`, so there is no `elf` pointer to return through the pointer, so and the call will return `DW_DLV_NO_ENTRY`. In addition, this function is also not meaningful for an object file that is not in the Elf format.

When it returns `DW_DLV_OK`, the function `dwarf_get_elf()` returns through the pointer `elf` the `Elf *` handle used to access the object represented by the `Dwarf_Debug` descriptor `dbg`. It returns `DW_DLV_ERROR` on error.

### 6.1.9 dwarf\_set\_tied\_dbg()

```
int dwarf_set_tied_dbg(  
    Dwarf_Debug dbg,  
    Dwarf_Debug tieddbg,  
    Dwarf_Error *error)
```

The function `dwarf_set_tied_dbg()` enables cross-object access of DWARF data. If a DWARF5 Package object has `DW_FORM_addrx` or `DW_FORM_GNU_addr_index` or one of the other indexed forms in DWARF5 in an address attribute one needs both the Package file and the executable to extract the actual address with `dwarf_formaddr()`. The utility function `dwarf_addr_form_is_indexed(form)` is a handy way to know if an address form is indexed. One does a normal `dwarf_elf_init_b()` or `dwarf_init()_b` on each object and then tie the two together with a call such as:

**Figure 9.** Example2 dwarf\_set\_died\_dbg()

```
void example2(Dwarf_Debug dbg, Dwarf_Debug tieddbg)
{
    Dwarf_Error error = 0;
    int res = 0;

    /* Do the dwarf_init_b() or dwarf_elf_init_b()
       calls to set
       dbg, tieddbg at this point. Then: */
    res = dwarf_set_tied_dbg(dbg,tieddbg,&error);
    if (res != DW_DLV_OK) {
        /* Something went wrong*/
    }
}
```

When done with both dbg and tieddbg do the normal finishing operations on both in any order.

It is possible to undo the tying operation with

**Figure 10.** Example3 dwarf\_set\_tied\_dbg() obsolete

```
void example3(Dwarf_Debug dbg)
{
    Dwarf_Error error = 0;
    int res = 0;
    res = dwarf_set_tied_dbg(dbg,NULL,&error);
    if (res != DW_DLV_OK) {
        /* Something went wrong*/
    }
}
```

It is not necessary to undo the tying operation before finishing on the dbg and tieddbg.

### 6.1.10 dwarf\_get\_tied\_dbg()

```
int dwarf_get_tied_dbg(
    Dwarf_Debug /*dbg*/,
    Dwarf_Debug * /*tieddbg_out*/,
    Dwarf_Error * /*error*/)
```

dwarf\_get\_tied\_dbg returns DW\_DLV\_OK and sets tieddbg\_out to the pointer to the 'tied' Dwarf\_Debug. If there is no 'tied' object tieddbg\_out is set to NULL.

On error it returns DW\_DLV\_ERROR.

It never returns DW\_DLV\_NO\_ENTRY.

### 6.1.11 dwarf\_finish()

```
int dwarf_finish(  
    Dwarf_Debug dbg,  
    Dwarf_Error *error)
```

The function `dwarf_finish()` releases all *Libdwarf* internal resources associated with the descriptor `dbg`, and invalidates `dbg`. It returns `DW_DLV_ERROR` if there is an error during the finishing operation. It returns `DW_DLV_OK` for a successful operation.

Because `int dwarf_init()` opens an Elf descriptor on its `fd` and `dwarf_finish()` does not close that descriptor, an app should use `dwarf_get_elf` and should call `elf_end` with the pointer returned through the `Elf**` handle created by `int dwarf_init()`.

### 6.1.12 dwarf\_set\_stringcheck()

```
int dwarf_set_stringcheck(  
    int stringcheck)
```

The function `int dwarf_set_stringcheck()` sets a global flag and returns the previous value of the global flag.

If the `stringcheck` global flag is zero (the default) `libdwarf` does string length validity checks (the checks do slow `libdwarf` down very slightly). If the `stringcheck` global flag is non-zero `libdwarf` does not do string length validity checks.

The global flag is really just 8 bits long, upperbits are not noticed or recorded.

### 6.1.13 dwarf\_set\_reloc\_application()

```
int dwarf_set_reloc_application(  
    int apply)
```

The function `int dwarf_set_reloc_application()` sets a global flag and returns the previous value of the global flag.

If the `reloc_application` global flag is non-zero (the default) then the applicable `.rela` section (if one exists) will be processed and applied to any DWARF section when it is read in. If the `reloc_application` global flag is zero no such relocation-application is attempted.

Not all machine types (elf header `e_machine`) or all relocations are supported, but then very few relocation types apply to DWARF debug sections.

The global flag is really just 8 bits long, upperbits are not noticed or recorded.

It seems unlikely anyone will need to call this function.

#### 6.1.14 `dwarf_record_cmdline_options()`

```
int dwarf_record_cmdline_options(  
    Dwarf_Cmdline_Options options)
```

The function `int dwarf_record_cmdline_options()` copies a `Dwarf_Cmdline_Options` structure from consumer code to `libdwarf`.

The structure is defined in `libdwarf.h`.

The initial version of this structure has a single field `check_verbose_mode` which, if non-zero, tells `libdwarf` to print some detailed messages to `stdout` in case certain errors are detected.

The default for this value is `FALSE` (0) so the extra messages are off by default.

#### 6.1.15 `dwarf_object_init_b()`

```
int dwarf_object_init_b(  
    Dwarf_Obj_Access_Interface* obj,  
    Dwarf_Handler errhand,  
    Dwarf_Ptr errarg,  
    unsigned groupnumber,  
    Dwarf_Debug* dbg,  
    Dwarf_Error* error)
```

The function `int dwarf_object_init_b()` enables access to non-Elf object files by allowing the caller to then provide function pointers to code (user-written, not part of `libdwarf`) that will look, to `libdwarf`, as if `libdwarf` was reading Elf.

See `int dwarf_init_b()` for additional information on the arguments passed in (the `obj` argument here is a set of function pointers and describing how to access non-Elf files is beyond the scope of this document).

As a hint, note that the source files with `dwarf_elf_init_file_ownership()` (`dwarf_original_elf_init.c`) and `dwarf_elf_object_access_init()` (`dwarf_elf_access.c`) are the only sources that would need replacement for a different object format. The replacement would need to emulate certain conventions of Elf objects, (mainly that

section index 0 is an empty section) but the rest of libdwarf uses what these two source files set up without knowing how to operate on Elf.

Writing the functions needed to support non-Elf will require study of Elf and of the object format involved. The topic is beyond the scope of this document.

#### 6.1.16 dwarf\_object\_init()

```
int dwarf_object_init(  
    Dwarf_Obj_Access_Interface* obj,  
    Dwarf_Handler errhand,  
    Dwarf_Ptr errarg,  
    Dwarf_Debug* dbg,  
    Dwarf_Error* error)
```

The function `int dwarf_object_init()` is the same as `int dwarf_object_init_b()` except `int dwarf_object_init()` is missing the `groupnumber` argument so DWARF5 split dwarf objects cannot be fully handled.

#### 6.1.17 dwarf\_get\_real\_section\_name()

```
int dwarf_get_real_section_name( Dwarf_Debug dbg,  
    const char * std_section_name,  
    const char ** actual_sec_name_out,  
    Dwarf_Small * marked_compressed,  
    Dwarf_Small * marked_zlib_compressed,  
    Dwarf_Small * marked_shf_compressed,  
    Dwarf_Unsigned * compressed_length,  
    Dwarf_Unsigned * uncompressed_length,  
    Dwarf_Error * error);
```

Elf sections are sometimes compressed to reduce the disk footprint of the sections. It's sometimes interesting to library users what the real name was in the object file and whether it was compressed. Libdwarf uncompresses such sections automatically. It's not usually necessary to know the true name or anything about compression.

The caller passes in a `Dwarf_Debug` pointer and a standard section name such as `".debug_info"`. On success the function returns (through the other arguments) the true section name and a flag which, if non-zero means the section was compressed and a flag which, if non-zero means the section had the Elf section flag `SHF_COMPRESSED` set. The caller must ensure that the memory pointed to by `actual_sec_name_out`, `marked_zcompressed`, and `marked_zlib_compressed`, `marked_shf_compressed`, `compressed_length`, `uncompressed_length`, is zero at the point of call.

The flag `*marked_compressed`, if non-zero, means the section name started with `.zdebug` (indicating compression was done).

The flag `marked_zlib_compressed`, if non-zero means the initial bytes of the section start with the ASCII characters `ZLIB` and the section was compressed.

The flag `marked_shf_compressed` if non-zero means the Elf section `sh_flag SHF_COMPRESSED` is set and the section was compressed.. The flag value in an elf section header is  $(1 \ll 11)$  (0x800).

The value `compressed_length` is passed back through the pointer if and only if the section is compressed and the pointer is non-null.

The value `uncompressed_length` is passed back through the pointer if and only if the section is compressed and the pointer is non-null.

If the section name passed in is not used by `libdwarf` for this object file the function returns `DW_DLV_NO_ENTRY`

On error the function returns `DW_DLV_ERROR`.

The string pointed to by `*actual_sec_name_out` must not be `free()`d.

### 6.1.18 `dwarf_package_version()`

```
const char * dwarf_package_version(void);
```

The package version is set in `config.h` (from its value in `configure.ac` and in `CMakeLists.txt` in the source tree) at the build time of the library. A pointer to a static string is returned by this function. The format is standard ISO date format. For example "20180718". It's not entirely clear how this actually helps. But there is a request for this and we provide it as of 23 October 2019.

## 6.2 Object Type Detectors

These are used by `libdwarf` and may be of use generally. They have no connection to any `Dwarf_Debug` data as you see from the arguments passed in.

### 6.2.1 `dwarf_object_detector_path()`

```
int dwarf_object_detector_path(const char *path,
                              char *outpath,
                              unsigned long outpath_len,
                              unsigned *ftype,
                              unsigned *endian,
                              unsigned *offsetsize,
                              Dwarf_Unsigned *filesize,
                              int * errcode);
```

On success the function returns `DW_DLV_OK`, and returns various data through the arguments (described just below). This works identically across all supported object file types.

If `DW_DLV_NO_ENTRY` is returned there is no such file and nothing else is done or returned.

If `DW_DLV_ERROR` is returned a `Dwarf_Error` is returned through the error pointer. and nothing else is done or returned.

Now we turn to the arguments. Pass in the name of the object file via the `path` argument.

To `outpath` pass in a pointer big enough to hold the passed-in path if that were doubled plus adding 100 characters. Then pass that length in the `outpath_len` argument. The path will be copied to `outpath`. In the case of certain MacOS dSYM object files the final output of the dSYM file (with MacOS conventional directories added) is copied into `outpath`. Where the MacOS local directory tree is missing or incomplete `outpath` will be left as a zero-length string.

To entirely skip the MacOS special treatment pass 0 as arguments to `outpath` and `outpath_len`.

The `fctype` pointer argument returns `DW_FTYPE_ELF`, `DW_FTYPE_MACH_O` , `DW_FTYPE_PE` , `DW_FTYPE_ARCHIVE` or `DW_FTYPE_UNKNOWN` to the caller. The `DW_FTYPE_ARCHIVE` value says nothing whatever about the contents of the archive.

The `endian` pointer argument returns `DW_ENDIAN_BIG`, `DW_ENDIAN_LITTLE` , `DW_ENDIAN_SAME` , `DW_ENDIAN_OPPOSITE` or `DW_ENDIAN_UNKNOWN` to the caller.

The `offsetsize` pointer argument returns a size value from the object file. If the object file uses 32-bit offsets it returns 32, and if 64-bit offsets it returns 64. Each object type uses such values but the ways the value is used varies.

The `filesize` pointer argument returns the size, in bytes, of the object file. This is essentially useless for `DW_FTYPE_ARCHIVE` files, one thinks.

The `errcode` pointer argument returns (if and only if `DW_DLV_ERROR` is returned by the function) an integer error code. At this time there is no handy function to turn that error code into a string. In the libdwarf source you will find that code in the `DW_DLE_*` error list.

### **6.2.2 dwarf\_object\_detector\_fd()**



```
int dwarf_object_detector_fd(int fd,
    unsigned *ftype,
    unsigned *endian,
    unsigned *offsetsize,
    Dwarf_Unsigned *filesize,
    int * errcode);
```

`dwarf_object_detector_fd()` is the same as `dwarf_object_detector_path()` except that no path strings apply to `dwarf_object_detector_fd()`.

## 6.3 Section Group Operations

The section group data is essential information when processing an object with COMDAT section group DWARF sections or with both split-dwarf (.dwo sections) and non-split dwarf sections.

It relies on Elf section groups, whereas some compilers rely instead on relocation information to identify section groups. These relocation-specified groupings are not understood at this time.

A standard DWARF2 or DWARF3 or DWARF4 object (Old Standard Object, or OSO) will not contain any of those new sections. The DWARF4 standard, Appendix E.1 "Using Compilation Units" offers an overview of COMDAT section groups. `libdwarf` assigns the group number one(1) to OSO DWARF. Any sections that are split dwarf (section name ending in .dwo or one of the two special DWP index sections) are assigned group number two(2) by `libdwarf`. COMDAT section groups are assigned groups numbers 3 and higher as needed.

The COMDAT section group uses are not well defined, but popular compilations systems are using such sections. There is no meaningful documentation that we can find (so far) on how the COMDAT section groups are used, so `libdwarf` is based on observations of what compilers generate.

### 6.3.1 dwarf\_sec\_group\_sizes()

```
int dwarf_dwarf_sec_group_sizes(
    Dwarf_Debug dbg,
    Dwarf_Unsigned * section_count_out,
    Dwarf_Unsigned * group_count_out,
    Dwarf_Unsigned * selected_group_out,
    Dwarf_Unsigned * map_entry_count_out,
    Dwarf_Error * error)
```

The function `dwarf_sec_group_sizes()` may be called on any open `Dwarf_Debug`. It returns `DW_DLV_OK` on success and returns values via the pointer

arguments.

Once the Dwarf\_Debug is open the group information is set and it will not change for the life of this Dwarf\_Debug.

The \*section\_count\_out is set to the number of sections in the object. Many of the sections will be irrelevant to libdwarf.

The \*group\_count\_out is set to the number of groups in the object (as libdwarf counts them). An OSO will have exactly one group. A DWP object will have exactly one group. If is more than one group consumer code will likely want to open additional Dwarf\_Debug objects and request relevant information to process the DWARF contents. An executable or a DWP object will always have a \*group\_count\_out of one(1). An executable or a shared library cannot have any COMDAT section groups as the linker will have dealt with them.

The \*selected\_group\_out is set to the group number that this Dwarf\_Debug will focus on. See dwarf\_sec\_group\_map() for additional details on how \*selected\_group\_out is interpreted.

The \*map\_entry\_count\_out is set to the number of entries in the map. See dwarf\_sec\_group\_map().

On failure it returns DW\_DLV\_ERROR and sets \*error

The initial implementation never returns DW\_DLV\_ERROR or DW\_DLV\_NO\_ENTRY but callers should allow for that possibility.

### 6.3.2 dwarf\_sec\_group\_map()

```
int dwarf_sec_group_map(  
    Dwarf_Debug      dbg,  
    Dwarf_Unsigned   map_entry_count,  
    Dwarf_Unsigned * group_numbers_array,  
    Dwarf_Unsigned * section_numbers_array,  
    const char      ** sec_names_array,  
    Dwarf_Error      * error)
```

The function dwarf\_sec\_group\_map() may be called on any open Dwarf\_Debug.

The caller must allocate map\_entry\_count arrays used in the following three arguments the and pass the appropriate pointer into the function as well as passing in map\_entry\_count itself.

The map entries returned cover all the DWARF related sections in the object though the selected\_group value will dictate which of the sections in the Dwarf\_Debug will actually be accessed via the usual libdwarf functions. That is, only sections in the selected group may be directly accessed though libdwarf may indirectly access sections

in section group one(1) so relevant details can be accessed, such as abbreviation tables etc. Describing the details of this access outside the current `selected_group` goes beyond what this document covers (as of this writing).

It returns `DW_DLV_OK` on success and sets values into the user-allocated array elements (sorted by section number):

```
group_numbers_array[0]... group_numbers_array[map_entry_count-1]
section_numbers_array[0]... section_numbers_array[map_entry_count-1]
sec_names_array[0]... sec_names_array[map_entry_count-1]
```

`group_numbers_array[0]` for example is set to a group number. One(1), or two(2) or if there are COMDAT groups it will be three(3) or higher.

`section_numbers_array[0]` for example is set to a valid Elf section number relevant to DWARF (each section number shown will be greater than zero).

`sec_names_array[0]` for example is set to a pointer to a string containing the Elf section name of the Elf section number in `sections_number_array[0]`.

On error the function will return `DW_DLV_ERROR` or `DW_DLV_NO_ENTRY` which indicates a serious problem with this object.

Here is an example of use of these functions.

```
void examplesecgroup(Dwarf_Debug dbg)
{
    int res = 0;
    Dwarf_Unsigned section_count = 0;
    Dwarf_Unsigned group_count;
    Dwarf_Unsigned selected_group = 0;
    Dwarf_Unsigned group_map_entry_count = 0;
    Dwarf_Unsigned *sec_nums = 0;
    Dwarf_Unsigned *group_nums = 0;
    const char ** sec_names = 0;
    Dwarf_Error error = 0;
    Dwarf_Unsigned i = 0;

    res = dwarf_sec_group_sizes(dbg, &section_count,
                                &group_count, &selected_group, &group_map_entry_count,
                                &error);
    if(res != DW_DLV_OK) {
        /* Something is badly wrong*/
        return;
    }
    /* In an object without split-dwarf sections
       or COMDAT sections we now have
       selected_group == 1. */
    sec_nums = calloc(group_map_entry_count, sizeof(Dwarf_Unsigned));
    if(!sec_nums) {
        /* FAIL. out of memory */
        return;
    }
    group_nums = calloc(group_map_entry_count, sizeof(Dwarf_Unsigned));
    if(!group_nums) {
        free(group_nums);
        /* FAIL. out of memory */
        return;
    }
    sec_names = calloc(group_map_entry_count, sizeof(char*));
    if(!sec_names) {
        free(group_nums);
        free(sec_nums);
        /* FAIL. out of memory */
        return;
    }

    res = dwarf_sec_group_map(dbg, group_map_entry_count,
                              group_nums, sec_nums, sec_names, &error);
    if(res != DW_DLV_OK) {
```

```
        /* FAIL. Something badly wrong. */
    }
    for( i = 0; i < group_map_entry_count; ++i) {
        /* Now do something with
           group_nums[i], sec_nums[i], sec_names[i] */
    }
    free(group_nums);
    free(sec_nums);
    /* The strings are in Elf data.
       Do not free() the strings themselves.*/
    free(sec_names);
}
```

## 6.4 Section size operations

These operations are informative but not normally needed.

### 6.4.1 dwarf\_get\_section\_max\_offsets\_b()

```
int dwarf_get_section_max_offsets_b(Dwarf_debug dbg,
    Dwarf_Unsigned * /*debug_info_size*/,
    Dwarf_Unsigned * /*debug_abbrev_size*/,
    Dwarf_Unsigned * /*debug_line_size*/,
    Dwarf_Unsigned * /*debug_loc_size*/,
    Dwarf_Unsigned * /*debug_aranges_size*/,
    Dwarf_Unsigned * /*debug_macinfo_size*/,
    Dwarf_Unsigned * /*debug_pubnames_size*/,
    Dwarf_Unsigned * /*debug_str_size*/,
    Dwarf_Unsigned * /*debug_frame_size*/,
    Dwarf_Unsigned * /*debug_ranges_size*/,
    Dwarf_Unsigned * /*debug_pubtypes_size*/,
    Dwarf_Unsigned * /*debug_types_size/);
```

The function `dwarf_get_section_max_offsets_b()` an open `Dwarf_Dbg` and reports on the section sizes by pushing section size values back through the pointers.

Created in October 2011.

### 6.4.2 dwarf\_get\_section\_max\_offsets()

```
int dwarf_get_section_max_offsets(Dwarf_debug dbg,
    Dwarf_Unsigned * /*debug_info_size*/,
    Dwarf_Unsigned * /*debug_abbrev_size*/,
    Dwarf_Unsigned * /*debug_line_size*/,
    Dwarf_Unsigned * /*debug_loc_size*/,
    Dwarf_Unsigned * /*debug_aranges_size*/,
    Dwarf_Unsigned * /*debug_macinfo_size*/,
    Dwarf_Unsigned * /*debug_pubnames_size*/,
    Dwarf_Unsigned * /*debug_str_size*/,
    Dwarf_Unsigned * /*debug_frame_size*/,
    Dwarf_Unsigned * /*debug_ranges_size*/,
    Dwarf_Unsigned * /*debug_pubtypes_size*/);
```

The function is the same as `dwarf_get_section_max_offsets_b()` except it is missing the `debug_types_size()` argument. Though obsolete it is still supported.

## 6.5 Printf Callbacks

This is new in August 2013.

The `dwarf_print_lines()` function is intended as a helper to programs like `dwarfdump` and show some line internal details in a way only the internals of `libdwarf` can show them. But using `printf` directly in `libdwarf` means the caller has limited control of where the output appears. So now the 'printf' output is passed back to the caller through a callback function whose implementation is provided by the caller.

Any code calling `libdwarf` can ignore the functions described in this section completely. If the functions are ignored the messages (if any) from `libdwarf` will simply not appear anywhere.

The `libdwarf.h` header file defines struct `Dwarf_Printf_Callback_Info_s` and `dwarf_register_printf_callback` for those `libdwarf` callers wishing to implement the callback. In this section we describe how one uses that interface. The applications `dwarfdump` and `dwarfdump2` are examples of how these may be used.

### 6.5.1 dwarf\_register\_printf\_callback

```
struct Dwarf_Printf_Callback_Info_s
    dwarf_register_printf_callback(Dwarf_Debug dbg,
    struct Dwarf_Printf_Callback_Info_s * newvalues);
```

The `dwarf_register_printf_callback()` function can only be called after the `Dwarf_Debug` instance has been initialized, the call makes no sense at other times. The function returns the current value of the structure. If `newvalues` is non-null then the

passed-in values are used to initialize the libdwarf internal callback data (the values returned are the values before the newvalues are recorded). If newvalues is null no change is made to the libdwarf internal callback data.

### 6.5.2 Dwarf\_Printf\_Callback\_Info\_s

```
struct Dwarf_Printf_Callback_Info_s {
    void *                dp_user_pointer;
    dwarf_printf_callback_function_type dp_fptr;
    char *                dp_buffer;
    unsigned int          dp_buffer_len;
    int                   dp_buffer_user_provided;
    void *                dp_reserved;
};
```

First we describe the fields as applicable in setting up for a call to `dwarf_register_printf_callback()`.

The field `dp_user_pointer` is remembered by libdwarf and passed back in any call libdwarf makes to the user's callback function. It is otherwise ignored by libdwarf.

The field `dp_fptr` is either NULL or a pointer to a user-implemented function.

If the field `dp_buffer_user_provided` is non-zero then `dp_buffer_len` and `dp_buffer` must be set by the user and libdwarf will use that buffer without doing any malloc of space. If the field `dp_buffer_user_provided` is zero then the input fields `dp_buffer_len` and `dp_buffer` are ignored by libdwarf and space is malloc'd as needed.

The field `dp_reserved` is ignored, it is reserved for future use.

When the structure is returned by `dwarf_register_printf_callback()` the values of the fields before the `dwarf_register_printf_callback()` call are returned.

### 6.5.3 dwarf\_printf\_callback\_function\_type

```
typedef void (* dwarf_printf_callback_function_type)(void * user_pointer,
    const char * linecontent);
```

Any application using the callbacks needs to use the function `dwarf_register_printf_callback()` and supply a function matching the above function prototype from libdwarf.h.

#### 6.5.4 Example of printf callback use in a C++ application using libdwarf

```
struct Dwarf_Printf_Callback_Info_s printfcallbackdata;  
    memset(&printfcallbackdata,0,sizeof(printfcallbackdata));  
    printfcallbackdata.dp_fptr = printf_callback_for_libdwarf;  
    dwarf_register_printf_callback(dbg,&printfcallbackdata);
```

Assuming the user implements something like the following function in her application:

```
void  
printf_callback_for_libdwarf(void *userdata,const char *data)  
{  
    cout << data;  
}
```

It is crucial that the user's callback function copies or prints the data immediately. Once the user callback function returns the data pointer may change or become stale without warning.

### 6.6 Debugging Information Entry Delivery Operations

These functions are concerned with accessing debugging information entries, whether from a .debug\_info, .debug\_types, .debug\_info.dwo, or .debug\_types.dwo .

Since all such sections use similar formats, one set of functions suffices.

#### 6.6.1 dwarf\_get\_die\_section\_name()

```
int  
dwarf_get_die_section_name(Dwarf_Debug dbg,  
    Dwarf_Bool is_info,  
    const char ** sec_name,  
    Dwarf_Error * error);
```

dwarf\_get\_die\_section\_name() lets consumers access the object section name when no specific DIE is at hand. This is useful for applications wanting to print the name, but of course the object section name is not really a part of the DWARF information. Most applications will probably not call this function. It can be called at any time after the Dwarf\_Debug initialization is done. See also dwarf\_get\_die\_section\_name\_b().

The function dwarf\_get\_die\_section\_name() operates on the either the .debug\_info[dwo] section (if is\_info is non-zero) or .debug\_types[dwo] section (if



`is_info` is zero).

If the function succeeds, `*sec_name` is set to a pointer to a string with the object section name and the function returns `DW_DLV_OK`. Do not free the string whose pointer is returned. For non-Elf objects it is possible the string pointer returned will be `NULL` or will point to an empty string. It is up to the calling application to recognize this possibility and deal with it appropriately.

If the section does not exist the function returns `DW_DLV_NO_ENTRY`.

If there is an internal error detected the function returns `DW_DLV_ERROR` and sets the `*error` pointer.

### 6.6.2 `dwarf_get_die_section_name_b()`

```
int
dwarf_get_die_section_name_b(Dwarf_Die die,
    const char ** sec_name,
    Dwarf_Error * error);
```

`dwarf_get_die_section_name_b()` lets consumers access the object section name when one has a DIE. This is useful for applications wanting to print the name, but of course the object section name is not really a part of the DWARF information. Most applications will probably not call this function. It can be called at any time after the `Dwarf_Debug` initialization is done. See also `dwarf_get_die_section_name()`.

If the function succeeds, `*sec_name` is set to a pointer to a string with the object section name and the function returns `DW_DLV_OK`. Do not free the string whose pointer is returned. For non-Elf objects it is possible the string pointer returned will be `NULL` or will point to an empty string. It is up to the calling application to recognize this possibility and deal with it appropriately.

If the section does not exist the function returns `DW_DLV_NO_ENTRY`.

If there is an internal error detected the function returns `DW_DLV_ERROR` and sets the `*error` pointer.

### 6.6.3 `dwarf_next_cu_header_d()`

```
int dwarf_next_cu_header_d(
    Dwarf_debug dbg,
    Dwarf_Bool is_info,
    Dwarf_Unsigned *cu_header_length,
    Dwarf_Half     *version_stamp,
    Dwarf_Unsigned *abbrev_offset,
    Dwarf_Half     *address_size,
    Dwarf_Half     *offset_size,
    Dwarf_Half     *extension_size,
    Dwarf_Sig8     *signature,
    Dwarf_Unsigned *typeoffset,
    Dwarf_Unsigned *next_cu_header,
    Dwarf_Half     *header_cu_type,
    Dwarf_Error    *error);
```

The function `dwarf_next_cu_header_d()` operates on either the `.debug_info` section (if `is_info` is non-zero) or `.debug_types` section (if `is_info` is zero). It returns `DW_DLV_ERROR` if it fails, and `DW_DLV_OK` if it succeeds.

If it succeeds, `*next_cu_header` is set to the offset in the `.debug_info` section of the next compilation-unit header if it succeeds. On reading the last compilation-unit header in the `.debug_info` section it contains the size of the `.debug_info` or `debug_types` section. Beginning 22 April 2019 `next_cu_header` will not be used to return the offset if `next_cu_header` is null. Be cautious using a null argument unless you know that only a suitably recent version of `libdwarf` will be used.

The next call to `dwarf_next_cu_header_b()` returns `DW_DLV_NO_ENTRY` without reading a compilation-unit or setting `*next_cu_header`. Subsequent calls to `dwarf_next_cu_header()` repeat the cycle by reading the first compilation-unit and so on.

The other values returned through pointers are the values in the compilation-unit header. If any of `cu_header_length`, `version_stamp`, `abbrev_offset`, `address_size`, `offset_size`, `extension_size`, `signature`, or `typeoffset`, is `NULL`, the argument is ignored (meaning it is not an error to provide a `NULL` pointer for any or all of these arguments).

`cu_header_length` returns the length in bytes of the compilation unit header.

`version_stamp` returns the section version, which would be (for `.debug_info`) 2 for DWARF2, 3 for DWARF3, 4 for DWARF4, or 5 for DWARF5..

`abbrev_offset` returns the `.debug_abbrev` section offset of the abbreviations for this compilation unit.

`address_size` returns the size of an address in this compilation unit. Which is usually 4 or 8.

`offset_size` returns the size in bytes of an offset for the compilation unit. The offset size is 4 for 32bit dwarf and 8 for 64bit dwarf. This is the offset size in dwarf data, not

the address size inside the executable code. The offset size can be 4 even if embedded in a 64bit elf file (which is normal for 64bit elf), and can be 8 even in a 32bit elf file (which probably will never be seen in practice).

The `extension_size` pointer is only relevant if the `offset_size` pointer returns 8. The value is not normally useful but is returned through the pointer for completeness. The pointer `extension_size` returns 0 if the CU is MIPS/IRIX non-standard 64bit dwarf (MIPS/IRIX 64bit dwarf was created years before DWARF3 defined 64bit dwarf) and returns 4 if the dwarf uses the standard 64bit extension (the 4 is the size in bytes of the 0xffffffff in the initial length field which indicates the following 8 bytes in the `.debug_info` section are the real length). See the DWARF3 or DWARF4 standard, section 7.4.

The `signature` pointer is only relevant if the CU has a type signature, and if relevant the 8 byte type signature of the `.debug_types` CU header is assigned through the pointer.

The `typeoffset` pointer is only relevant the CU has a type signature if relevant the local offset within the CU of the the type offset the `.debug_types` entry represents is assigned through the pointer. The `typeoffset` matters because a `DW_AT_type` referencing the type unit may reference an inner type, such as a C++ class in a C++ namespace, but the type itself has the enclosing namespace in the `.debug_type` `type_unit`.

The `header_cu_type` pointer is applicable to all CU headers. The value returned through the pointer is either `DW_UT_compile` `DW_UT_partial` `DW_UT_type` and identifies the header type of this CU. In DWARF4 a `DW_UT_type` will be in `.debug_types`, but in DWARF5 these compilation units are in `.debug_info` and the Debug Fission (ie Split Dwarf) `.debug_info.dwo` sections.

#### 6.6.4 `dwarf_next_cu_header_c()`

```
int dwarf_next_cu_header_c(
    Dwarf_debug dbg,
    Dwarf_Bool is_info,
    Dwarf_Unsigned *cu_header_length,
    Dwarf_Half *version_stamp,
    Dwarf_Unsigned *abbrev_offset,
    Dwarf_Half *address_size,
    Dwarf_Half *offset_size,
    Dwarf_Half *extension_size,
    Dwarf_Sig8 *signature,
    Dwarf_Unsigned *typeoffset,
    Dwarf_Unsigned *next_cu_header,
    Dwarf_Error *error);
```

The function `dwarf_next_cu_header_c()` operates on the either the `.debug_info` section (if `is_info` is non-zero) or `.debug_types` section (if `is_info` is zero).

It operates exactly like `dwarf_next_cu_header_d()` but is missing the `header_type` field. This is kept for compatibility. All code using this should be changed to use `dwarf_next_cu_header_d()`

### 6.6.5 `dwarf_next_cu_header_b()`

```
int dwarf_next_cu_header_b(
    Dwarf_debug dbg,
    Dwarf_Unsigned *cu_header_length,
    Dwarf_Half      *version_stamp,
    Dwarf_Unsigned *abbrev_offset,
    Dwarf_Half      *address_size,
    Dwarf_Half      *offset_size,
    Dwarf_Half      *extension_size,
    Dwarf_Unsigned *next_cu_header,
    Dwarf_Error     *error);
```

This is obsolete as of October 2011 though supported.

The function `dwarf_next_cu_header_b()` operates on the `.debug_info` section. It operates exactly like `dwarf_next_cu_header_c()` but is missing the `signature`, and `typeoffset` fields. This is kept for compatibility. All code using this should be changed to use `dwarf_next_cu_header_c()`

### 6.6.6 `dwarf_next_cu_header()`

The following is the original form, missing the `offset_size`, `extension_size`, `signature`, and `typeoffset` fields in `dwarf_next_cu_header_c()`. This is kept for compatibility. All code using this should be changed to use `dwarf_next_cu_header_c()`

```
int dwarf_next_cu_header(
    Dwarf_debug dbg,
    Dwarf_Unsigned *cu_header_length,
    Dwarf_Half      *version_stamp,
    Dwarf_Unsigned *abbrev_offset,
    Dwarf_Half      *address_size,
    Dwarf_Unsigned *next_cu_header,
    Dwarf_Error     *error);
```

### 6.6.7 `dwarf_siblingof_b()`

```
int dwarf_siblingof_b(  
    Dwarf_Debug dbg,  
    Dwarf_Die die,  
    Dwarf_Bool is_info,  
    Dwarf_Die *return_sib,  
    Dwarf_Error *error)
```

The function `dwarf_siblingof_b()` returns `DW_DLV_ERROR` and sets the error pointer on error. If there is no sibling it returns `DW_DLV_NO_ENTRY`. When it succeeds, `dwarf_siblingof_b()` returns `DW_DLV_OK` and sets `*return_sib` to the `Dwarf_Die` descriptor of the sibling of `die`.

If `is_info` is non-zero then the `die` is assumed to refer to a `.debug_info` DIE. If `is_info` is zero then the `die` is assumed to refer to a `.debug_types` DIE. Note that the first call (the call that gets the compilation-unit DIE in a compilation unit) passes in a `NULL` `die` so having the caller pass in `is_info` is essential. And if `die` is non-`NULL` it is still essential for the call to pass in `is_info` set properly to reflect the section the DIE came from. The function `dwarf_get_die_infotypes_flag()` is of interest as it returns the proper `is_info` value from any non-`NULL` `die` pointer.

If `die` is `NULL`, the `Dwarf_Die` descriptor of the first die in the compilation-unit is returned. This die has the `DW_TAG_compile_unit`, `DW_TAG_partial_unit`, or `DW_TAG_type_unit` tag.

**Figure 11.** Example4 `dwarf_siblingof()`

```
void example4(Dwarf_Debug dbg,Dwarf_Die in_die,Dwarf_Bool is_info)  
{  
    Dwarf_Die return_sib = 0;  
    Dwarf_Error error = 0;  
    int res = 0;  
  
    /* in_die might be NULL or a valid Dwarf_Die */  
    res = dwarf_siblingof_b(dbg,in_die,is_info,&return_sib, &error);  
    if (res == DW_DLV_OK) {  
        /* Use return_sib here. */  
        dwarf_dealloc_die(return_sib);  
        /* This original form still works.  
        dwarf_dealloc(dbg, return_sib, DW_DLA_DIE);  
        */  
        /* return_sib is no longer usable for anything, we  
        ensure we do not use it accidentally with: */  
        return_sib = 0;  
    }  
}
```

### 6.6.8 dwarf\_siblingof()

```
int dwarf_siblingof(  
    Dwarf_Debug dbg,  
    Dwarf_Die die,  
    Dwarf_Die *return_sib,  
    Dwarf_Error *error)
```

`int dwarf_siblingof()` operates exactly the same as `int dwarf_siblingof_b()`, but `int dwarf_siblingof()` refers only to `.debug_info` DIEs.

### 6.6.9 dwarf\_child()

```
int dwarf_child(  
    Dwarf_Die die,  
    Dwarf_Die *return_kid,  
    Dwarf_Error *error)
```

The function `dwarf_child()` returns `DW_DLV_ERROR` and sets the error die on error. If there is no child it returns `DW_DLV_NO_ENTRY`. When it succeeds, `dwarf_child()` returns `DW_DLV_OK` and sets `*return_kid` to the `Dwarf_Die` descriptor of the first child of `die`. The function `dwarf_siblingof()` can be used with the return value of `dwarf_child()` to access the other children of `die`.

**Figure 12.** Example5 `dwarf_child()`

```
void example5(Dwarf_Die in_die)
{
    Dwarf_Die return_kid = 0;
    Dwarf_Error error = 0;
    int res = 0;

    res = dwarf_child(in_die, &return_kid, &error);
    if (res == DW_DLV_OK) {
        /* Use return_kid here. */
        dwarf_dealloc_die(return_kid);
        /* The original form of dealloc still works
           dwarf_dealloc(dbg, return_kid, DW_DLA_DIE);
           */
        /* return_die is no longer usable for anything, we
           ensure we do not use it accidentally with: */
        return_kid = 0;
    }
}
```

#### 6.6.10 dwarf\_offdie\_b()

```
int dwarf_offdie_b(
    Dwarf_Debug dbg,
    Dwarf_Off offset,
    Dwarf_Bool is_info,
    Dwarf_Die *return_die,
    Dwarf_Error *error)
```

The function `dwarf_offdie_b()` returns `DW_DLV_ERROR` and sets the error die on error. When it succeeds, `dwarf_offdie_b()` returns `DW_DLV_OK` and sets `*return_die` to the the `Dwarf_Die` descriptor of the debugging information entry at `offset` in the section containing debugging information entries i.e the `.debug_info` section. A return of `DW_DLV_NO_ENTRY` means that the `offset` in the section is of a byte containing all 0 bits, indicating that there is no abbreviation code. Meaning this 'die offset' is not the offset of a real die, but is instead an offset of a null die, a padding die, or of some random zero byte: this should not be returned in normal use.

It is the user's responsibility to make sure that `offset` is the start of a valid debugging information entry. The result of passing it an invalid offset could be chaos.

If `is_info` is non-zero the `offset` must refer to a `.debug_info` section offset. If `is_info` zero the `offset` must refer to a `.debug_types` section offset. Error returns or misleading values may result if the `is_info` flag or the `offset` value are incorrect.

**Figure 13.** Example6 `dwarf_offdie_b()`

```
void example6(Dwarf_Debug dbg,Dwarf_Off die_offset,Dwarf_Bool is_info)
{
    Dwarf_Error error = 0;
    Dwarf_Die return_die = 0;
    int res = 0;

    res = dwarf_offdie_b(dbg,die_offset,is_info,&return_die, &error);
    if (res == DW_DLV_OK) {
        /* Use return_die here. */
        dwarf_dealloc_die(return_die);
        /* The original form still works:
           dwarf_dealloc(dbg, return_die, DW_DLA_DIE);
        */
        /* return_die is no longer usable for anything, we
           ensure we do not use it accidentally with: */
        return_die = 0;
    } else {
        /* res could be NO ENTRY or ERROR, so no
           dealloc necessary. */
    }
}
```

### 6.6.11 dwarf\_offdie()

```
int dwarf_offdie(
    Dwarf_Debug dbg,
    Dwarf_Off offset,
    Dwarf_Die *return_die,
    Dwarf_Error *error)
```

The function `dwarf_offdie()` is obsolete, use `dwarf_offdie_b()` instead. The function is still supported in the library, but only references the `.debug_info` section.

### 6.6.12 dwarf\_validate\_die\_sibling()

```
int validate_die_sibling(
    Dwarf_Die sibling,
    Dwarf_Off *offset)
```

When used correctly in a depth-first walk of a DIE tree this function validates that any `DW_AT_sibling` attribute gives the same offset as the direct tree walk. That is the only purpose of this function.



The function `dwarf_validate_die_sibling()` returns `DW_DLV_OK` if the last die processed in a depth-first DIE tree walk was the same offset as generated by a call to `dwarf_siblingof()`. Meaning that the `DW_AT_sibling` attribute value, if any, was correct.

If the conditions are not met then `DW_DLV_ERROR` is returned and `*offset` is set to the offset in the `.debug_info` section of the last DIE processed. If the application prints the offset a knowledgeable user may be able to figure out what the compiler did wrong.

## 6.7 Debugging Information Entry Query Operations

These queries return specific information about debugging information entries or a descriptor that can be used on subsequent queries when given a `Dwarf_Die` descriptor. Note that some operations are specific to debugging information entries that are represented by a `Dwarf_Die` descriptor of a specific type. For example, not all debugging information entries contain an attribute having a name, so consequently, a call to `dwarf_diename()` using a `Dwarf_Die` descriptor that does not have a name attribute will return `DW_DLV_NO_ENTRY`. This is not an error, i.e. calling a function that needs a specific attribute is not an error for a die that does not contain that specific attribute.

There are several methods that can be used to obtain the value of an attribute in a given die:

1. Call `dwarf_hasattr()` to determine if the debugging information entry has the attribute of interest prior to issuing the query for information about the attribute.
2. Supply an `error` argument, and check its value after the call to a query indicates an unsuccessful return, to determine the nature of the problem. The `error` argument will indicate whether an error occurred, or the specific attribute needed was missing in that die.
3. Arrange to have an error handling function invoked upon detection of an error (see `dwarf_init()`).
4. Call `dwarf_attrlist()` and iterate through the returned list of attributes, dealing with each one as appropriate.

### 6.7.1 `dwarf_get_die_infotypes_flag()`

`Dwarf_Bool dwarf_get_die_infotypes_flag(Dwarf_Die die)`

The function `dwarf_tag()` returns the section flag indicating which section the DIE

originates from. If the returned value is non-zero the DIE originates from the `.debug_info` section. If the returned value is zero the DIE originates from the `.debug_types` section.

### 6.7.2 `dwarf_tag()`

```
int dwarf_tag(  
    Dwarf_Die die,  
    Dwarf_Half *tagval,  
    Dwarf_Error *error)
```

The function `dwarf_tag()` returns the tag of `die` through the pointer `tagval` if it succeeds. It returns `DW_DLV_OK` if it succeeds. It returns `DW_DLV_ERROR` on error.

### 6.7.3 `dwarf_dieoffset()`

```
int dwarf_dieoffset(  
    Dwarf_Die die,  
    Dwarf_Off *return_offset,  
    Dwarf_Error *error)
```

When it succeeds, the function `dwarf_dieoffset()` returns `DW_DLV_OK` and sets `*return_offset` to the position of `die` in the section containing debugging information entries (the `return_offset` is a section-relative offset). In other words, it sets `return_offset` to the offset of the start of the debugging information entry described by `die` in the section containing dies i.e `.debug_info`. It returns `DW_DLV_ERROR` on error.

### 6.7.4 `dwarf_addr_form_is_indexed()`

`dwarf_addr_form_is_indexed(form)` is a utility function to make it simple to determine if a form is one of the indexed forms (there are several such in DWARF5). See DWARF5 section 7.5.5 Classes and Forms for more information.

```
int dwarf_addr_form_is_indexed(Dwarf_Half form);
```

It returns `TRUE` if the form is one of the indexed address forms (such as `DW_FORM_addrx1`) and `FALSE` otherwise.

### 6.7.5 `dwarf_debug_addr_index_to_addr()`

```
int dwarf_debug_addr_index_to_addr(Dwarf_Die /*die*/,
    Dwarf_Unsigned index,
    Dwarf_Addr *return_addr,
    Dwarf_Error *error);
```

Attributes with form DW\_FORM\_addrx, the operation DW\_OP\_addrx, or certain of the split-dwarf location list entries give an index value to a machine address in the .debug\_addr section (which is always in .debug\_addr even when the form/operation are in a split dwarf .dwo section).

On successful return this function turns such an index into a target address value through the pointer return\_addr .

If there is an error this may return DW\_DLV\_ERROR and it will have returned an error through \*error.

If there is no available .debug\_addr section this may return DW\_DLV\_NO\_ENTRY.

### 6.7.6 dwarf\_die\_CU\_offset()

```
int dwarf_die_CU_offset(
    Dwarf_Die die,
    Dwarf_Off *return_offset,
    Dwarf_Error *error)
```

The function dwarf\_die\_CU\_offset() is similar to dwarf\_dieoffset(), except that it puts the offset of the DIE represented by the Dwarf\_Die die, from the start of the compilation-unit that it belongs to rather than the start of .debug\_info (the return\_offset is a CU-relative offset).

### 6.7.7 dwarf\_die\_offsets()

```
int dwarf_die_offsets(
    Dwarf_Die die,
    Dwarf_Off *global_off,
    Dwarf_Off *cu_off,
    Dwarf_Error *error)
```

The function dwarf\_die\_offsets() is a combination of dwarf\_dieoffset() and dwarf\_die\_cu\_offset() in that it returns both the global .debug\_info offset and the CU-relative offset of the die in a single call.

### 6.7.8 dwarf\_ptr\_CU\_offset()

```
int dwarf_ptr_CU_offset(  
    Dwarf_CU_Context cu_context,  
    Dwarf_Byte_ptr di_ptr ,  
    Dwarf_Off *cu_off)
```

Given a valid CU context pointer and a pointer into that CU context, the function `dwarf_ptr_CU_offset()` returns `DW_DLV_OK` and sets `*cu_off` to the CU-relative (local) offset in that CU.

### 6.7.9 dwarf\_CU\_dieoffset\_given\_die()

```
int dwarf_CU_dieoffset_given_die(  
    Dwarf_Die given_die,  
    Dwarf_Off *return_offset,  
    Dwarf_Error *error)
```

The function `dwarf_CU_dieoffset_given_die()` is similar to `dwarf_die_CU_offset()`, except that it puts the global offset of the CU DIE owning `given_die` of `.debug_info` (the `return_offset` is a global section offset).

This is useful when processing a DIE tree and encountering an error or other surprise in a DIE, as the `return_offset` can be passed to `dwarf_offdie_b()` to return a pointer to the CU die of the CU owning the `given_die` passed to `dwarf_CU_dieoffset_given_die()`. The consumer can extract information from the CU die and the `given_die` (in the normal way) and print it.

An example (a snippet) of code using this function follows. It assumes that `in_die` is a DIE in `.debug_info` that, for some reason, you have decided needs CU context printed (assuming `print_die_data` does some reasonable printing).

**Figure 14.** Example7 dwarf\_CU\_dieoffset\_given\_die()

```
void example7(Dwarf_Debug dbg, Dwarf_Die in_die, Dwarf_Bool is_info)
{
    int res = 0;
    Dwarf_Off cudieoff = 0;
    Dwarf_Die cudie = 0;
    Dwarf_Error error = 0;

    res = dwarf_CU_dieoffset_given_die(in_die, &cudieoff, &error);
    if(res != DW_DLV_OK) {
        /* FAIL */
        return;
    }
    res = dwarf_offdie_b(dbg, cudieoff, is_info, &cudie, &error);
    if(res != DW_DLV_OK) {
        /* FAIL */
        return;
    }
    /* do something with cu_die */
    dwarf_dealloc_die(cudie);
    /* The original form still works.
       dwarf_dealloc(dbg, cudie, DW_DLA_DIE);
    */
}
y
```

#### 6.7.10 dwarf\_die\_CU\_offset\_range()

```
int dwarf_die_CU_offset_range(
    Dwarf_Die die,
    Dwarf_Off *cu_global_offset,
    Dwarf_Off *cu_length,
    Dwarf_Error *error)
```

The function `dwarf_die_CU_offset_range()` returns the offset of the beginning of the CU and the length of the CU. The offset and length are of the entire CU that this DIE is a part of. It is used by `dwarfdump` (for example) to check the validity of offsets. Most applications will have no reason to call this function.

#### 6.7.11 dwarf\_diename()

```
int dwarf_diename(  
    Dwarf_Die die,  
    char ** return_name,  
    Dwarf_Error *error)
```

When it succeeds, the function `dwarf_diename()` returns `DW_DLV_OK` and sets `*return_name` to a pointer to a null-terminated string of characters that represents the name attribute (`DW_AT_name`) of `die`.

The storage pointed to by a successful return of `dwarf_diename()` should not be freed as the text is a string in static memory (for some error cases) or a string residing in a DWARF data section.

Up to March 2020 this document said that `dwarf_dealloc` with `DW_DLA_STRING` should be applied to the string returned through the pointer. That was always incorrect. However, doing the `dwarf_dealloc(dbg,xxx,DW_DLA_STRING)` that was previously called for does not result in any error (`dwarf_dealloc` avoids freeing strings like this).

It returns `DW_DLV_NO_ENTRY` if `die` does not have a name attribute. It returns `DW_DLV_ERROR` if an error occurred.

### 6.7.12 dwarf\_die\_text()

```
int dwarf_die_text(  
    Dwarf_Die die,  
    Dwarf_Half attrnum,  
    char ** return_name,  
    Dwarf_Error *error)
```

When it succeeds, the function `dwarf_die_text()` returns `DW_DLV_OK` and sets `*return_name` to a pointer to a null-terminated string of characters that represents a string-value attribute of `die` if an attribute `attrnum` is present.

The storage pointed to by a successful return of `dwarf_die_text()` must never be freed, the string is in the DWARF data and is not dynamically allocated.

As of March 2020 the description here has been corrected. `dwarf_dealloc()` should never have been applied to a string returned by `dwarf_die_text()`.

It returns `DW_DLV_NO_ENTRY` if `die` does not have the attribute `attrnum`. It returns `DW_DLV_ERROR` if an error occurred.

### 6.7.13 dwarf\_die\_abbrev\_code()

```
int dwarf_die_abbrev_code( Dwarf_Die die)
```

The function returns the abbreviation code of the DIE. That is, it returns the abbreviation "index" into the abbreviation table for the compilation unit of which the DIE is a part. It

cannot fail. No errors are possible. The pointer `die()` must not be NULL.

#### 6.7.14 `dwarf_die_abbrev_children_flag()`

```
int dwarf_die_abbrev_children_flag( Dwarf_Die die,
    Dwarf_Half *has_child)
```

The function returns the has-children flag of the die passed in through the `*has_child` passed in and returns DW\_DLV\_OK on success. A non-zero value of `*has_child` means the die has children.

On failure it returns DW\_DLV\_ERROR.

The function was developed to let consumer code do better error reporting in some circumstances, it is not generally needed.

#### 6.7.15 `dwarf_die_abbrev_global_offset()`

```
int dwarf_die_abbrev_global_offset(Dwarf_Die die,
    Dwarf_Off * abbrev_offset,
    Dwarf_Unsigned * abbrev_count,
    Dwarf_Error* error);
```

The function allows more detailed printing of abbreviation data. It is handy for analyzing abbreviations but is not normally needed by applications. The function first appears in March 2016.

On success the function returns DW\_DLV\_OK and sets `*abbrev_offset` to the global offset in the `.debug_abbrev` section of the abbreviation. It also sets `*abbrev_count` to the number of attribute/form pairs in the abbreviation entry. It is possible, though unusual, for the count to be zero (meaning there is abbreviation instance and a TAG instance which have no attributes).

On failure it returns DW\_DLV\_ERROR and sets `*error`

It should never return DW\_DLV\_NO\_ENTRY, but callers should allow for that possibility..

#### 6.7.16 `dwarf_get_version_of_die()`

```
int dwarf_get_version_of_die(Dwarf_Die die,
    Dwarf_Half *version,
    Dwarf_Half *offset_size)
```

The function returns the CU context version through `*version` and the CU context offset-size through `*offset_size` and returns DW\_DLV\_OK on success.

In case of error, the only errors possible involve an inappropriate NULL die pointer so no Dwarf\_Debug pointer is available. Therefore setting a Dwarf\_Error would not be very meaningful (there is no Dwarf\_Debug to attach it to). The function returns DW\_DLV\_ERROR on error.

The values returned through the pointers are the values two arguments to dwarf\_get\_form\_class() requires.

### 6.7.17 dwarf\_attrlist()

```
int dwarf_attrlist(  
    Dwarf_Die die,  
    Dwarf_Attribute** attrbuf,  
    Dwarf_Signed *attrcount,  
    Dwarf_Error *error)
```

When it returns DW\_DLV\_OK, the function dwarf\_attrlist() sets attrbuf to point to an array of Dwarf\_Attribute descriptors corresponding to each of the attributes in die, and returns the number of elements in the array through attrcount. DW\_DLV\_NO\_ENTRY is returned if the count is zero (no attrbuf is allocated in this case). DW\_DLV\_ERROR is returned on error. On a successful return from dwarf\_attrlist(), each of the Dwarf\_Attribute descriptors should be individually freed using dwarf\_dealloc() with the allocation type DW\_DLA\_ATTR, followed by free-ing the list pointed to by \*attrbuf using dwarf\_dealloc() with the allocation type DW\_DLA\_LIST, when no longer of interest (see dwarf\_dealloc()).

Freeing the attrlist:

**Figure 15.** Example8 dwarf\_attrlist() free



```
void example8(Dwarf_Debug dbg, Dwarf_Die somedie)
{
    Dwarf_Signed atcount = 0;
    Dwarf_Attribute *atlist = 0;
    Dwarf_Error error = 0;
    int errv = 0;

    errv = dwarf_attrlist(somedie, &atlist, &atcount, &error);
    if (errv == DW_DLV_OK) {
        Dwarf_Signed i = 0;

        for (i = 0; i < atcount; ++i) {
            /* use atlist[i] */
            dwarf_dealloc_attribute(atlist[i]);
            /* The original form still works.
               dwarf_dealloc(dbg, atlist[i], DW_DLA_ATTR);
            */
        }
        dwarf_dealloc(dbg, atlist, DW_DLA_LIST);
    }
}
```

### 6.7.18 dwarf\_hasattr()

```
int dwarf_hasattr(
    Dwarf_Die die,
    Dwarf_Half attr,
    Dwarf_Bool *return_bool,
    Dwarf_Error *error)
```

When it succeeds, the function `dwarf_hasattr()` returns `DW_DLV_OK` and sets `*return_bool` to *non-zero* if `die` has the attribute `attr` and *zero* otherwise. If it fails, it returns `DW_DLV_ERROR`.

### 6.7.19 dwarf\_attr()

```
int dwarf_attr(
    Dwarf_Die die,
    Dwarf_Half attr,
    Dwarf_Attribute *return_attr,
    Dwarf_Error *error)
```

When it returns `DW_DLV_OK`, the function `dwarf_attr()` sets `*return_attr` to the `Dwarf_Attribute` descriptor of `die` having the attribute `attr`. It returns `DW_DLV_NO_ENTRY` if `attr` is not contained in `die`. It returns `DW_DLV_ERROR` if

an error occurred.

### 6.7.20 dwarf\_lowpc()

```
int dwarf_lowpc(  
    Dwarf_Die      die,  
    Dwarf_Addr     * return_lowpc,  
    Dwarf_Error    * error)
```

The function `dwarf_lowpc()` returns `DW_DLV_OK` and sets `*return_lowpc` to the low program counter value associated with the die descriptor if die represents a debugging information entry with the `DW_AT_low_pc` attribute. It returns `DW_DLV_NO_ENTRY` if die does not have this attribute. It returns `DW_DLV_ERROR` if an error occurred.

### 6.7.21 dwarf\_highpc\_b()

```
int dwarf_highpc_b(  
    Dwarf_Die      die,  
    Dwarf_Addr     * return_highpc,  
    Dwarf_Half     * return_form*,  
    enum Dwarf_Form_Class * return_class*,  
    Dwarf_Error    *error)
```

The function `dwarf_highpc_b()` returns `DW_DLV_OK` and sets `*return_highpc` to the value of the `DW_AT_high_pc` attribute.

It also sets `*return_form` to the FORM of the attribute. Beginning 22 April 2019 `return_form` will not be used to return the form class if `return_form` is null. Be cautious using a null argument unless you know that only a suitably recent version of `libdwarf` will be used.

It sets `*return_class` to the form class of the attribute. Beginning 22 April 2019 `return_class` will not be used to return the form class if `return_class` is null. Be cautious using a null argument unless you know that only a suitably recent version of `libdwarf` will be used.

If the form class returned is `DW_FORM_CLASS_ADDRESS` the `return_highpc` is an actual pc address (1 higher than the address of the last pc in the address range).. If the form class returned is `DW_FORM_CLASS_CONSTANT` the `return_highpc` is an offset from the value of the the DIE's low PC address (see DWARF4 section 2.17.2 Contiguous Address Range).

It returns DW\_DLV\_NO\_ENTRY if die does not have the DW\_AT\_high\_pc attribute.

It returns DW\_DLV\_ERROR if an error occurred.

### 6.7.22 dwarf\_highpc()

```
int dwarf_highpc(  
    Dwarf_Die die,  
    Dwarf_Addr * return_highpc,  
    Dwarf_Error *error)
```

The function dwarf\_highpc() returns DW\_DLV\_OK and sets \*return\_highpc the high program counter value associated with the die descriptor if die represents a debugging information entry with the DW\_AT\_high\_pc attribute and the form is DW\_FORM\_addr (meaning the form is of class address).

This function is useless for a DW\_AT\_high\_pc which is encoded as a constant (which was first possible in DWARF4).

It returns DW\_DLV\_NO\_ENTRY if die does not have this attribute.

It returns DW\_DLV\_ERROR if an error occurred or if the form is not of class address.

### 6.7.23 dwarf\_dietype\_offset()

```
int dwarf_dietype_offset(Dwarf_Die /*die*/,  
    Dwarf_Off * /*return_off*/,  
    Dwarf_Error * /*error*/);
```

On success the function dwarf\_dietype\_offset() returns the offset referred to by DW\_AT\_type attribute of die.

DW\_DLV\_NO\_ENTRY is returned if the die has no DW\_AT\_type attribute.

DW\_DLV\_ERROR is returned if an error is detected.

This feature was introduced in February 2016.

### 6.7.24 dwarf\_offset\_list()

```
int dwarf_offset_list(Dwarf_Debug dbg,
    Dwarf_Off      offset,
    Dwarf_Bool     is_info,
    Dwarf_Off      ** offbuf,
    Dwarf_Unsigned * offcnt,
    Dwarf_Error    * error);
```

On success The function `dwarf_offset_list()` returns `DW_DLV_OK` and sets `*offbuf` to point to an array of the offsets of the direct children of the die at `offset`. It sets `*offcnt` to point to the count of entries in the `offset` array

In case of error it returns `DW_DLV_OK`.

It does not return `DW_DLV_NO_ENTRY` but callers should allow for that possibility anyway.

This feature was introduced in March 2016.

Freeing the `offset_list` is done as follows.:

**Figure 16.** Example `offset_list dwarf_offset_list()` free

```
void exampleoffset_list(Dwarf_Debug dbg, Dwarf_Off dieoffset,
    Dwarf_Bool is_info)
{
    Dwarf_Unsigned offcnt = 0;
    Dwarf_Off *offbuf = 0;
    Dwarf_Error error = 0;
    int errv = 0;

    errv = dwarf_offset_list(dbg, dieoffset, is_info,
        &offbuf, &offcnt, &error);
    if (errv == DW_DLV_OK) {
        Dwarf_Unsigned i = 0;

        for (i = 0; i < offcnt; ++i) {
            /* use offbuf[i] */
        }
        dwarf_dealloc(dbg, offbuf, DW_DLA_LIST);
    }
}
```

### 6.7.25 dwarf\_bytesize()

```
Dwarf_Signed dwarf_bytesize(  
    Dwarf_Die      die,  
    Dwarf_Unsigned *return_size,  
    Dwarf_Error    *error)
```

When it succeeds, `dwarf_bytesize()` returns `DW_DLV_OK` and sets `*return_size` to the number of bytes needed to contain an instance of the aggregate debugging information entry represented by `die`. It returns `DW_DLV_NO_ENTRY` if `die` does not contain the byte size attribute `DW_AT_byte_size`. It returns `DW_DLV_ERROR` if an error occurred.

### 6.7.26 dwarf\_bitsize()

```
int dwarf_bitsize(  
    Dwarf_Die die,  
    Dwarf_Unsigned *return_size,  
    Dwarf_Error *error)
```

When it succeeds, `dwarf_bitsize()` returns `DW_DLV_OK` and sets `*return_size` to the number of bits occupied by the bit field value that is an attribute of the given `die`. It returns `DW_DLV_NO_ENTRY` if `die` does not contain the bit size attribute `DW_AT_bit_size`. It returns `DW_DLV_ERROR` if an error occurred.

### 6.7.27 dwarf\_bitoffset()

```
int dwarf_bitoffset(  
    Dwarf_Die die,  
    Dwarf_Unsigned *return_size,  
    Dwarf_Error *error)
```

When it succeeds, `dwarf_bitoffset()` returns `DW_DLV_OK` and sets `*return_size` to the number of bits to the left of the most significant bit of the bit field value. This bit offset is not necessarily the net bit offset within the structure or class, since `DW_AT_data_member_location` may give a byte offset to this DIE and the bit offset returned through the pointer does not include the bits in the byte offset. It returns `DW_DLV_NO_ENTRY` if `die` does not contain the bit offset attribute `DW_AT_bit_offset`. It returns `DW_DLV_ERROR` if an error occurred.

### 6.7.28 dwarf\_srclang()

```
int dwarf_srclang(  
    Dwarf_Die die,  
    Dwarf_Unsigned *return_lang,  
    Dwarf_Error *error)
```

When it succeeds, `dwarf_srclang()` returns `DW_DLV_OK` and sets `*return_lang` to a code indicating the source language of the compilation unit represented by the descriptor `die`. It returns `DW_DLV_NO_ENTRY` if `die` does not represent a source file debugging information entry (i.e. contain the attribute `DW_AT_language`). It returns `DW_DLV_ERROR` if an error occurred.

### 6.7.29 dwarf\_arrayorder()

```
int dwarf_arrayorder(  
    Dwarf_Die die,  
    Dwarf_Unsigned *return_order,  
    Dwarf_Error *error)
```

When it succeeds, `dwarf_arrayorder()` returns `DW_DLV_OK` and sets `*return_order` a code indicating the ordering of the array represented by the descriptor `die`. It returns `DW_DLV_NO_ENTRY` if `die` does not contain the array order attribute `DW_AT_ordering`. It returns `DW_DLV_ERROR` if an error occurred.

## 6.8 Attribute Queries

Based on the attributes form, these operations are concerned with returning uninterpreted attribute data. Since it is not always obvious from the return value of these functions if an error occurred, one should always supply an `error` parameter or have arranged to have an error handling function invoked (see `dwarf_init()`) to determine the validity of the returned value and the nature of any errors that may have occurred.

A `Dwarf_Attribute` descriptor describes an attribute of a specific die. Thus, each `Dwarf_Attribute` descriptor is implicitly associated with a specific die.

### 6.8.1 dwarf\_hasform()

```
int dwarf_hasform(  
    Dwarf_Attribute attr,  
    Dwarf_Half form,  
    Dwarf_Bool *return_hasform,  
    Dwarf_Error *error)
```

The function `dwarf_hasform()` returns `DW_DLV_OK` and puts a *non-zero* value in the `*return_hasform` boolean if the attribute represented by the

Dwarf\_Attribute descriptor `attr` has the attribute form `form`. If the attribute does not have that form `zero` is put into `*return_hasform`. `DW_DLV_ERROR` is returned on error.

### 6.8.2 dwarf\_whatform()

```
int dwarf_whatform(  
    Dwarf_Attribute attr,  
    Dwarf_Half      *return_form,  
    Dwarf_Error     *error)
```

When it succeeds, `dwarf_whatform()` returns `DW_DLV_OK` and sets `*return_form` to the attribute form code of the attribute represented by the Dwarf\_Attribute descriptor `attr`. It returns `DW_DLV_ERROR` on error.

An attribute using `DW_FORM_indirect` effectively has two forms. This function returns the 'final' form for `DW_FORM_indirect`, not the `DW_FORM_indirect` itself. This function is what most applications will want to call.

### 6.8.3 dwarf\_whatform\_direct()

```
int dwarf_whatform_direct(  
    Dwarf_Attribute attr,  
    Dwarf_Half      *return_form,  
    Dwarf_Error     *error)
```

When it succeeds, `dwarf_whatform_direct()` returns `DW_DLV_OK` and sets `*return_form` to the attribute form code of the attribute represented by the Dwarf\_Attribute descriptor `attr`. It returns `DW_DLV_ERROR` on error. An attribute using `DW_FORM_indirect` effectively has two forms. This returns the form 'directly' in the initial form field. That is, it returns the 'initial' form of the attribute.

So when the form field is `DW_FORM_indirect` this call returns the `DW_FORM_indirect` form, which is sometimes useful for dump utilities.

It is confusing that the `_direct()` function returns `DW_FORM_indirect` if an indirect form is involved. Just think of this as returning the initial form the first form value seen for the attribute, which is also the final form unless the initial form is `DW_FORM_indirect`.

### 6.8.4 dwarf\_whatattr()

```
int dwarf_whatattr(  
    Dwarf_Attribute attr,  
    Dwarf_Half      *return_attr,  
    Dwarf_Error     *error)
```

When it succeeds, `dwarf_whatattr()` returns `DW_DLV_OK` and sets `*return_attr` to the attribute code represented by the `Dwarf_Attribute` descriptor `attr`. It returns `DW_DLV_ERROR` on error.

### 6.8.5 dwarf\_formref()

```
int dwarf_formref(  
    Dwarf_Attribute attr,  
    Dwarf_Off      *return_offset,  
    Dwarf_Error     *error)
```

When it succeeds, `dwarf_formref()` returns `DW_DLV_OK` and sets `*return_offset` to the CU-relative offset represented by the descriptor `attr` if the form of the attribute belongs to the `REFERENCE` class. `attr` must be a CU-local reference, not form `DW_FORM_ref_addr` and not `DW_FORM_sec_offset`. It is an error for the form to not belong to the `REFERENCE` class. It returns `DW_DLV_ERROR` on error.

Beginning November 2010: All `DW_DLV_ERROR` returns set `*return_offset`. Most errors set `*return_offset` to zero, but for error `DW_DLE_ATTR_FORM_OFFSET_BAD` the function sets `*return_offset` to the invalid offset (which allows the caller to print a more detailed error message).

See also `dwarf_global_formref` below.

### 6.8.6 dwarf\_global\_formref()

```
int dwarf_global_formref(  
    Dwarf_Attribute attr,  
    Dwarf_Off      *return_offset,  
    Dwarf_Error     *error)
```

When it succeeds, `dwarf_global_formref()` returns `DW_DLV_OK` and sets `*return_offset` to the section-relative offset represented by the descriptor `attr` if the form of the attribute belongs to the `REFERENCE` or other section-references classes.

`attr` can be any legal `REFERENCE` class form plus `DW_FORM_ref_addr` or `DW_FORM_sec_offset`. It is an error for the form to not belong to one of the reference classes. It returns `DW_DLV_ERROR` on error. See also `dwarf_formref`



above.

The caller must determine which section the offset returned applies to. The function `dwarf_get_form_class()` is useful to determine the applicable section.

The function converts CU relative offsets from forms such as `DW_FORM_ref4` into global section offsets.

### 6.8.7 `dwarf_convert_to_global_offset()`

```
int dwarf_convert_to_global_offset(  
    Dwarf_Attribute attr,  
    Dwarf_Off      offset,  
    Dwarf_Off      *return_offset,  
    Dwarf_Error    *error)
```

When it succeeds, `dwarf_convert_to_global_offset()` returns `DW_DLV_OK` and sets `*return_offset` to the section-relative offset represented by the cu-relative offset `offset` if the form of the attribute belongs to the `REFERENCE` class. `attr` must be a CU-local reference (DWARF class `REFERENCE`) or form `DW_FORM_ref_addr` and the `attr` must be directly relevant for the calculated `*return_offset` to mean anything.

The function returns `DW_DLV_ERROR` on error.

The function is not strictly necessary but may be a convenience for attribute printing in case of error.

### 6.8.8 `dwarf_formaddr()`

```
int dwarf_formaddr(  
    Dwarf_Attribute attr,  
    Dwarf_Addr      * return_addr,  
    Dwarf_Error     *error)
```

When it succeeds, `dwarf_formaddr()` returns `DW_DLV_OK` and sets `*return_addr` to the address represented by the descriptor `attr` if the form of the attribute belongs to the `ADDRESS` class. It is an error for the form to not belong to this class. It returns `DW_DLV_ERROR` on error.

One possible error that can arise (in a `.dwo` object file or a `.dwp` package file) is `DW_DLE_MISSING_NEEDED_DEBUG_ADDR_SECTION`. Such an error means that the `.dwo` or `.dwp` file is missing the `.debug_addr` section. When opening a `.dwo` object file or a `.dwp` package file one should also open the corresponding executable and

use `dwarf_set_tied_dbg()` to associate the objects before calling `dwarf_formaddr()`.

### H 3 "dwarf\_get\_debug\_addr\_index()"

```
int dwarf_get_debug_addr_index(  
    Dwarf_Attribute attr,  
    Dwarf_Unsigned * return_index,  
    Dwarf_Error *error)
```

`dwarf_get_debug_addr_index()` is only valid on attributes with form `DW_FORM_GNU_addr_index` or `DW_FORM_addrx`.

The function makes it possible to print the index from a dwarf dumper program.

When it succeeds, `dwarf_get_debug_addr_index()` returns `DW_DLV_OK` and sets `*return_index` to the attribute's index (into the `.debug_addr` section).

It returns `DW_DLV_ERROR` on error.

## 6.8.9 dwarf\_get\_debug\_str\_index()

```
int dwarf_get_debug_str_index(  
    Dwarf_Attribute attr,  
    Dwarf_Unsigned * return_index,  
    Dwarf_Error * error);
```

For an attribute with form `DW_FORM_strx` or `DW_FORM_GNU_str_index` this function retrieves the index (which refers to a `.debug_str_offsets` section in this `.dwo`).

If successful, the function `dwarf_get_debug_str_index()` returns `DW_DLV_OK` and returns the index through the `return_index()` pointer.

If the passed in attribute does not have this form or there is no valid compilation unit context for the attribute the function returns `DW_DLV_ERROR`.

`DW_DLV_NO_ENTRY` is not returned.

## 6.8.10 dwarf\_formflag()

```
int dwarf_formflag(  
    Dwarf_Attribute attr,  
    Dwarf_Bool * return_bool,  
    Dwarf_Error *error)
```

When it succeeds, `dwarf_formflag()` returns `DW_DLV_OK` and sets `*return_bool` to the (one unsigned byte) flag value. Any non-zero value means true. A zero value means false.

Before 29 November 2012 this would only return 1 or zero through the pointer, but that was always a strange thing to do. The DWARF specification has always been clear that any non-zero value means true. The function should report the value found truthfully, and now it does.

It returns `DW_DLV_ERROR` on error or if the `attr` does not have form flag.

#### 6.8.11 dwarf\_formudata()

```
int dwarf_formudata(  
    Dwarf_Attribute attr,  
    Dwarf_Unsigned * return_uvalue,  
    Dwarf_Error * error)
```

The function `dwarf_formudata()` returns `DW_DLV_OK` and sets `*return_uvalue` to the `Dwarf_Unsigned` value of the attribute represented by the descriptor `attr` if the form of the attribute belongs to the `CONSTANT` class. It is an error for the form to not belong to this class. It returns `DW_DLV_ERROR` on error.

Never returns `DW_DLV_NO_ENTRY`.

For DWARF2 and DWARF3, `DW_FORM_data4` and `DW_FORM_data8` are possibly class `CONSTANT`, and for DWARF4 and later they are definitely class `CONSTANT`.

#### 6.8.12 dwarf\_formsdata()

```
int dwarf_formsdata(  
    Dwarf_Attribute attr,  
    Dwarf_Signed * return_svalue,  
    Dwarf_Error *error)
```

The function `dwarf_formsdata()` returns `DW_DLV_OK` and sets `*return_svalue` to the `Dwarf_Signed` value of the attribute represented by the descriptor `attr` if the form of the attribute belongs to the `CONSTANT` class. It is an error for the form to not belong to this class. If the size of the data attribute referenced is

smaller than the size of the `Dwarf_Signed` type, its value is sign extended. It returns `DW_DLV_ERROR` on error.

Never returns `DW_DLV_NO_ENTRY`.

For DWARF2 and DWARF3, `DW_FORM_data4` and `DW_FORM_data8` are possibly class `CONSTANT`, and for DWARF4 and later they are definitely class `CONSTANT`.

### 6.8.13 dwarf\_formblock()

```
int dwarf_formblock(  
    Dwarf_Attribute attr,  
    Dwarf_Block ** return_block,  
    Dwarf_Error * error)
```

The function `dwarf_formblock()` returns `DW_DLV_OK` and sets `*return_block` to a pointer to a `Dwarf_Block` structure containing the value of the attribute represented by the descriptor `attr` if the form of the attribute belongs to the `BLOCK` class. It is an error for the form to not belong to this class. The storage pointed to by a successful return of `dwarf_formblock()` should be freed using the allocation type `DW_DLA_BLOCK`, when no longer of interest (see `dwarf_dealloc()`). It returns `DW_DLV_ERROR` on error.

### 6.8.14 dwarf\_formstring()

```
int dwarf_formstring(  
    Dwarf_Attribute attr,  
    char ** return_string,  
    Dwarf_Error *error)
```

The function `dwarf_formstring()` returns `DW_DLV_OK` and sets `*return_string` to a pointer to a null-terminated string containing the value of the attribute represented by the descriptor `attr` if the form of the attribute belongs to the `STRING` class. It is an error for the form to not belong to this class. The storage pointed to by a successful return of `dwarf_formstring()` should not be freed. The pointer points into existing DWARF memory and the pointer becomes stale/invalid after a call to `dwarf_finish`. `dwarf_formstring()` returns `DW_DLV_ERROR` on error.

### 6.8.15 dwarf\_formsig8()

```
int dwarf_formsig8(  
    Dwarf_Attribute attr,  
    Dwarf_Sig8 * return_sig8,  
    Dwarf_Error * error)
```

The function `dwarf_formsig8()` returns `DW_DLV_OK` and copies the 8 byte signature to a `Dwarf_Sig8` structure provided by the caller if the form of the attribute is of form `DW_FORM_ref_sig8` ( a member of the `REFERENCE` class). It is an error for the form to be anything but `DW_FORM_ref_sig8`. It returns `DW_DLV_ERROR` on error.

This form is used to refer to a type unit.

### 6.8.16 dwarf\_formexprloc()

```
int dwarf_formexprloc(  
    Dwarf_Attribute attr,  
    Dwarf_Unsigned * return_exprlen,  
    Dwarf_Ptr * block_ptr,  
    Dwarf_Error * error)
```

The function `dwarf_formexprloc()` returns `DW_DLV_OK` and sets the two values thru the pointers to the length and bytes of the `DW_FORM_exprloc` entry if the form of the attribute is of form `DW_FORM_exprloc`. It is an error for the form to be anything but `DW_FORM_exprloc`. It returns `DW_DLV_ERROR` on error.

On success the value set through the `return_exprlen` pointer is the length of the location expression. On success the value set through the `block_ptr` pointer is a pointer to the bytes of the location expression itself.

### 6.8.17 dwarf\_get\_form\_class()

```
enum Dwarf_Form_Class dwarf_get_form_class(  
    Dwarf_Half dwversion,  
    Dwarf_Half attrnum,  
    Dwarf_Half offset_size,  
    Dwarf_Half form)
```

The function is just for the convenience of `libdwarf` clients that might wish to categorize the `FORM` of a particular attribute. The `DWARF` specification divides `FORMs` into classes in Chapter 7 and this function figures out the correct class for a form.

The `dwversion` passed in shall be the dwarf version of the compilation unit involved (2 for `DWARF2`, 3 for `DWARF3`, 4 for `DWARF 4`). The `attrnum` passed in shall be the attribute number of the attribute involved (for example, `DW_AT_name` ). The `offset_size` passed in shall be the length of an offset in the current compilation unit

(4 for 32bit dwarf or 8 for 64bit dwarf). The form passed in shall be the attribute form number. If form DW\_FORM\_indirect is passed in DW\_FORM\_CLASS\_UNKNOWN will be returned as this form has no defined 'class'.

When it returns DW\_FORM\_CLASS\_UNKNOWN the function is simply saying it could not determine the correct class given the arguments presented. Some user-defined attributes might have this problem.

The function dwarf\_get\_version\_of\_die() may be helpful in filling out arguments for a call to dwarf\_get\_form\_class().

### 6.8.18 dwarf\_discr\_list()

```
int dwarf_discr_list(  
    Dwarf_Debug dbg,  
    Dwarf_Small * blockpointer,  
    Dwarf_Unsigned blocklen,  
    Dwarf_Dsc_Head * dsc_head_out,  
    Dwarf_Unsigned * dsc_array_length_out,  
    Dwarf_Error * error)  
Dwarf_Error *error)
```

When it succeeds, dwarf\_discr\_list() returns DW\_DLV\_OK and sets \*dsc\_head\_out to a pointer to the discriminant information for the discriminant list and sets \*dsc\_array\_length\_out to the count of discriminant entries. The only current applicability is the block value of a DW\_AT\_discr\_list attribute.

Those values are useful for calls to dwarf\_discr\_entry\_u() or dwarf\_discr\_entry\_s() to get the actual discriminant values. See the example below. It returns DW\_DLV\_NO\_ENTRY if the block is empty. It returns DW\_DLV\_ERROR if an error occurred.

When the call was successful and the Dwarf\_Dsc\_Head is no longer needed, call dwarf\_dealloc() to free all the space related to this.

```
void example_discr_list(Dwarf_Debug dbg,
    Dwarf_Die die,
    Dwarf_Attribute attr,
    Dwarf_Half attrnum,
    Dwarf_Bool isunsigned,
    Dwarf_Half theform,
    Dwarf_Error *err)
{
    /* The example here assumes that
       attribute attr is a DW_AT_discr_list.
       isunsigned should be set from the signedness
       of the parent of 'die' per DWARF rules for
       DW_AT_discr_list. */
    enum Dwarf_Form_Class fc = DW_FORM_CLASS_UNKNOWN;
    Dwarf_Half version = 0;
    Dwarf_Half offset_size = 0;
    int wres = 0;

    wres = dwarf_get_version_of_die(die,&version,&offset_size);
    if (wres != DW_DLV_OK) {
        /* FAIL */
        return;
    }
    fc = dwarf_get_form_class(version,attrnum,offset_size,theform);
    if (fc == DW_FORM_CLASS_BLOCK) {
        int fres = 0;
        Dwarf_Block *tempb = 0;
        fres = dwarf_formblock(attr, &tempb, err);
        if (fres == DW_DLV_OK) {
            Dwarf_Dsc_Head h = 0;
            Dwarf_Unsigned u = 0;
            Dwarf_Unsigned arraycount = 0;
            int sres = 0;

            sres = dwarf_discr_list(dbg,
                (Dwarf_Small *)tempb->bl_data,
                tempb->bl_len,
                &h,&arraycount,err);
            if (sres == DW_DLV_NO_ENTRY) {
                /* Nothing here. */
                dwarf_dealloc(dbg, tempb, DW_DLA_BLOCK);
                return;
            }
            if (sres == DW_DLV_ERROR) {
                /* FAIL . */
                dwarf_dealloc(dbg, tempb, DW_DLA_BLOCK);
            }
        }
    }
}
```

```
    return;
}
for(u = 0; u < arraycount; u++) {
    int u2res = 0;
    Dwarf_Half dtype = 0;
    Dwarf_Signed dlow = 0;
    Dwarf_Signed dhigh = 0;
    Dwarf_Unsigned ulow = 0;
    Dwarf_Unsigned uhigh = 0;

    if (isunsigned) {
        u2res = dwarf_discr_entry_u(h,u,
            &dtype,&ulow,&uhigh,err);
    } else {
        u2res = dwarf_discr_entry_s(h,u,
            &dtype,&dlow,&dhigh,err);
    }
    if( u2res == DW_DLV_ERROR) {
        /* Something wrong */
        dwarf_dealloc(dbg,h,DW_DLA_DSC_HEAD);
        dwarf_dealloc(dbg, tempb, DW_DLA_BLOCK);
        return;
    }
    if( u2res == DW_DLV_NO_ENTRY) {
        /* Impossible. u < arraycount. */
        dwarf_dealloc(dbg,h,DW_DLA_DSC_HEAD);
        dwarf_dealloc(dbg, tempb, DW_DLA_BLOCK);
        return;
    }
    /* Do something with dtype, and whichever
       of ulow, uhigh,dlow,dhigh got set.
       Probably save the values somewhere.
       Simple casting of dlow to ulow (or vice versa)
       will not get the right value due to the nature
       of LEB values. Similarly for uhigh, dhigh.
       One must use the right call.

       */
}
dwarf_dealloc(dbg,h,DW_DLA_DSC_HEAD);
dwarf_dealloc(dbg, tempb, DW_DLA_BLOCK);
}
}
```



### 6.8.19 dwarf\_discr\_entry\_u()

```
int dwarf_discr_entry_u(  
    Dwarf_Dsc_Head dsc_head,  
    Dwarf_Unsigned dsc_array_index,  
    Dwarf_Half *dsc_type,  
    Dwarf_Unsigned *dsc_low,  
    Dwarf_Unsigned *dsc_high,  
    Dwarf_Error *error)
```

When it succeeds, `dwarf_discr_entry_u()` returns `DW_DLV_OK` and sets `*dsc_type`, `*dsc_low`, and `*dsc_high` to the discriminant values for that index. Valid `dsc_array_index` values are zero to `(dsc_array_length_out - 1)` from a `dwarf_discr_list()` call.

If `*dsc_type` is `DW_DSC_label` `*dsc_low` is set to the discriminant value and `*dsc_high` is set to zero.

If `*dsc_type` is `DW_DSC_range` `*dsc_low` is set to the low end of the discriminant range and `*dsc_high` is set to the high end of the discriminant range.

Due to the nature of the LEB numbers in the discriminant representation in DWARF one must call the correct one of `dwarf_discr_entry_u()` or `dwarf_discr_entry_s()` based on whether the discriminant is signed or unsigned. Casting an unsigned to signed is not always going to get the right value.

If `dsc_array_index` is outside the range of valid indexes the function returns `DW_DLV_NO_ENTRY`. On error it returns `DW_DLV_ERROR` and sets `*error` to an error pointer.

### 6.8.20 dwarf\_discr\_entry\_s()

```
int dwarf_discr_entry_s(  
    Dwarf_Dsc_Head dsc_head,  
    Dwarf_Unsigned dsc_array_index,  
    Dwarf_Half *dsc_type,  
    Dwarf_Signed *dsc_low,  
    Dwarf_Signed *dsc_high,  
    Dwarf_Error *error)
```

This is identical to `dwarf_discr_entry_u()` except that the discriminant values are signed values in this interface. Callers must check the discriminant type and call the correct function.

## 6.9 Location List Operations, Raw `.debug_loclists`

This set of interfaces is to read the (entire) `.debug_loclists` section without reference to any DIE. As such these can only present the raw data from the file. There is no way in these interfaces to get actual addresses. These might be of interest if you want to know exactly what the compiler output in the `.debug_loclists` section. "dwarfdump ----print-raw-loclists" (try adding -v or -vvv) makes these calls.

Here is an example using all the following calls.

**Figure 17.** Example Raw Loclist

```
int example_raw_loclist(Dwarf_Debug dbg,Dwarf_Error *error)
{
    Dwarf_Unsigned count = 0;
    int res = 0;
    Dwarf_Unsigned i = 0;

    res = dwarf_load_loclists(dbg,&count,error);
    if (res != DW_DLV_OK) {
        return res;
    }
    for(i=0 ; i < count ; ++i) {
        Dwarf_Unsigned header_offset = 0;
        Dwarf_Small    offset_size = 0;
        Dwarf_Small    extension_size = 0;
        unsigned       version = 0; /* 5 */
        Dwarf_Small    address_size = 0;
        Dwarf_Small    segment_selector_size = 0;
        Dwarf_Unsigned offset_entry_count = 0;
        Dwarf_Unsigned offset_of_offset_array = 0;
        Dwarf_Unsigned offset_of_first_locentry = 0;
        Dwarf_Unsigned offset_past_last_locentry = 0;

        res = dwarf_get_loclist_context_basics(dbg,i,
            &header_offset,&offset_size,&extension_size,
            &version,&address_size,&segment_selector_size,
            &offset_entry_count,&offset_of_offset_array,
            &offset_of_first_locentry,
            &offset_past_last_locentry,error);
        if (res != DW_DLV_OK) {
            return res;
        }
        {
            Dwarf_Unsigned e = 0;
            unsigned colmax = 4;
            unsigned col = 0;
            Dwarf_Unsigned global_offset_of_value = 0;

            for ( ; e < offset_entry_count; ++e) {
                Dwarf_Unsigned value = 0;
                int resc = 0;

                resc = dwarf_get_loclist_offset_index_value(dbg,
                    i,e,&value,
                    &global_offset_of_value,error);
                if (resc != DW_DLV_OK) {
                    return resc;
                }
            }
        }
    }
}
```

```
    }
    /* Do something */
    col++;
    if (col == colmax) {
        col = 0;
    }
}

}
{
    Dwarf_Unsigned curoffset = offset_of_first_loceentry;
    Dwarf_Unsigned endoffset = offset_past_last_loceentry;
    int rese = 0;
    Dwarf_Unsigned ct = 0;

    for ( ; curoffset < endoffset; ++ct ) {
        unsigned entrylen = 0;
        unsigned code = 0;
        Dwarf_Unsigned v1 = 0;
        Dwarf_Unsigned v2 = 0;
        rese = dwarf_get_loclist_lle(dbg,i,
            curoffset,endoffset,
            &entrylen,
            &code,&v1,&v2,error);
        if (rese != DW_DLV_OK) {
            return rese;
        }
        curoffset += entrylen;
        if (curoffset > endoffset) {
            return DW_DLV_ERROR;
        }
    }
}
}
return DW_DLV_OK;
}
```

### 6.9.1 dwarf\_load\_loclists()

```
int dwarf_load_loclists(  
    Dwarf_Debug dbg,  
    Dwarf_Unsigned *loclists_count,  
    Dwarf_Error *error)
```

On a successful call to `dwarf_load_loclists()` the function returns `DW_DLV_OK`, sets `*loclists_count` (if and only if `loclists_count` is non-null) to the number of distinct section contents that exist. A small amount of data for each Location List Table (DWARF5 section 7.29) is recorded in `dbg` as a side effect. Normally `libdwarf` will have already called this, but if an application never requests any `.debug_info` data the section might not be loaded. If the section is loaded this returns very quickly and will set `*loclists_count` just as described in this paragraph.

If there is no `.debug_loclists` section in the object file this function returns `DW_DLV_NO_ENTRY`.

If something is malformed it returns `DW_DLV_ERROR` and sets `*error` to the applicable error pointer describing the problem.

There is no dealloc call. Calling `dwarf_finish()` releases the modest amount of memory recorded for this section as a side effect.

### 6.9.2 dwarf\_get\_loclist\_context\_basics()

```
int dwarf_get_loclist_context_basics(Dwarf_Debug dbg,  
    Dwarf_Unsigned context_index,  
    Dwarf_Unsigned * header_offset,  
    Dwarf_Small * offset_size,  
    Dwarf_Small * extension_size,  
    unsigned * version, /* 5 */  
    Dwarf_Small * address_size,  
    Dwarf_Small * segment_selector_size,  
    Dwarf_Unsigned * offset_entry_count,  
    Dwarf_Unsigned * offset_of_offset_array,  
    Dwarf_Unsigned * offset_of_first_locentry,  
    Dwarf_Unsigned * offset_past_last_locentry,  
    Dwarf_Error * /*err*/);
```

On success this returns `DW_DLV_OK` and returns values through the pointer arguments (other than `dbg` or `error`)

A call to `dwarf_load_loclists()` that succeeds gets you the count of contexts and `dwarf_get_loclist_context_basics()` for any "`i`  $\geq 0$  and `i`  $<$  count" gets you the context values relevant to `.debug_loclists`.

Any of the pointer-arguments for returning context values can be passed in as 0 (in which case they will be skipped).

You will want `*offset_entry_count` so you can call `dwarf_get_loclist_offset_index_value()` usefully.

If the `context_index` passed in is out of range the function returns `DW_DLV_NO_ENTRY`

At the present time `DW_DLV_ERROR` is never returned.

### 6.9.3 dwarf\_get\_loclist\_offset\_index\_value()

```
int dwarf_get_loclist_offset_index_value(Dwarf_Debug dbg,
    Dwarf_Unsigned context_index,
    Dwarf_Unsigned offsetentry_index,
    Dwarf_Unsigned * offset_value_out,
    Dwarf_Unsigned * global_offset_value_out,
    Dwarf_Error *error)
```

On success `dwarf_get_loclist_offset_index_value()` returns `DW_DLV_OK`, sets `* offset_value_out` to the value in the Range List Table offset array, and sets `* global_offset_value_out` to the section offset (in `.debug_addr`) of the offset value.

Pass in `context_index` exactly as the same field passed to `dwarf_get_loclist_context_basics()`.

Pass in `offset_entry_index` based on the return field `offset_entry_count` from `dwarf_get_loclist_context_basics()`, meaning for that `context_index` an `offset_entry_index >=0` and `< offset_entry_count`.

Pass in `offset_entry_count` exactly as the same field passed to `dwarf_get_loclist_context_basics()`.

If one of the indexes passed in is out of range `DW_DLV_NO_ENTRY` will be returned and no return arguments touched.

If there is some corruption of DWARF5 data then `DW_DLV_ERROR` might be returned and `*error` set to the error details.

### 6.9.4 dwarf\_get\_loclist\_lle()

```
int dwarf_get_loclist_lle(  
    Dwarf_Debug dbg,  
    Dwarf_Unsigned contextnumber,  
    Dwarf_Unsigned entry_offset,  
    Dwarf_Unsigned endoffset,  
    unsigned *entrylen,  
    unsigned *entry_kind,  
    Dwarf_Unsigned *entry_operand1,  
    Dwarf_Unsigned *entry_operand2,  
    Dwarf_Unsigned *expr_ops_blocksize,  
    Dwarf_Unsigned *expr_ops_offset,  
    Dwarf_Small    **expr_opsdata,  
    Dwarf_Error *error)
```

On success it returns a single DW\_RLE\* record (see dwarf.h) fields.

contextnumber is the number of the current loclist context.

entry\_offset is the section offset (section-global offset) of the next record.

endoffset is one past the last entry in this rle context.

\*entrylen returns the length in the .debug\_loclists section of the particular record returned. It's used to increment to the next record within this loclist context.

\*entrykind returns is the DW\_RLE\* number.

Some record kinds have 1 or 0 operands, most have two operands (the records describing ranges).

\*expr\_ops\_blocksize returns the size, in bytes, of the Dwarf Expression (some operations have no Dwarf Expression and those that do can have a zero length blocksize).

\*expr\_ops\_offset returns the offset (in the .debug\_loclists section) of the first byte of the Dwarf Expression.

\*expr\_opsdata returns a pointer to the bytes of the Dwarf Expression.

If the contextnumber is out of range it will return DW\_DLV\_NO\_ENTRY.

If the .debug\_loclists section is malformed or the entry\_offset is incorrect it may return DW\_DLV\_ERROR.

## 6.10 Location List operations .debug\_loc & .debug\_loclists

These operations apply to the .debug\_loc section in DWARF2, DWARF3, DWARF4, and DWARF5 object files. Earlier versions still work as well as ever, but they only deal with, at most, DWARF2, DWARF3, and DWARF4.

### 6.10.1 dwarf\_get\_loclist\_c()

```
int dwarf_get_loclist_c (Dwarf_Attribute attr,  
    Dwarf_Loc_Head_c * loclist_head,  
    Dwarf_Unsigned    * locCount,  
    Dwarf_Error       * error);
```

This function returns a pointer that is, in turn, used to make possible calls to return the details of the location list.

The incoming argument `attr` should have one of the FORMs of a location expression or location list.

On success this returns `DW_DLV_OK` and sets `*loclist_head` to a pointer used in further calls (see the example and descriptions that follow it). `locCount` is set to the number of entries in the location list (or if the FORM is of a location expression the `locCount` will be set to one). At this point one cannot yet tell if it was a location list or a location expression (see `. dwarf_get_locdesc_entry_c{ }`).

In case of error `DW_DLV_ERROR` is returned and `*error` is set to an error designation.

A return of `DW_DLV_NO_ENTRY` may be possible but is a bit odd.



```
void example_loclistcv5(Dwarf_Debug dbg,Dwarf_Attribute someattr)
{
    Dwarf_Unsigned lcount = 0;
    Dwarf_Loc_Head_c loclist_head = 0;
    Dwarf_Error error = 0;
    int lres = 0;

    lres = dwarf_get_loclist_c(someattr,&loclist_head,&lcount,&error);
    if (lres == DW_DLV_OK) {
        Dwarf_Unsigned i = 0;

        /* Before any return remember to call
           dwarf_loc_head_c_dealloc(loclist_head); */
        for (i = 0; i < lcount; ++i) {
            Dwarf_Small loclist_lkind = 0;
            Dwarf_Small lle_value = 0;
            Dwarf_Unsigned rawval1 = 0;
            Dwarf_Unsigned rawval2 = 0;
            Dwarf_Bool debug_addr_unavailable = FALSE;
            Dwarf_Addr lopc = 0;
            Dwarf_Addr hipc = 0;
            Dwarf_Unsigned loclist_expr_op_count = 0;
            Dwarf_Locdesc_c locdesc_entry = 0;
            Dwarf_Unsigned expression_offset = 0;
            Dwarf_Unsigned locdesc_offset = 0;

            lres = dwarf_get_locdesc_entry_d(loclist_head,
                i,
                &lle_value,
                &rawval1,&rawval2,
                &debug_addr_unavailable,
                &lopc,&hipc,
                &loclist_expr_op_count,
                &locdesc_entry,
                &loclist_lkind,
                &expression_offset,
                &locdesc_offset,
                &error);
            if (lres == DW_DLV_OK) {
                Dwarf_Unsigned j = 0;
                int opres = 0;
                Dwarf_Small op = 0;

                for (j = 0; j < loclist_expr_op_count; ++j) {
                    Dwarf_Unsigned raw1 = 0;
                    Dwarf_Unsigned raw2 = 0;
```

```
Dwarf_Unsigned raw3 = 0;
Dwarf_Unsigned opd1 = 0;
Dwarf_Unsigned opd2 = 0;
Dwarf_Unsigned opd3 = 0;
Dwarf_Unsigned offsetforbranch = 0;

opres = dwarf_get_location_op_value_d(
    locdesc_entry,
    j,&op,
    &raw1,&raw2,&raw3,
    &opd1, &opd2,&opd3,&offsetforbranch,
    &error);
if (opres == DW_DLV_OK) {
    /* Do something with the operators.
       Usually you want to use opd1,2,3
       as appropriate. Calculations
       involving base addresses etc
       have already been incorporated
       in opd1,2,3. */
} else {
    dwarf_dealloc_error(dbg,error);
    dwarf_loc_head_c_dealloc(loclist_head);
    /*Something is wrong. */
    return;
}
}
} else {
    /* Something is wrong. Do something. */
    dwarf_loc_head_c_dealloc(loclist_head);
    dwarf_dealloc_error(dbg,error);
    return;
}
}
}
/* Always call dwarf_loc_head_c_dealloc()
   to free all the memory associated with loclist_head. */
if (error) {
    dwarf_dealloc_error(dbg,error);
}
dwarf_loc_head_c_dealloc(loclist_head);
loclist_head = 0;
return;
}
```

### 6.10.2 dwarf\_get\_locdesc\_entry\_d()

Earlier versions of this work with earlier versions of DWARF. This works with all DWARF from DWARF2 on.

```
int dwarf_get_locdesc_entry_d(Dwarf_Loc_Head_c /*loclist_head*/,
    Dwarf_Unsigned    index,
    Dwarf_Small        *lle_value_out,
    Dwarf_Addr         *rawval1_out,
    Dwarf_Addr         *rawval2_out,
    Dwarf_Bool         *debug_addr_unavailable,
    Dwarf_Addr         *lopc_out,
    Dwarf_Addr         *hipc_out,
    Dwarf_Unsigned     *loc_expr_op_count_out,
    Dwarf_Locdesc_c    *locentry_out,
    Dwarf_Small        *loclist_kind,
    Dwarf_Unsigned     *expression_offset_out,
    Dwarf_Unsigned     *locdesc_offset_out,
    Dwarf_Error         *error);
```

This function returns overall information about a location list or location description. Details about location operators are retrieved by a call to `dwarf_get_location_op_value_d()` (described below). In case of success DW\_DLV\_OK is returned and arguments are set through the pointers to return values to the caller. Now we describe each argument.

`*loclist_kind` returns DW\_LKIND\_expression, DW\_LKIND\_loclist, DW\_LKIND\_GNU\_exp\_list, or DW\_LKIND\_loclists.

DW\_LKIND\_expression means the 'list' is really just a location expression. The only entry is with index zero.

DW\_LKIND\_loclist, means the list is from DWARF2, DWARF3, or DWARF4. The `*lle_value_out` value has been synthesized as if it were a DWARF5 expression.

DW\_LKIND\_GNU\_exp\_list, means the list is from a DWARF4 .debug\_loc.dwo object section. It is an experimental version from before DWARF5 was published. The `*lle_value_out` is DW\_LLEX\_start\_end\_entry (or one of the other DW\_LLEX values).

DW\_LKIND\_loclists means this is a DWARF5 loclist, so DW\_LLE\_start\_end is an example of one possible `*lle_value_out` values. In addition, if `*debug_addr_unavailable` is set it means the `*lopc_out` and `*hipc_out` could not be correctly set (so are meaningless) because the .debug\_addr section is missing. Very likely the .debug\_addr section is in the executable and that file needs to be opened and attached to the current Dwarf\_Debug with `dwarf_set_tied_dbg()`.

`*rawval1_out` returns the value of the first operand in the location list entry. Uninterpreted. Useful for reporting or for those wishing to do their own calculation of `lopc`.

\*rawval2\_out returns the value of the second operand in the location list entry. Uninterpreted. Useful for reporting or for those wishing to do their own calculation of hpc.

The argument loc\_expr\_op\_count\_out returns the number of operators in the location expression involved (which may be zero).

The argument locentry\_out returns an identifier used in calls to dwarf\_get\_location\_op\_value\_d().

The argument expression\_offset\_out returns the offset (in the .debug\_loc(.dso) or .debug\_info(.dwo) of the location expression itself (possibly useful for debugging).

The argument locdesc\_offset\_out returns the offset (in the section involved (see loclist\_kind) of the location list entry itself (possibly useful for debugging).

In case of error DW\_DLV\_ERROR is returned and \*error is set to an error designation.

A return of DW\_DLV\_NO\_ENTRY may be possible but is a bit odd.

### 6.10.3 dwarf\_get\_loclist\_head\_kind()

```
int dwarf_get_loclist_head_kind(  
    Dwarf_Loclists_Head head,  
    unsigned int * kind,  
    Dwarf_Error *error)
```

Though one should test the return code, at present this always returns DW\_DLV\_OK, and sets \*kind to the DW\_LKIND\* value for this head.

At the present time neither DW\_DLV\_ERROR nor DW\_DLV\_NO\_ENTRY is returned.

### 6.10.4 dwarf\_get\_location\_op\_value\_d()

```
int dwarf_get_location_op_value_d(Dwarf_Locdesc_c locdesc,  
    Dwarf_Unsigned    index,  
    Dwarf_Small      * atom_out,  
    Dwarf_Unsigned * operand1,  
    Dwarf_Unsigned * operand2,  
    Dwarf_Unsigned * operand3,  
    Dwarf_Unsigned * rawop1,  
    Dwarf_Unsigned * rawop2,  
    Dwarf_Unsigned * rawop3,  
    Dwarf_Unsigned * offset_for_branch,  
    Dwarf_Error*    error);
```

On success The function `dwarf_get_location_op_value_d()` returns the information for the single operator number `index` from the location expression `locdesc`. It sets the following values.

`atom_out` is set to the applicable operator code, for example `DW_OP_reg5`.

`operand1`, `operand2`, and `operand3` are set to the operator operands as applicable (see DWARF documents on the operands for each operator). All additions of base fields, if any, have been done already. `operand3` is new as of DWARF5.

In some cases `operand3` is actually a pointer into section data in memory and `operand2` has the length of the data at `operand3`. Callers must extract the bytes and deal with endianness issues of the extracted value.

`rawop1`, `rawop2`, and `rawop3` are set to the operator operands as applicable (see DWARF documents on the operands for each operator) before any base values were added in.. As for the previous, sometimes dealing with `rawop3` means interpreting it as a pointer and doing a dereference.

More on the pointer values in `Dwarf_Unsigned`: When a DWARF operand is not of a size fixed by dwarf or whose type is unknown, or is possibly too large for a dwarf stack entry, `libdwarf` will insert a pointer (to memory in the dwarf data somewhere) as the operand value. `DW_OP_implicit_value` operand 2, `DW_OP_[GNU_]entry_value` operand 2, and `DW_OP_[GNU_]const_type` operand 3 are instances of this. The problem with the values is that `libdwarf` is unclear what the type of the value is so we pass the problem to you, the callers!

`offset_for_branch` is set to the offset (in bytes) in this expression of this operator. The value makes it possible for callers to implement the operator branch operators.

In case of an error, the function returns `DW_DLV_ERROR` and sets `*error` to an error value.

`DW_DLV_NO_ENTRY` is probably not a possible return value, but please test for it anyway.

### **6.10.5 dwarf\_loclist\_from\_expr\_c()**

This is now obsolete, though it works as well as ever, so if it works for your object codes you may continue to use it.

```
int dwarf_loclist_from_expr_c(Dwarf_Debug dbg,
    Dwarf_Ptr      expression_in,
    Dwarf_Unsigned  expression_length,
    Dwarf_Half      address_size,
    Dwarf_Half      offset_size,
    Dwarf_Small      dwarf_version,
    Dwarf_Loc_Head_c* loc_head,
    Dwarf_Unsigned * listlen,
    Dwarf_Error * error);
```

This interface is not sufficient to work properly as it fails to pass in data from the Compilation Unit. The earlier versions `dwarf_loclist_from_expr()`, `dwarf_loclist_from_expr_a()`, and `dwarf_loclist_from_expr_b` are all similarly deficient. These suffice for early DWARF locations but cannot work for every kind of DWARF5 location list or location.

Frame operators such as `DW_CFA_def_cfa_expression` have a location expression and the `location_expression` is accessed with this function.

On success it returns `DW_DLV_OK` and sets the two return arguments (explained a few lines later here).

The `expression_in` argument must contain a valid pointer to location expression bytes. The `expression_length` argument must contain the length of that location expression in bytes.

The `address_size` argument must contain the size of an address on the target machine for this expression (normally 4 or 8). The `offset_size` argument must contain the size of an offset in the expression (normally 4, sometimes 8). The `version` argument must contain the `dwarf_version` of the expression (2,3,4, or 5).

The returned value `*loc_head` is used to actually access the location expression details (see the example following).

The returned value `*listlen` is the number of location expressions (ie 1) in the location list (for uniformity of access we make it look like a single-entry location list).

On error the function returns `DW_DLV_ERROR` and sets `*error` to reflect the error.

A return of `DW_DLV_NO_ENTRY` is probably impossible, but callers should assume it is possible. No return arguments are set in this case.

```
void
example_locexpr(Dwarf_Debug dbg,Dwarf_Ptr expr_bytes,
    Dwarf_Unsigned expr_len,
    Dwarf_Half addr_size,
    Dwarf_Half offset_size,
    Dwarf_Half version)
{
    Dwarf_Loc_Head_c head = 0;
    Dwarf_Locdesc_c locentry = 0;
    int res2 = 0;
    Dwarf_Unsigned lopc = 0;
    Dwarf_Unsigned hipc = 0;
    Dwarf_Unsigned ulistlen = 0;
    Dwarf_Unsigned ulocentry_count = 0;
    Dwarf_Unsigned section_offset = 0;
    Dwarf_Unsigned locdesc_offset = 0;
    Dwarf_Small lle_value = 0;
    Dwarf_Small loclist_source = 0;
    Dwarf_Unsigned i = 0;
    Dwarf_Error error = 0;

    res2 = dwarf_loclist_from_expr_c(dbg,
        expr_bytes,expr_len,
        addr_size,
        offset_size,
        version,
        &head,
        &ulistlen,
        &error);
    if(res2 == DW_DLV_NO_ENTRY) {
        return;
    }
    if(res2 == DW_DLV_ERROR) {
        return;
    }
    /* These are a location expression, not loclist.
       So we just need the 0th entry. */
    res2 = dwarf_get_locdesc_entry_c(head,
        0, /* Data from 0th LocDesc */
        &lle_value,
        &lopc, &hipc,
        &ulocentry_count,
        &locentry,
        &loclist_source,
        &section_offset,
        &locdesc_offset,
```

```
    &error);
if (res2 == DW_DLV_ERROR) {
    dwarf_loc_head_c_dealloc(head);
    return;
} else if (res2 == DW_DLV_NO_ENTRY) {
    dwarf_loc_head_c_dealloc(head);
    return;
}
/* ASSERT: ulistlen == 1 */
for (i = 0; i < ulocentry_count;++i) {
    Dwarf_Small op = 0;
    Dwarf_Unsigned opd1 = 0;
    Dwarf_Unsigned opd2 = 0;
    Dwarf_Unsigned opd3 = 0;
    Dwarf_Unsigned offsetforbranch = 0;

    res2 = dwarf_get_location_op_value_c(locentry,
        i, &op,&opd1,&opd2,&opd3,&offsetforbranch,
        &error);
    /* Do something with the expression operator and operands */
    if (res2 != DW_DLV_OK) {
        dwarf_loc_head_c_dealloc(head);
        return;
    }
}
dwarf_loc_head_c_dealloc(head);
}
```

#### **6.10.6 dwarf\_loc\_head\_c\_dealloc()**

```
void dwarf_loc_head_c_dealloc(Dwarf_Loc_Head_c loclist_head);
```

This function takes care of all the details so one does not have to `_dwarf_dealloc()` the pieces individually, though code that continues to do the pieces individually still works.

This function frees all the memory associated with the `loclist_head`. There is no return value. It's good practice to set `loclist_head` to zero immediately after the call, as the pointer is stale at that point.

#### **6.10.7 dwarf\_loclist\_n()**



```
int dwarf_loclist_n(  
    Dwarf_Attribute attr,  
    Dwarf_Locdesc ***llbuf,  
    Dwarf_Signed *listlen,  
    Dwarf_Error *error)
```

This interface cannot handle DWARF5 or Split Dwarf. Use `dwarf_get_loclist_c()` and related functions instead (as of November 2015). The function `dwarf_loclist_n()` sets `*llbuf` to point to an array of `Dwarf_Locdesc` pointers corresponding to each of the location expressions in a location list, and sets `*listlen` to the number of elements in the array and returns `DW_DLV_OK` if the attribute is appropriate.

This is the preferred function for `Dwarf_Locdesc` as it is the interface allowing access to an entire loclist. (use of `dwarf_loclist_n()` is suggested as the better interface, though `dwarf_loclist()` is still supported.)

If the attribute is a reference to a location list (`DW_FORM_data4` or `DW_FORM_data8`) the location list entries are used to fill in all the fields of the `Dwarf_Locdesc(s)` returned.

If the attribute is a location description (`DW_FORM_block2` or `DW_FORM_block4`) then some of the `Dwarf_Locdesc` values of the single `Dwarf_Locdesc` record are set to 'sensible' but arbitrary values. Specifically, `ld_lopc` is set to 0 and `ld_hipc` is set to all-bits-on. And `*listlen` is set to 1.

If the attribute is a reference to a location expression (`DW_FORM_locexpr`) then some of the `Dwarf_Locdesc` values of the single `Dwarf_Locdesc` record are set to 'sensible' but arbitrary values. Specifically, `ld_lopc` is set to 0 and `ld_hipc` is set to all-bits-on. And `*listlen` is set to 1.

It returns `DW_DLV_ERROR` on error.

`dwarf_loclist_n()` works on `DW_AT_location`, `DW_AT_data_member_location`, `DW_AT_vtable_elem_location`, `DW_AT_string_length`, `DW_AT_use_location`, and `DW_AT_return_addr` attributes.

If the attribute is `DW_AT_data_member_location` the value may be of class `CONSTANT`. `dwarf_loclist_n()` is unable to read class `CONSTANT`, so you need to first determine the class using `dwarf_get_form_class()` and if it is class `CONSTANT` call `dwarf_formsdata()` or `dwarf_formudata()` to get the constant value (you may need to call both as DWARF4 does not define the signedness of the constant value).

Storage allocated by a successful call of `dwarf_loclist_n()` should be deallocated when no longer of interest (see `dwarf_dealloc()`). The block of `Dwarf_Loc` structs pointed to by the `ld_s` field of each `Dwarf_Locdesc` structure should be deallocated with the allocation type `DW_DLA_LOC_BLOCK`. and the `llbuf[]` space pointed to should be deallocated with allocation type `DW_DLA_LOCDDESC`. This should be followed by deallocation of the `llbuf` using the allocation type `DW_DLA_LIST`.

```
void example9(Dwarf_Debug dbg,Dwarf_Attribute someattr)
{
    Dwarf_Signed lcount = 0;
    Dwarf_Locdesc **llbuf = 0;
    Dwarf_Error error = 0;
    int lres = 0;

    lres = dwarf_loclist_n(someattr, &llbuf,&lcount,&error);
    if (lres == DW_DLV_OK) {
        Dwarf_Signed i = 0;
        for (i = 0; i < lcount; ++i) {
            /* Use llbuf[i]. Both Dwarf_Locdesc and the
               array of Dwarf_Loc it points to are
               defined in libdwarf.h: they are
               not opaque structs. */
            dwarf_dealloc(dbg, llbuf[i]->ld_s, DW_DLA_LOC_BLOCK);
            dwarf_dealloc(dbg,llbuf[i], DW_DLA_LOCDDESC);
        }
        dwarf_dealloc(dbg, llbuf, DW_DLA_LIST);
    }
}
```

### 6.10.8 dwarf\_loclist()

```
int dwarf_loclist(
    Dwarf_Attribute attr,
    Dwarf_Locdesc **llbuf,
    Dwarf_Signed *listlen,
    Dwarf_Error *error)
```

Use `dwarf_get_loclist_c()` and related functions instead (as of November 2015). The function `dwarf_loclist()` sets `*llbuf` to point to a `Dwarf_Locdesc` pointer for the single location expression it can return. It sets `*listlen` to 1. and returns `DW_DLV_OK` if the attribute is appropriate.

It is less flexible than `dwarf_loclist_n()` in that `dwarf_loclist()` can handle a maximum of one location expression, not a full location list. If a location-list is present it returns only the first location-list entry location description. Use `dwarf_loclist_n()` instead.

It returns `DW_DLV_ERROR` on error. `dwarf_loclist()` works on `DW_AT_location`, `DW_AT_data_member_location`, `DW_AT_vtable_elem_location`, `DW_AT_string_length`, `DW_AT_use_location`, and `DW_AT_return_addr` attributes.

Storage allocated by a successful call of `dwarf_loclist()` should be deallocated

when no longer of interest (see `dwarf_dealloc()`). The block of `Dwarf_Loc` structs pointed to by the `ld_s` field of each `Dwarf_Locdesc` structure should be deallocated with the allocation type `DW_DLA_LOC_BLOCK`. This should be followed by deallocation of the `llbuf` using the allocation type `DW_DLA_LOCDISC`.

**Figure 18.** Example `dwarf_loclist()`

```
void examplea(Dwarf_Debug dbg,Dwarf_Attribute someattr)
{
    Dwarf_Signed lcount = 0;
    Dwarf_Locdesc *llbuf = 0;
    Dwarf_Error error = 0;
    int lres = 0;

    lres = dwarf_loclist(someattr, &llbuf,&lcount,&error);
    if (lres == DW_DLV_OK) {
        /* lcount is always 1, (and has always been 1) */
        /* Use llbuf here. */

        dwarf_dealloc(dbg, llbuf->ld_s, DW_DLA_LOC_BLOCK);
        dwarf_dealloc(dbg, llbuf, DW_DLA_LOCDISC);
    }
}
```

### 6.10.9 `dwarf_loclist_from_expr()`

```
int dwarf_loclist_from_expr(
    Dwarf_Debug dbg,
    Dwarf_Ptr bytes_in,
    Dwarf_Unsigned bytes_len,
    Dwarf_Locdesc **llbuf,
    Dwarf_Signed *listlen,
    Dwarf_Error *error)
```

Use `dwarf_loclist_from_expr_b()` instead. This function is obsolete.

The function `dwarf_loclist_from_expr()` sets `*llbuf` to point to a `Dwarf_Locdesc` pointer for the single location expression which is pointed to by `*bytes_in` (whose length is `*bytes_len`). It sets `*listlen` to 1. and returns `DW_DLV_OK` if decoding is successful. Some sources of bytes of expressions are dwarf expressions in frame operations like `DW_CFA_def_cfa_expression`, `DW_CFA_expression`, and `DW_CFA_val_expression`.

Any `address_size` data in the location expression is assumed to be the same size as the default `address_size` for the object being read (normally 4 or 8).

It returns `DW_DLV_ERROR` on error.

Storage allocated by a successful call of `dwarf_loclist_from_expr()` should be deallocated when no longer of interest (see `dwarf_dealloc()`). The block of `Dwarf_Loc` structs pointed to by the `ld_s` field of each `Dwarf_Locdesc` structure should be deallocated with the allocation type `DW_DLA_LOC_BLOCK`. This should be followed by deallocation of the `llbuf` using the allocation type `DW_DLA_LOCDESC`.

**Figure 19.** Exampleb `dwarf_loclist_from_expr()`

```
void exampleb(Dwarf_Debug dbg, Dwarf_Ptr data, Dwarf_Unsigned len)
{
    Dwarf_Signed lcount = 0;
    Dwarf_Locdesc *llbuf = 0;
    Dwarf_Error error = 0;
    int lres = 0;

    lres = dwarf_loclist_from_expr(dbg, data, len, &llbuf, &lcount,
                                   &error);
    if (lres == DW_DLV_OK) {
        /* lcount is always 1 */
        /* Use llbuf here.*/

        dwarf_dealloc(dbg, llbuf->ld_s, DW_DLA_LOC_BLOCK);
        dwarf_dealloc(dbg, llbuf, DW_DLA_LOCDESC);
    }
}
```

#### 6.10.10 `dwarf_loclist_from_expr_b()`

```
int dwarf_loclist_from_expr_a(
    Dwarf_Ptr bytes_in,
    Dwarf_Unsigned bytes_len,
    Dwarf_Half addr_size,
    Dwarf_Half offset_size,
    Dwarf_Half version_stamp,
    Dwarf_Locdesc **llbuf,
    Dwarf_Signed *listlen,
    Dwarf_Error *error)
```

This function is obsolete. The function `dwarf_loclist_from_expr_b()` is identical to `dwarf_loclist_from_expr_a()` in every way except that the caller passes an additional argument `version_stamp` containing the version stamp (2 for DWARF2, etc) of the CU using this location expression and an additional argument of the offset size of the CU using this location expression. The `DW_OP_GNU_implicit_pointer` operation requires this version and offset information to be correctly processed.

The `addr_size` argument (from 27April2009) is needed to correctly interpret frame information as different compilation units can have different address sizes. DWARF4 adds `address_size` to the CIE header.

### 6.10.11 dwarf\_loclist\_from\_expr\_a()

```
int dwarf_loclist_from_expr_a(  
    Dwarf_Ptr bytes_in,  
    Dwarf_Unsigned bytes_len,  
    Dwarf_Half addr_size,  
    Dwarf_Locdesc **llbuf,  
    Dwarf_Signed *listlen,  
    Dwarf_Error *error)
```

This function is obsolete. Use `dwarf_loclist_from_expr_b()` instead. This function is obsolete.

The function `dwarf_loclist_from_expr_a()` is identical to `dwarf_loclist_from_expr()` in every way except that the caller passes the additional argument `addr_size` containing the address size (normally 4 or 8) applying this location expression.

The `addr_size` argument (added 27April2009) is needed to correctly interpret frame information as different compilation units can have different address sizes. DWARF4 adds `address_size` to the CIE header.

## 6.11 Line Number Operations

These functions are concerned with accessing line number entries, mapping debugging information entry objects to their corresponding source lines, and providing a mechanism for obtaining information about line number entries. Although, the interface talks of "lines" what is really meant is "statements". In case there is more than one statement on the same line, there will be at least one descriptor per statement, all with the same line number. If column number is also being represented they will have the column numbers of the start of the statements also represented.

There can also be more than one `Dwarf_Line` per statement. For example, if a file is preprocessed by a language translator, this could result in translator output showing 2 or more sets of line numbers per translated line of output.

As of October 2015 there are two sets of overall access and release functions. The older set of functions is `dwarf_srclines()` with `dwarf_srclines_dealloc()`. This set does not handle line table headers with no lines.

A newer set is `dwarf_srclines_b()` with `dwarf_srclines_from_linecontext()` and `dwarf_srclines_dealloc_b()`. These functions provide for handling both DWARF2 through DWARF5 details and give access to line header information even if there are no lines in a particular compilation unit's line table.

### 6.11.1 Get A Set of Lines (including skeleton line tables)

This set of functions works on any DWARF version. DWARF2,3,4,5 and the DWARF4 based experimental two-level line tables are all supported. What was once done by `dwarf_srclines()` alone is now done with two calls as described here.

The interfaces support reading GNU two-level line tables. The format of such tables is a topic beyond the scope of this document.

### 6.11.2 `dwarf_srclines_b()`

This is the

```
int dwarf_srclines_b(  
    Dwarf_Die die,  
    Dwarf_Unsigned *version_out,  
    Dwarf_Bool      *is_single_table,  
    Dwarf_Line_Context *context_out,  
    Dwarf_Error *error)
```

`dwarf_srclines_b()` takes a single argument as input, a pointer to a compilation-unit (CU) DIE. The other arguments are used to return values to the caller. On success `DW_DLV_OK` is returned and values are returned through the pointers. If there is no line table `DW_DLV_NO_ENTRY` is returned and no values are returned through the pointers. If `DW_DLV_ERROR` is returned the involved is returned through the `error` pointer.

The values returned on success are:

`*version_out()` is set to the version number from the line table header for this CU. The experimental two-level line table value is 0xf006. Standard numbers are 2,3,4 and 5.

`*is_single_table()` is set to non-zero if the line table is an ordinary single line table. If the line table is anything else (either a line table header with no lines or an experimental two-level line table) it is set to zero.

`*context_out()` is set to an opaque pointer to a `Dwarf_Line_Context` record which in turn is used to get other data from this line table. See below.

See `*dwarf_srclines_dealloc_b()` for examples showing correct use.

### 6.11.3 `dwarf_get_line_section_name_from_die()`

```
int dwarf_get_line_section_name_from_die(  
    Dwarf_Die die,  
    const char ** sec_name,  
    Dwarf_Error *error)
```

`*dwarf_get_line_section_name_from_die()` retrieves the object file section

name of the applicable line section. This is useful for applications wanting to print the name, but of course the object section name is not really a part of the DWARF information. Most applications will probably not call this function. It can be called at any time after the Dwarf\_Debug initialization is done.

If the function succeeds, `*sec_name` is set to a pointer to a string with the object section name and the function returns `DW_DLV_OK`. Do not free the string whose pointer is returned. For non-Elf objects it is possible the string pointer returned will be NULL or will point to an empty string. It is up to the calling application to recognize this possibility and deal with it appropriately.

If the section does not exist the function returns `DW_DLV_NO_ENTRY`.

If there is an internal error detected the function returns `DW_DLV_ERROR` and sets the `*error` pointer.

#### 6.11.4 dwarf\_srclines\_from\_linecontext()

```
int dwarf_srclines_from_linecontext(  
    Dwarf_Line_Context line_context,  
    Dwarf_Line ** linebuf,  
    Dwarf_Signed *linecount,  
    Dwarf_Error *error)
```

`*dwarf_srclines_from_linecontext()` gives access to the line tables. On success it returns `DW_DLV_OK` and passes back line tables through the pointers.

Though `DW_DLV_OK` will not be returned callers should assume it is possible.

On error `DW_DLV_ERROR` is returned and the error code set through the `error` pointer.

On success:

`*linebuf` is set to an array of `Dwarf_Line` pointers.

`*linecount` is set to the number of pointers in the array.

#### 6.11.5 dwarf\_srclines\_two\_levelfrom\_linecontext()

```
int dwarf_srclines_two_levelfrom_linecontext(  
    Dwarf_Line_Context line_context,  
    Dwarf_Line ** linebuf,  
    Dwarf_Signed *linecount,  
    Dwarf_Line ** linebuf_actuais,  
    Dwarf_Signed *linecount_actuais,  
    Dwarf_Error *error)
```

`*dwarf_srclines_two_levelfrom_linecontext()` gives access to the line

tables. On success it returns DW\_DLV\_OK and passes back line tables through the pointers.

Though DW\_DLV\_OK will not be returned callers should assume it is possible.

On error DW\_DLV\_ERROR is returned and the error code set through the `error` pointer.

On success:

`*linebuf` is set to an array of Dwarf\_Line pointers.

`*linecount` is set to the number of pointers in the array.

If one is not intending that the experimental two-level line tables are of interest then pass NULL for `*linebuf_actuais` and `*linecount_actuais`. The NULL pointers notify the library that the second table is not to be passed back.

If a line table is actually a two-level tables `*linebuf` is set to point to an array of Logicals lines. `*linecount` is set to the number of Logicals. `*linebuf_actuais` is set to point to an array of Actuals lines. `*linecount_actuais` is set to the number of Actuals.

#### 6.11.6 dwarf\_srclines\_dealloc\_b()

```
void dwarf_srclines_dealloc_b(  
    Dwarf_Line_Context line_context,  
    Dwarf_Error *error)
```

This does a complete deallocation of the memory of the Dwarf\_Line\_Context and the Dwarf\_Line array (or arrays) that came from the Dwarf\_Line\_Context. On return you should set any local pointers to these buffers to NULL as a reminder that any use of the local pointers would be to stale memory.

**Figure 20.** Example of dwarf\_srclines\_b()



```
void examplec(Dwarf_Die cu_die)
{
    /* EXAMPLE: DWARF5 style access. */
    Dwarf_Line *linebuf = 0;
    Dwarf_Signed linecount = 0;
    Dwarf_Line *linebuf_actuals = 0;
    Dwarf_Signed linecount_actuals = 0;
    Dwarf_Line_Context line_context = 0;
    Dwarf_Signed linecount_total = 0;
    Dwarf_Small table_count = 0;
    Dwarf_Unsigned lineversion = 0;
    Dwarf_Error err = 0;
    int sres = 0;
    /* ... */
    /* we use 'return' here to signify we can do nothing more
       at this point in the code. */
    sres = dwarf_srclines_b(cu_die,&lineversion,
        &table_count,&line_context,&err);
    if (sres != DW_DLV_OK) {
        /* Handle the DW_DLV_NO_ENTRY or DW_DLV_ERROR
           No memory was allocated so there nothing
           to dealloc. */
        return;
    }
    if (table_count == 0) {
        /* A line table with no actual lines.
           This occurs in a DWARF5 or DWARF5
           DW_TAG_type_unit
           as such has no lines of code
           but needs data for
           DW_AT_decl_file attributes. */
        /*...do something, see dwarf_srclines_files_count()
           etc below. */

        dwarf_srclines_dealloc_b(line_context);
        /* All the memory is released, the line_context
           and linebuf zeroed now
           as a reminder they are stale. */
        linebuf = 0;
        line_context = 0;
    } else if (table_count == 1) {
        Dwarf_Signed i = 0;
        Dwarf_Signed baseindex = 0;
        Dwarf_Signed file_count = 0;
        Dwarf_Signed endindex = 0;
        /* Standard dwarf 2,3,4, or 5 line table */
    }
}
```

```
/* Do something. */

/* First let us index through all the files listed
   in the line table header. */
sres = dwarf_srclines_files_indexes(line_context,
    &baseindex,&file_count,&endindex,&err);
if (sres != DW_DLV_OK) {
    /* Something badly wrong! */
    return;
}
/* Works for DWARF2,3,4 (one-based index)
   and DWARF5 (zero-based index) */
for (i = baseindex; i < endindex; i++) {
    Dwarf_Unsigned dirindex = 0;
    Dwarf_Unsigned modtime = 0;
    Dwarf_Unsigned flength = 0;
    Dwarf_Form_Data16 *md5data = 0;
    int vres = 0;
    const char *name = 0;

    vres = dwarf_srclines_files_data_b(line_context,i,
        &name,&dirindex, &modtime,&flength,
        &md5data,&err);
    if (vres != DW_DLV_OK) {
        /* something very wrong. */
        return;
    }
    /* Do something. */
}

/* For this case where we have a line table we will likely
   wish to get the line details: */
sres = dwarf_srclines_from_linecontext(line_context,
    &linebuf,&linecount,
    &err);
if (sres != DW_DLV_OK) {
    /* Error. Clean up the context information. */
    dwarf_srclines_dealloc_b(line_context);
    return;
}
/* The lines are normal line table lines. */
for (i = 0; i < linecount; ++i) {
    /* use linebuf[i] */
}
dwarf_srclines_dealloc_b(line_context);
```

```
/* All the memory is released, the line_context
   and linebuf zeroed now as a reminder they are stale */
linebuf = 0;
line_context = 0;
linecount = 0;
} else {
    Dwarf_Signed i = 0;
    /* ASSERT: table_count == 2,
       Experimental two-level line table. Version 0xf006
       We do not define the meaning of this non-standard
       set of tables here. */

    /* For 'something C' (two-level line tables)
       one codes something like this
       Note that we do not define the meaning or use of two-level line
       tables as these are experimental, not standard DWARF. */
    sres = dwarf_srclines_two_level_from_linecontext(line_context,
        &linebuf,&linecount,
        &linebuf_actuais,&linecount_actuais,
        &err);
    if (sres == DW_DLV_OK) {
        for (i = 0; i < linecount; ++i) {
            /* use linebuf[i], these are the 'logicals' entries. */
        }
        for (i = 0; i < linecount_actuais; ++i) {
            /* use linebuf_actuais[i], these are the actuals entries
            */
        }
        dwarf_srclines_dealloc_b(line_context);
        line_context = 0;
        linebuf = 0;
        linecount = 0;
        linebuf_actuais = 0;
        linecount_actuais = 0;
    } else if (sres == DW_DLV_NO_ENTRY) {
        /* This should be impossible, but do something.    */
        /* Then Free the line_context */
        dwarf_srclines_dealloc_b(line_context);
        line_context = 0;
        linebuf = 0;
        linecount = 0;
        linebuf_actuais = 0;
        linecount_actuais = 0;
    } else {
        /* ERROR, show the error or something.
           Free the line_context. */
        dwarf_srclines_dealloc_b(line_context);
    }
}
```

```
        line_context = 0;
        linebuf = 0;
        linecount = 0;
        linebuf_actuals = 0;
        linecount_actuals = 0;
    }
}
```

## 6.12 Line Context Details (DWARF5 style)

New in October 2015. When a `Dwarf_Line_Context` has been returned by `dwarf_srclines_b()` that line context data's details can be retrieved with the following set of calls.

### 6.12.1 `dwarf_srclines_table_offset()`

```
int dwarf_srclines_table_offset(Dwarf_Line_Context line_context,
    Dwarf_Unsigned * offset,
    Dwarf_Error * error);
```

On success, this function returns the offset (in the object file line section) of the actual line data (i.e. after the line header for this compilation unit) through the `offset` pointer. The offset is probably only of interest when printing detailed information about a line table header.

In case of error, `DW_DLV_ERROR` is returned and the error is set through the `error` pointer. `DW_DLV_NO_ENTRY` will not be returned.

### 6.12.2 `dwarf_srclines_version()`

```
int dwarf_srclines_version(Dwarf_Line_Context line_context,
    Dwarf_Unsigned * version,
    Dwarf_Error * error);
```

On success `DW_DLV_OK` is returned and the line table version number is returned through the `version` pointer.

In case of error, `DW_DLV_ERROR` is returned and the error is set through the `error` pointer. `DW_DLV_NO_ENTRY` will not be returned.

### 6.12.3 `dwarf_srclines_comp_dir()`

```
int dwarf_srclines_comp_dir(Dwarf_Line_Context line_context,
    const char ** compilation_directory,
    Dwarf_Error * error);
```

On success this returns a pointer to the compilation directory string for this line table in `*compilation_directory`. That compilation string may be NULL or the empty string. The string pointer is valid until the `line_context` has been deallocated.

In case of error, `DW_DLV_ERROR` is returned and the error is set through the error pointer. `DW_DLV_NO_ENTRY` will not be returned.

#### 6.12.4 dwarf\_srclines\_files\_indexes()

```
int dwarf_srclines_files_indexes(Dwarf_Line_Context line_context,
    Dwarf_Signed * baseindex,
    Dwarf_Signed * count,
    Dwarf_Signed * endindex,
    Dwarf_Error * error);
```

With DWARF5 the base file number index in the line table changed from zero (DWARF2,3,4) to one (DWARF5). Which meant iterating through the valid source file indexes became messy if one used the older `dwarf_srclines_files_count()` function (zero-based and one-based indexing being incompatible). See Figure "Examplec dwarf\_srclines\_b()" above for use of this function in accessing file names.

The base index of files in the files list of a line table header will be returned through `baseindex`.

The number of files in the files list of a line table header will be returned through `count`.

The end index of files in the files list of a line table header will be returned through `endindex`.

In case of error, `DW_DLV_ERROR` is returned and the error is set through the error pointer. `DW_DLV_NO_ENTRY` will not be returned.

#### 6.12.5 dwarf\_srclines\_files\_count()

```
int dwarf_srclines_files_count(Dwarf_Line_Context line_context,
    Dwarf_Signed * count,
    Dwarf_Error * error);
```

On success, the number of files in the files list of a line table header will be returned through `count`.

In case of error, DW\_DLV\_ERROR is returned and the error is set through the error pointer. DW\_DLV\_NO\_ENTRY will not be returned.

### 6.12.6 dwarf\_srclines\_files\_data\_b()

This supplants dwarf\_srclines\_files\_data() as of March 2018 to allow access to the md5 value in DWARF5. The function dwarf\_srclines\_files\_data() continues to be supported.

```
int dwarf_srclines_files_data_b(Dwarf_Line_Context line_context,
    Dwarf_Signed      index,
    const char **      name,
    Dwarf_Unsigned *   directory_index,
    Dwarf_Unsigned *   last_mod_time,
    Dwarf_Unsigned *   file_length,
    Dwarf_Form_Data16 ** md5_value,
    Dwarf_Error *      error);
```

On success, data about a single file in the files list will be returned through the pointers. See DWARF documentation for the meaning of these fields. count. Valid index values are 1 through count, reflecting the way the table is defined by DWARF2,3,4. For a dwarf5 line table index values 0...count-1 are legal. This is certainly awkward.

If md5\_value is non-null it is used to pass a back a pointer to a Dwarf\_Form\_Data16 md5 value if the md5 value is present. Otherwise a zero value is passed back to indicate there was no such field. The 16-byte value pointed to is inside the line\_context, so if you want to keep the value you should probably copy it to storage you control.

This returns the raw files data from the line table header.

In case of error, DW\_DLV\_ERROR is returned and the error is set through the error pointer. DW\_DLV\_NO\_ENTRY will not be returned.

### 6.12.7 dwarf\_srclines\_files\_data()

This interface was created in October 2015. It cannot return the DWARF5 MD5 value. See the newer dwarf\_srclines\_files\_data\_b().

```
int dwarf_srclines_files_data(Dwarf_Line_Context line_context,
    Dwarf_Signed      index,
    const char **      name,
    Dwarf_Unsigned *   directory_index,
    Dwarf_Unsigned *   last_mod_time,
    Dwarf_Unsigned *   file_length,
    Dwarf_Error *      error);
```

On success, data about a single file in the files list will be returned through the pointers.

See DWARF documentation for the meaning of these fields. `count`. Valid `index` values are 1 through `count`, reflecting the way the table is defined by DWARF2,3,4. For a dwarf5 line table index values 0...count-1 are legal. This is certainly awkward.

This returns the raw files data from the line table header.

In case of error, `DW_DLV_ERROR` is returned and the error is set through the `error` pointer. `DW_DLV_NO_ENTRY` will not be returned.

### 6.12.8 dwarf\_srclines\_include\_dir\_count()

```
int dwarf_srclines_include_dir_count(Dwarf_Line_Context line_context,
    Dwarf_Signed * count,
    Dwarf_Error * error);
```

On success, the number of files in the includes list of a line table header will be returned through `count`.

Valid `index`. values are 1 through `count`, reflecting the way the table is defined by DWARF 2,3 and 4. For a dwarf5 line table index values 0...count-1 are legal. This is certainly awkward.

In case of error, `DW_DLV_ERROR` is returned and the error is set through the `error` pointer. `DW_DLV_NO_ENTRY` will not be returned.

### 6.12.9 dwarf\_srclines\_include\_dir\_data()

```
int dwarf_srclines_include_dir_data(Dwarf_Line_Context line_context,
    Dwarf_Signed index,
    const char ** name,
    Dwarf_Error * error);
```

On success, data about a single file in the include files list will be returned through the pointers. See DWARF documentation for the meaning of these fields.

Valid `index`. values are 1 through `count`, reflecting the way the table is defined by DWARF.

In case of error, `DW_DLV_ERROR` is returned and the error is set through the `error` pointer. `DW_DLV_NO_ENTRY` will not be returned.

### 6.12.10 dwarf\_srclines\_subprog\_count()

```
int dwarf_srclines_subprog_count(Dwarf_Line_Context
line_context,
    Dwarf_Signed * count,
    Dwarf_Error * error);
```

This is only useful with experimental two-level line tables.

### 6.12.11 dwarf\_srclines\_subprog\_data()

```
int dwarf_srclines_subprog_data(Dwarf_Line_Context
line_context,
    Dwarf_Signed index,
    const char ** name,
    Dwarf_Unsigned * decl_file,
    Dwarf_Unsigned * decl_line,
    Dwarf_Error * error);
```

This is only useful with experimental two-level line tables.

## 6.13 Get A Set of Lines (DWARF2,3,4 style)

The function returns information about every source line for a particular compilation-unit. The compilation-unit is specified by the corresponding die. It does not support line tables with no lines very well nor does it support experimental two-level line tables.

### 6.13.1 dwarf\_srclines()

```
int dwarf_srclines(
    Dwarf_Die die,
    Dwarf_Line **linebuf,
    Dwarf_Signed *linecount,
    Dwarf_Error *error)
```

This function is not useful for DWARF5 skeleton line tables nor for two-level line tables. It works for DWARF2,3,4,5 ordinary single line tables. The function `dwarf_srclines()` places all line number descriptors for a single compilation unit into a single block, sets `*linebuf` to point to that block, sets `*linecount` to the number of descriptors in this block and returns `DW_DLV_OK`.

To get a more detailed view of the contents of a dwarf line table header see `dwarf_srclines_b()` and the routines that use the `Dwarf_Line_Context` information, such as `dwarf_srcfiles_comp_dir()`, `dwarf_srclines_files_count()`, `dwarf_srclines_include_dir_count()` and similar functions.

The compilation-unit is indicated by the given die which must be a compilation-unit die. It returns `DW_DLV_ERROR` on error. On successful return, line number information should be freed using `dwarf_srclines_dealloc()` when no longer of interest.

**Figure 21.** Exampled `dwarf_srclines()`



```
void exampled(Dwarf_Debug dbg,Dwarf_Die somedie)
{
    Dwarf_Signed count = 0;
    Dwarf_Line *linebuf = 0;
    Dwarf_Signed i = 0;
    Dwarf_Error error = 0;
    int sres = 0;

    sres = dwarf_srclines(somedie, &linebuf,&count, &error);
    if (sres == DW_DLV_OK) {
        for (i = 0; i < count; ++i) {
            /* use linebuf[i] */
        }
        dwarf_srclines_dealloc(dbg, linebuf, count);
    }
}
```

An alternative using `dwarf_dealloc()` directly is no longer (as of 2015) described here. It works as well as ever, but it has been obsolete since 2005. still works, but does not completely free all data allocated. The `dwarf_srclines_dealloc()` routine was created to fix the problem of incomplete deallocation.

## 6.14 Get the set of Source File Names

The function returns the names of the source files that have contributed to the compilation-unit represented by the given DIE. Only the source files named in the statement program prologue (which in current DWARF standards is referred to as the Line Table Header) are returned.

### 6.14.1 dwarf\_srcfiles()

This works for for all line tables.

```
int dwarf_srcfiles(
    Dwarf_Die die,
    char ***srcfiles,
    Dwarf_Signed *srccount,
    Dwarf_Error *error)
```

When it succeeds `dwarf_srcfiles()` returns `DW_DLV_OK` and puts the number of source files named in the statement program prologue indicated by the given die into `*srccount`. Source files defined in the statement program are ignored. The given die should have the tag `DW_TAG_compile_unit`, `DW_TAG_partial_unit`, or `DW_TAG_type_unit`. The location pointed to by `srcfiles` is set to point to a list of pointers to null-terminated strings that name the source files.

On a successful return from `dwarf_srcfiles()` each of the strings returned should

be individually freed using `dwarf_dealloc()` with the allocation type `DW_DLA_STRING` when no longer of interest. This should be followed by free-ing the list using `dwarf_dealloc()` with the allocation type `DW_DLA_LIST`. It returns `DW_DLV_ERROR` on error. It returns `DW_DLV_NO_ENTRY` if there is no corresponding statement program (i.e., if there is no line information).

It requires a little care to index into the `srcfiles` array using the `DW_AT_decl_file` or `DW_AT_decl_call_file` values.

The value zero (0) is reserved and means such zero-value attributes do not refer to any file name.

If value is non-zero subtract one (1) from the value and use that to index the `srcfiles` array. Of course being careful that the index is not too large for the array.

For DWARF5 an index of 0 (meaning value-1) refers to the compilation unit value identical to `DW_AT_name`. For earlier DWARF it refers to some file referenced in the compilation unit, but not any special file.

**Figure 22.** Exampled `dwarf_srcfiles()`

```
void examplee(Dwarf_Debug dbg,Dwarf_Die somedie)
{
    Dwarf_Signed count = 0;
    char **srcfiles = 0;
    Dwarf_Signed i = 0;
    Dwarf_Error error = 0;
    int res = 0;

    res = dwarf_srcfiles(somedie, &srcfiles,&count,&error);
    if (res == DW_DLV_OK) {
        for (i = 0; i < count; ++i) {
            /* use srcfiles[i] */
            dwarf_dealloc(dbg, srcfiles[i], DW_DLA_STRING);
        }
        dwarf_dealloc(dbg, srcfiles, DW_DLA_LIST);
    }
}
```

## 6.15 Get Information About a Single Line Table Line

The following functions can be used on the `Dwarf_Line` descriptors returned by `dwarf_srclines()` or `dwarf_srclines_from_linecontext()` to obtain information about the source lines.

### 6.15.1 `dwarf_linebeginstatement()`

```
int dwarf_linebeginstatement(  
    Dwarf_Line line,  
    Dwarf_Bool *return_bool,  
    Dwarf_Error *error)
```

The function `dwarf_linebeginstatement()` returns `DW_DLV_OK` and sets `*return_bool` to *non-zero* (if line represents a line number entry that is marked as beginning a statement). or *zero* ((if line represents a line number entry that is not marked as beginning a statement). It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

### 6.15.2 dwarf\_lineendsequence()

```
int dwarf_lineendsequence(  
    Dwarf_Line line,  
    Dwarf_Bool *return_bool,  
    Dwarf_Error *error)
```

The function `dwarf_lineendsequence()` returns `DW_DLV_OK` and sets `*return_bool` *non-zero* (in which case line represents a line number entry that is marked as ending a text sequence) or *zero* (in which case line represents a line number entry that is not marked as ending a text sequence). A line number entry that is marked as ending a text sequence is an entry with an address one beyond the highest address used by the current sequence of line table entries (that is, the table entry is a `DW_LNE_end_sequence` entry (see the DWARF specification)).

The function `dwarf_lineendsequence()` returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

### 6.15.3 dwarf\_lineno()

```
int dwarf_lineno(  
    Dwarf_Line line,  
    Dwarf_Unsigned * returned_lineno,  
    Dwarf_Error * error)
```

The function `dwarf_lineno()` returns `DW_DLV_OK` and sets `*return_lineno` to the source statement line number corresponding to the descriptor line. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

### 6.15.4 dwarf\_line\_srcfileno()

```
int dwarf_line_srcfileno(  
    Dwarf_Line      line,  
    Dwarf_Unsigned * returned_fileno,  
    Dwarf_Error     * error)
```

The function `dwarf_line_srcfileno()` returns `DW_DLV_OK` and sets `*returned_fileno` to the source statement line number corresponding to the descriptor file number. When the number returned through `*returned_fileno` is zero it means the file name is unknown (see the DWARF2/3 line table specification). When the number returned through `*returned_fileno` is non-zero it is a file number: subtract 1 from this file number to get an index into the array of strings returned by `dwarf_srcfiles()` (verify the resulting index is in range for the array of strings before indexing into the array of strings). The file number may exceed the size of the array of strings returned by `dwarf_srcfiles()` because `dwarf_srcfiles()` does not return files names defined with the `DW_DLE_define_file` operator. The function `dwarf_line_srcfileno()` returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

### 6.15.5 dwarf\_lineaddr()

```
int dwarf_lineaddr(  
    Dwarf_Line  line,  
    Dwarf_Addr  *return_lineaddr,  
    Dwarf_Error *error)
```

The function `dwarf_lineaddr()` returns `DW_DLV_OK` and sets `*return_lineaddr` to the address associated with the descriptor line. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

### 6.15.6 dwarf\_lineoff()

```
int dwarf_lineoff(  
    Dwarf_Line line,  
    Dwarf_Signed * return_lineoff,  
    Dwarf_Error *error)
```

The function `dwarf_lineoff()` returns `DW_DLV_OK` and sets `*return_lineoff` to the column number at which the statement represented by `line` begins.

It sets `return_lineoff` to zero if the column number of the statement is not represented (meaning the producer library call was given zero as the column number). Zero is the correct value meaning "left edge" as defined in the DWARF2/3/4 speciation (section 6.2.2).

Before December 2011 zero was not returned through the `return_lineoff` pointer, -1 was returned through the pointer. The reason for this oddity is unclear, lost in history.

But there is no good reason for -1.

The type of `return_lineoff` is a pointer-to-signed, but there is no good reason for the value to be signed, the DWARF specification does not deal with negative column numbers. However, changing the declaration would cause compilation errors for little benefit, so the pointer-to-signed is left unchanged.

On error it returns `DW_DLV_ERROR`. It never returns `DW_DLV_NO_ENTRY`.

### 6.15.7 `dwarf_lineoff_b()`

```
int dwarf_lineoff_b(  
    Dwarf_Line line,  
    Dwarf_Unsigned * return_lineoff,  
    Dwarf_Error *error)
```

The function `dwarf_lineoff_b()` returns exactly the same as `dwarf_lineoff()` except the line offset returned through `return_lineoff()` is an unsigned value. The signed return offset never made much sense but was harmless since line lengths are limited by most language standards.

### 6.15.8 `dwarf_linesrc()`

```
int dwarf_linesrc(  
    Dwarf_Line line,  
    char ** return_linesrc,  
    Dwarf_Error *error)
```

The function `dwarf_linesrc()` returns `DW_DLV_OK` and sets `*return_linesrc` to a pointer to a null-terminated string of characters that represents the name of the source-file where `line` occurs. It returns `DW_DLV_ERROR` on error.

If the applicable file name in the line table Statement Program Prolog does not start with a `'/'` character the string in `DW_AT_comp_dir` (if applicable and present) or the applicable directory name from the line Statement Program Prolog is prepended to the file name in the line table Statement Program Prolog to make a full path.

The storage pointed to by a successful return of `dwarf_linesrc()` should be freed using `dwarf_dealloc()` with the allocation type `DW_DLA_STRING` when no longer of interest. It never returns `DW_DLV_NO_ENTRY`.

### 6.15.9 `dwarf_lineblock()`

```
int dwarf_lineblock(  
    Dwarf_Line line,  
    Dwarf_Bool *return_bool,  
    Dwarf_Error *error)
```

The function `dwarf_lineblock()` returns `DW_DLV_OK` and sets `*return_linesrc` to non-zero (i.e. true) if the line is marked as beginning a basic block) or zero (i.e. false) (if the line is marked as not beginning a basic block). It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

#### 6.15.10 dwarf\_is\_addr\_set()

```
int dwarf_line_is_addr_set(  
    Dwarf_Line line,  
    Dwarf_Bool *return_bool,  
    Dwarf_Error *error)
```

The function `dwarf_line_is_addr_set()` returns `DW_DLV_OK` and sets `*return_bool` to non-zero (i.e. true) if the line is marked as being a `DW_LNE_set_address` operation) or zero (i.e. false) (if the line is marked as not being a `DW_LNE_set_address` operation). It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

This is intended to allow consumers to do a more useful job printing and analyzing DWARF data, it is not strictly necessary.

#### 6.15.11 dwarf\_prologue\_end\_etc()

```
int dwarf_prologue_end_etc(Dwarf_Line line,  
    Dwarf_Bool *prologue_end,  
    Dwarf_Bool *epilogue_begin,  
    Dwarf_Unsigned *isa,  
    Dwarf_Unsigned *discriminator,  
    Dwarf_Error *error)
```

The function `dwarf_prologue_end_etc()` returns `DW_DLV_OK` and sets the returned fields to values currently set. While it is pretty safe to assume that the `isa` and `discriminator` values returned are very small integers, there is no restriction in the standard. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

This function is new in December 2011.

## 6.16 Accelerated Access By Name operations

These operations operate on the `.debug_pubnames` section as well as all the other sections with this specific format and purpose:

`.debug_pubtypes`,  
`.debug_typednames`,  
`.debug_varnames`,  
`.debug_funcnames`,  
and `.debug_weaknames`. The first in the list is generic DWARF 2,3,4. The second in the list is generic DWARF 3,4. The rest are SGI specific and rarely used.

The interface types are `Dwarf_Global`, `Dwarf_Type`, `Dwarf_Weak`, `Dwarf_Func`, and `Dwarf_Var`. Only `Dwarf_Global` is a real type. The others are opaque pointers with no actual definition or instantiation and can be converted to `Dwarf_Global` with a simple cast.

In hindsight it would have been simpler to write a single set of interfaces for Accelerated Access By Name.

### 6.16.1 Fine Tuning Accelerated Access

By default the various `dwarf_get*()` functions here return an array of pointers to opaque records with a `.debug_info` DIE offset and a string (the fields are accessible by function calls). While the actual `.debug_pubnames` (etc) section contains CU-local DIE offsets for the named things the accelerated access functions below return a `.debug_info` (or `.debug_types`) global section offset.

#### 6.16.1.1 `dwarf_return_empty_pubnames`

New March 2019. Mostly special for `dwarfdump`. If called with a flag value of one (1) it tells `libdwarf`, for any `pubnames`(etc) section list returned to add to the list an entry with a global-DIE-offset of zero (0) for any section Compilation Unit entry with no `pubnames`(etc) name( ie, an empty list for the Compilation Unit).

If called with a value of zero(0) (zero is the default set by any `dwarf_init*()` call) it causes such empty lists to be omitted from the array of pointers returned, which is the standard behavior of `libdwarf` since `libdwarf` was first written.

Since zero is never a valid DIE offset in `.debug_info` (or `.debug_types`) consumers requesting such can detect the special `Dwarf_Global` entries.

For example, calling

`dwarf_global_name_offsets()` on one of the special global records sets `*die_offset` to 0, `*return_name` to a pointer to an empty string, and `*cu_offset` to the offset of the compilation unit die in the `.debug_info` (or `.debug_types` if applicable) section.

```
int dwarf_return_empty_pubnames(Dwarf_Debug dbg,
    int flag,
    Dwarf_Error* error)
```

Callers should pass in one (1) or zero(0), no other value. On success it returns DW\_DLV\_OK. On failure it returns DW\_DLV\_ERROR;

The assumption is that programs calling this with value one (1) will be calling dwarf\_get\_globals\_header() to retrieve the relevant pubnames(etc) section Compilation Unit header.

#### 6.16.1.2 dwarf\_get\_globals\_header

New February 2019. For more complete dwarfdump printing. For each CU represented in .debug\_pubnames, etc, there is a .debug\_pubnames header. For any given Dwarf\_Global this returns the content of the applicable header.

This allows dwarfdump, or any DWARF dumper, to print pubnames(etc) specific CU header data.

```
int dwarf_get_globals_header(Dwarf_Global global,
    Dwarf_Off * offset_pub_header,
    Dwarf_Unsigned * offset_size,
    Dwarf_Unsigned * length_pub,
    Dwarf_Unsigned * version,
    Dwarf_Unsigned * header_info_offset,
    Dwarf_Unsigned * info_length,
    Dwarf_Error* error)
```

On success it returns DW\_DLV\_OK and it returns the header data (and calculated values) through the pointers. Casting Dwarf\_Type (etc) to Dwarf\_Global for a call to this function allows this to be used for any of these accelerated-access types.

### 6.16.2 Accelerated Access Pubnames

#### 6.16.2.1 dwarf\_get\_globals()

This is .debug\_pubnames and is standard DWARF2, DWARF3, and DWARF4.

```
int dwarf_get_globals(
    Dwarf_Debug dbg,
    Dwarf_Global **globals,
    Dwarf_Signed * return_count,
    Dwarf_Error *error)
```

The function dwarf\_get\_globals() returns DW\_DLV\_OK and sets \*return\_count to the count of pubnames represented in the section containing



pubnames i.e. `.debug_pubnames`. It also stores at `*globals`, a pointer to a list of `Dwarf_Global` descriptors, one for each of the pubnames in the `.debug_pubnames` section. The returned results are for the entire section. It returns `DW_DLV_ERROR` on error. It returns `DW_DLV_NO_ENTRY` if the `.debug_pubnames` section does not exist.

On a successful return from `dwarf_get_globals()`, the `Dwarf_Global` descriptors should be freed using `dwarf_globals_dealloc()`. `dwarf_globals_dealloc()` is new as of July 15, 2005 and is the preferred approach to freeing this memory..

Global names refer exclusively to names and offsets in the `.debug_info` section. See section 6.1.1 "Lookup by Name" in the dwarf standard.

**Figure 23.** Exampled `dwarf_get_globals()`

```
void examplef(Dwarf_Debug dbg)
{
    Dwarf_Signed count = 0;
    Dwarf_Global *globs = 0;
    Dwarf_Signed i = 0;
    Dwarf_Error error = 0;
    int res = 0;

    res = dwarf_get_globals(dbg, &globs, &count, &error);
    if (res == DW_DLV_OK) {
        for (i = 0; i < count; ++i) {
            /* use globs[i] */
        }
        dwarf_globals_dealloc(dbg, globs, count);
    }
}
```

The following code is deprecated as of July 15, 2005 as it does not free all relevant memory. This approach still works as well as it ever did. On a successful return from `dwarf_get_globals()`, the `Dwarf_Global` descriptors should be individually freed using `dwarf_dealloc()` with the allocation type `DW_DLA_GLOBAL_CONTEXT`, (or `DW_DLA_GLOBAL`, an older name, supported for compatibility) followed by the deallocation of the list itself with the allocation type `DW_DLA_LIST` when the descriptors are no longer of interest.

```
Dwarf_Signed cnt;
Dwarf_Global *globs;
int res;

res = dwarf_get_globals(dbg, &globs, &cnt, &error);
if (res == DW_DLV_OK) {

    /* OBSOLETE: DO NOT USE to deallocate*/
    for (i = 0; i < cnt; ++i) {
        /* use globs[i] */
        dwarf_dealloc(dbg, globs[i], DW_DLA_GLOBAL_CONTEXT);
    }
    dwarf_dealloc(dbg, globs, DW_DLA_LIST);
}
```

#### 6.16.2.2 dwarf\_globname()

```
int dwarf_globname(
    Dwarf_Global global,
    char **      return_name,
    Dwarf_Error *error)
```

The function `dwarf_globname()` returns `DW_DLV_OK` and sets `*return_name` to a pointer to a null-terminated string that names the pubname represented by the `Dwarf_Global` descriptor, `global`. It returns `DW_DLV_ERROR` on error. On a successful return from this function, the string should be freed using `dwarf_dealloc()`, with the allocation type `DW_DLA_STRING` when no longer of interest. It never returns `DW_DLV_NO_ENTRY`.

#### 6.16.2.3 dwarf\_global\_die\_offset()

```
int dwarf_global_die_offset(
    Dwarf_Global global,
    Dwarf_Off  *return_offset,
    Dwarf_Error *error)
```

The function `dwarf_global_die_offset()` returns `DW_DLV_OK` and sets `*return_offset` to the offset in the section containing DIEs, i.e. `.debug_info`, of the DIE representing the pubname that is described by the `Dwarf_Global` descriptor, `glob`. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

#### 6.16.2.4 dwarf\_global\_cu\_offset()

```
int dwarf_global_cu_offset(  
    Dwarf_Global global,  
    Dwarf_Off *return_offset,  
    Dwarf_Error *error)
```

The function `dwarf_global_cu_offset()` returns `DW_DLV_OK` and sets `*return_offset` to the offset in the section containing DIEs, i.e. `.debug_info`, of the compilation-unit header of the compilation-unit that contains the pubname described by the `Dwarf_Global` descriptor, `global`. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

#### 6.16.2.5 dwarf\_get\_cu\_die\_offset\_given\_cu\_header\_offset()

```
int dwarf_get_cu_die_offset_given_cu_header_offset_b(  
    Dwarf_Debug dbg,  
    Dwarf_Off in_cu_header_offset,  
    Dwarf_Bool is_info,  
    Dwarf_Off * out_cu_die_offset,  
    Dwarf_Error *error)
```

The function `dwarf_get_cu_die_offset_given_cu_header_offset()` returns `DW_DLV_OK` and sets `*out_cu_die_offset` to the offset of the compilation-unit DIE given the offset `in_cu_header_offset` of a compilation-unit header. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

If `is_info` is non-zero the `in_cu_header_offset` must refer to a `.debug_info` section offset. If `is_info` zero the `in_cu_header_offset` must refer to a `.debug_types` section offset. Chaos may result if the `is_info` flag is incorrect.

This effectively turns a compilation-unit-header offset into a compilation-unit DIE offset (by adding the size of the applicable CU header). This function is also sometimes useful with the `dwarf_weak_cu_offset()`, `dwarf_func_cu_offset()`, `dwarf_type_cu_offset()`, and `int dwarf_var_cu_offset()` functions, though for those functions the data is only in `.debug_info` by definition.

#### 6.16.2.6 dwarf\_get\_cu\_die\_offset\_given\_cu\_header\_offset()

```
int dwarf_get_cu_die_offset_given_cu_header_offset(  
    Dwarf_Debug dbg,  
    Dwarf_Off in_cu_header_offset,  
    Dwarf_Off * out_cu_die_offset,  
    Dwarf_Error *error)
```

This function is superseded by `dwarf_get_cu_die_offset_given_cu_header_offset_b()`, a function which is still supported though it refers only to the `.debug_info` section.

`dwarf_get_cu_die_offset_given_cu_header_offset()` added Rev 1.45, June, 2001.

This function was declared as 'optional' in `libdwarf.h` on IRIX systems so the `_MIPS_SYMBOL_PRESENT` predicate could be used at run time to determine if the version of `libdwarf` linked into an application has this function.

#### 6.16.2.7 `dwarf_global_name_offsets()`

```
int dwarf_global_name_offsets(  
    Dwarf_Global global,  
    char          **return_name,  
    Dwarf_Off     *die_offset,  
    Dwarf_Off     *cu_die_offset,  
    Dwarf_Error   *error)
```

The function `dwarf_global_name_offsets()` returns `DW_DLV_OK` and sets `*return_name` to a pointer to a null-terminated string that gives the name of the pubname described by the `Dwarf_Global` descriptor `global`. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`. It also returns in the locations pointed to by `die_offset`, and `cu_offset`, the global offset of the DIE representing the pubname, and the offset of the DIE representing the compilation-unit containing the pubname, respectively. On a successful return from `dwarf_global_name_offsets()` the storage pointed to by `return_name` should be freed using `dwarf_dealloc()`, with the allocation type `DW_DLA_STRING` when no longer of interest.

If a portion of `.debug_pubnames` ( or `.debug_types` etc) represents a compilation unit with no names there is a `.debug_pubnames` header there with no content. In that case a single `Dwarf_Global` record is created with the value of `*die_offset` zero and the name-pointer returned points to the empty string. A zero is never a valid DIE offset, so zero always means this is an uninteresting (`Dwarf_Global`).

### 6.16.3 Accelerated Access Pubtypes

Section `".debug_pubtypes"` is in DWARF3 and DWARF4.

These functions operate on the `.debug_pubtypes` section of the debugging information. The `.debug_pubtypes` section contains the names of file-scope user-defined types, the offsets of the DIEs that represent the definitions of those types, and the offsets of the compilation-units that contain the definitions of those types.

#### 6.16.3.1 `dwarf_get_pubtypes()`

This is standard DWARF3 and DWARF4.

```
int dwarf_get_pubtypes(  
    Dwarf_Debug dbg,  
    Dwarf_Type **types,  
    Dwarf_Signed *typecount,  
    Dwarf_Error *error)
```

The function `dwarf_get_pubtypes()` returns `DW_DLV_OK` and sets `*typecount` to the count of user-defined type names represented in the section containing user-defined type names, i.e. `.debug_pubtypes`. It also stores at `*types`, a pointer to a list of `Dwarf_Type` descriptors, one for each of the user-defined type names in the `.debug_pubtypes` section. The returned results are for the entire section. It returns `DW_DLV_NOCOUNT` on error. It returns `DW_DLV_NO_ENTRY` if the `.debug_pubtypes` section does not exist.

On a successful return from `dwarf_get_pubtypes()`, the `Dwarf_Type` descriptors should be freed using `dwarf_types_dealloc()`. `dwarf_types_dealloc()` is used for both `dwarf_get_pubtypes()` and `dwarf_get_types()` as the data types are the same.

Global type names refer exclusively to names and offsets in the `.debug_info` section. See section 6.1.1 "Lookup by Name" in the dwarf standard.

**Figure 24.** Exemplified `dwarf_get_pubtypes()`

```
void exampleleg(Dwarf_Debug dbg)  
{  
    Dwarf_Error error = 0;  
    Dwarf_Signed count = 0;  
    Dwarf_Type *types = 0;  
    Dwarf_Signed i = 0;  
    int res = 0;  
  
    res = dwarf_get_pubtypes(dbg, &types, &count, &error);  
    if (res == DW_DLV_OK) {  
        for (i = 0; i < count; ++i) {  
            /* use types[i] */  
        }  
        dwarf_types_dealloc(dbg, types, count);  
    }  
}
```

### 6.16.3.2 `dwarf_pubtypename()`

```
int dwarf_pubtypename(  
    Dwarf_Type type,  
    char **return_name,  
    Dwarf_Error *error)
```

The function `dwarf_pubtypename()` returns `DW_DLV_OK` and sets `*return_name` to a pointer to a null-terminated string that names the user-defined type represented by the `Dwarf_Type` descriptor, `type`. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`. On a successful return from this function, the string should be freed using `dwarf_dealloc()`, with the allocation type `DW_DLA_STRING` when no longer of interest.

#### 6.16.3.3 dwarf\_pubtype\_type\_die\_offset()

```
int dwarf_pubtype_type_die_offset(  
    Dwarf_Type type,  
    Dwarf_Off *return_offset,  
    Dwarf_Error *error)
```

The function `dwarf_pubtype_type_die_offset()` returns `DW_DLV_OK` and sets `*return_offset` to the offset in the section containing DIEs, i.e. `.debug_info`, of the DIE representing the user-defined type that is described by the `Dwarf_Type` descriptor, `type`. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

#### 6.16.3.4 dwarf\_pubtype\_cu\_offset()

```
int dwarf_pubtype_cu_offset(  
    Dwarf_Type type,  
    Dwarf_Off *return_offset,  
    Dwarf_Error *error)
```

The function `dwarf_pubtype_cu_offset()` returns `DW_DLV_OK` and sets `*return_offset` to the offset in the section containing DIEs, i.e. `.debug_info`, of the compilation-unit header of the compilation-unit that contains the user-defined type described by the `Dwarf_Type` descriptor, `type`. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

#### 6.16.3.5 dwarf\_pubtype\_name\_offsets()

```
int dwarf_pubtype_name_offsets(  
    Dwarf_Type    type,  
    char          ** returned_name,  
    Dwarf_Off *   die_offset,  
    Dwarf_Off *   cu_offset,  
    Dwarf_Error *error)
```

The function `dwarf_pubtype_name_offsets()` returns `DW_DLV_OK` and sets `*returned_name` to a pointer to a null-terminated string that gives the name of the user-defined type described by the `Dwarf_Type` descriptor `type`. It also returns in the locations pointed to by `die_offset`, and `cu_offset`, the offsets of the DIE representing the user-defined type, and the DIE representing the compilation-unit containing the user-defined type, respectively. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`. On a successful return from `dwarf_pubtype_name_offsets()` the storage pointed to by `returned_name` should be freed using `dwarf_dealloc()`, with the allocation type `DW_DLA_STRING` when no longer of interest.

#### 6.16.4 Accelerated Access Weaknames

This section is SGI specific and is not part of standard DWARF.

These functions operate on the `.debug_varnames` section of the debugging information. The `.debug_varnames` section contains the names of file-scope static variables, the offsets of the DIEs that represent the definitions of those variables, and the offsets of the compilation-units that contain the definitions of those variables.

These operations operate on the `.debug_weaknames` section of the debugging information.

##### 6.16.4.1 dwarf\_get\_weak()

```
int dwarf_get_weak(  
    Dwarf_Debug dbg,  
    Dwarf_Weak **weak,  
    Dwarf_Signed *weak_count,  
    Dwarf_Error *error)
```

The function `dwarf_get_weak()` returns `DW_DLV_OK` and sets `*weak_count` to the count of weak names represented in the section containing weak names i.e. `.debug_weaknames`. It returns `DW_DLV_ERROR` on error. It returns `DW_DLV_NO_ENTRY` if the section does not exist. It also stores in `*weak`, a pointer to a list of `Dwarf_Weak` descriptors, one for each of the weak names in the `.debug_weaknames` section. The returned results are for the entire section.

On a successful return from this function, the Dwarf\_Weak descriptors should be freed using dwarf\_weak\_dealloc() when the data is no longer of interest. dwarf\_weak\_dealloc() is new as of July 15, 2005.

**Figure 25.** Exampleh dwarf\_get\_weak()

```
void exampleh(Dwarf_Debug dbg)
{
    Dwarf_Error error = 0;
    Dwarf_Signed count = 0;
    Dwarf_Weak *weak = 0;
    Dwarf_Signed i = 0;
    int res = 0;

    res = dwarf_get_weak(dbg, &weak, &count, &error);
    if (res == DW_DLV_OK) {
        for (i = 0; i < count; ++i) {
            /* use weak[i] */
        }
        dwarf_weak_dealloc(dbg, weak, count);
    }
}
```

The following code is deprecated as of July 15, 2005 as it does not free all relevant memory. This approach still works as well as it ever did. On a successful return from dwarf\_get\_weak() the Dwarf\_Weak descriptors should be individually freed using dwarf\_dealloc() with the allocation type DW\_DLA\_WEAK\_CONTEXT, (or DW\_DLA\_WEAK, an older name, supported for compatibility) followed by the deallocation of the list itself with the allocation type DW\_DLA\_LIST when the descriptors are no longer of interest.

**Figure 26.** Examplei dwarf\_get\_weak() obsolete



```
void examplei(Dwarf_Debug dbg)
{
    Dwarf_Error error = 0;
    Dwarf_Signed count = 0;
    Dwarf_Weak *weak = 0;
    Dwarf_Signed i = 0;
    int res = 0;

    /* Obsolete, see exampleh instead */
    res = dwarf_get_weak(dbg, &weak, &count, &error);
    if (res == DW_DLV_OK) {
        /* OBSOLETE: do not use dealloc for this.
           See above */
        for (i = 0; i < count; ++i) {
            /* use weak[i] */
            dwarf_dealloc(dbg, weak[i], DW_DLA_WEAK);
        }
        dwarf_dealloc(dbg, weak, DW_DLA_LIST);
    }
}
```

#### 6.16.4.2 dwarf\_weakname()

```
int dwarf_weakname(
    Dwarf_Weak weak,
    char ** return_name,
    Dwarf_Error *error)
```

The function `dwarf_weakname()` returns `DW_DLV_OK` and sets `*return_name` to a pointer to a null-terminated string that names the weak name represented by the `Dwarf_Weak` descriptor, `weak`. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`. On a successful return from this function, the string should be freed using `dwarf_dealloc()`, with the allocation type `DW_DLA_STRING` when no longer of interest.

```
int dwarf_weak_die_offset(
    Dwarf_Weak weak,
    Dwarf_Off *return_offset,
    Dwarf_Error *error)
```

The function `dwarf_weak_die_offset()` returns `DW_DLV_OK` and sets `*return_offset` to the offset in the section containing DIEs, i.e. `.debug_info`, of the DIE representing the weak name that is described by the `Dwarf_Weak` descriptor, `weak`. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

#### 6.16.4.3 dwarf\_weak\_cu\_offset()

```
int dwarf_weak_cu_offset(  
    Dwarf_Weak weak,  
    Dwarf_Off *return_offset,  
    Dwarf_Error *error)
```

The function `dwarf_weak_cu_offset()` returns `DW_DLV_OK` and sets `*return_offset` to the offset in the section containing DIEs, i.e. `.debug_info`, of the compilation-unit header of the compilation-unit that contains the weak name described by the `Dwarf_Weak` descriptor, `weak`. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

#### 6.16.4.4 dwarf\_weak\_name\_offsets()

```
int dwarf_weak_name_offsets(  
    Dwarf_Weak weak,  
    char ** weak_name,  
    Dwarf_Off *die_offset,  
    Dwarf_Off *cu_offset,  
    Dwarf_Error *error)
```

The function `dwarf_weak_name_offsets()` returns `DW_DLV_OK` and sets `*weak_name` to a pointer to a null-terminated string that gives the name of the weak name described by the `Dwarf_Weak` descriptor `weak`. It also returns in the locations pointed to by `die_offset`, and `cu_offset`, the offsets of the DIE representing the weakname, and the DIE representing the compilation-unit containing the weakname, respectively. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`. On a successful return from `dwarf_weak_name_offsets()` the storage pointed to by `weak_name` should be freed using `dwarf_dealloc()`, with the allocation type `DW_DLA_STRING` when no longer of interest.

### 6.16.5 Accelerated Access Funcnames

This section is SGI specific and is not part of standard DWARF.

These function operate on the `.debug_funcnames` section of the debugging information. The `.debug_funcnames` section contains the names of static functions defined in the object, the offsets of the DIEs that represent the definitions of the corresponding functions, and the offsets of the start of the compilation-units that contain the definitions of those functions.

### 6.16.5.1 dwarf\_get\_funcs()

```
int dwarf_get_funcs(  
    Dwarf_Debug dbg,  
    Dwarf_Func **funcs,  
    Dwarf_Signed *func_count,  
    Dwarf_Error *error)
```

The function `dwarf_get_funcs()` returns `DW_DLV_OK` and sets `*func_count` to the count of static function names represented in the section containing static function names, i.e. `.debug_funcnames`. It also stores, at `*funcs`, a pointer to a list of `Dwarf_Func` descriptors, one for each of the static functions in the `.debug_funcnames` section. The returned results are for the entire section. It returns `DW_DLV_ERROR` on error. It returns `DW_DLV_NO_ENTRY` if the `.debug_funcnames` section does not exist.

On a successful return from `dwarf_get_funcs()`, the `Dwarf_Func` descriptors should be freed using `dwarf_funcs_dealloc()`. `dwarf_funcs_dealloc()` is new as of July 15, 2005.

**Figure 27.** Examplej `dwarf_get_funcs()`

```
void examplej(Dwarf_Debug dbg)  
{  
    Dwarf_Error error = 0;  
    Dwarf_Signed count = 0;  
    Dwarf_Func *funcs = 0;  
    Dwarf_Signed i = 0;  
    int fres = 0;  
  
    fres = dwarf_get_funcs(dbg, &funcs, &count, &error);  
    if (fres == DW_DLV_OK) {  
        for (i = 0; i < count; ++i) {  
            /* use funcs[i] */  
        }  
        dwarf_funcs_dealloc(dbg, funcs, count);  
    }  
}
```

The following code is deprecated as of July 15, 2005 as it does not free all relevant memory. This approach still works as well as it ever did. On a successful return from `dwarf_get_funcs()`, the `Dwarf_Func` descriptors should be individually freed using `dwarf_dealloc()` with the allocation type `DW_DLA_FUNC_CONTEXT`, (or `DW_DLA_FUNC`, an older name, supported for compatibility) followed by the deallocation of the list itself with the allocation type `DW_DLA_LIST` when the

descriptors are no longer of interest.

**Figure 28.** Examplek dwarf\_get\_funcs() obsolete

```
void examplek(Dwarf_Debug dbg)
{
    Dwarf_Error error = 0;
    Dwarf_Func *funcs = 0;
    Dwarf_Signed count = 0;
    Dwarf_Signed i = 0;
    int fres = 0;

    fres = dwarf_get_funcs(dbg, &funcs, &count, &error);
    if (fres == DW_DLV_OK) {
        /* OBSOLETE: see dwarf_funcs_dealloc() above */
        for (i = 0; i < count; ++i) {
            /* use funcs[i] */
            dwarf_dealloc(dbg, funcs[i], DW_DLA_FUNC);
        }
        dwarf_dealloc(dbg, funcs, DW_DLA_LIST);
    }
}
```

#### 6.16.5.2 dwarf\_funcname()

```
int dwarf_funcname(
    Dwarf_Func func,
    char **      return_name,
    Dwarf_Error *error)
```

The function `dwarf_funcname()` returns `DW_DLV_OK` and sets `*return_name` to a pointer to a null-terminated string that names the static function represented by the `Dwarf_Func` descriptor, `func`. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`. On a successful return from this function, the string should be freed using `dwarf_dealloc()`, with the allocation type `DW_DLA_STRING` when no longer of interest.

#### 6.16.5.3 dwarf\_func\_die\_offset()

```
int dwarf_func_die_offset(
    Dwarf_Func func,
    Dwarf_Off *return_offset,
    Dwarf_Error *error)
```

The function `dwarf_func_die_offset()`, returns `DW_DLV_OK` and sets `*return_offset` to the offset in the section containing DIEs, i.e. `.debug_info`, of the DIE representing the static function that is described by the `Dwarf_Func` descriptor, `func`. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

#### 6.16.5.4 `dwarf_func_cu_offset()`

```
int dwarf_func_cu_offset(  
    Dwarf_Func func,  
    Dwarf_Off  *return_offset,  
    Dwarf_Error *error)
```

The function `dwarf_func_cu_offset()` returns `DW_DLV_OK` and sets `*return_offset` to the offset in the section containing DIEs, i.e. `.debug_info`, of the compilation-unit header of the compilation-unit that contains the static function described by the `Dwarf_Func` descriptor, `func`. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

#### 6.16.5.5 `dwarf_func_name_offsets()`

```
int dwarf_func_name_offsets(  
    Dwarf_Func func,  
    char      **func_name,  
    Dwarf_Off  *die_offset,  
    Dwarf_Off  *cu_offset,  
    Dwarf_Error *error)
```

The function `dwarf_func_name_offsets()` returns `DW_DLV_OK` and sets `*func_name` to a pointer to a null-terminated string that gives the name of the static function described by the `Dwarf_Func` descriptor `func`. It also returns in the locations pointed to by `die_offset`, and `cu_offset`, the offsets of the DIE representing the static function, and the DIE representing the compilation-unit containing the static function, respectively. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`. On a successful return from `dwarf_func_name_offsets()` the storage pointed to by `func_name` should be freed using `dwarf_dealloc()`, with the allocation type `DW_DLA_STRING` when no longer of interest.

### 6.16.6 Accelerated Access Typenames

Section `"debug_typenames"` is SGI specific and is not part of standard DWARF. (However, an identical section is part of DWARF version 3 named `".debug_pubtypes"`, see `dwarf_get_pubtypes()` above.)

These functions operate on the `.debug_typenames` section of the debugging information.

The `.debug_typednames` section contains the names of file-scope user-defined types, the offsets of the DIEs that represent the definitions of those types, and the offsets of the compilation-units that contain the definitions of those types.

#### 6.16.6.1 `dwarf_get_types()`

```
int dwarf_get_types(  
    Dwarf_Debug dbg,  
    Dwarf_Type **types,  
    Dwarf_Signed *typecount,  
    Dwarf_Error *error)
```

The function `dwarf_get_types()` returns `DW_DLV_OK` and sets `*typecount` to the count of user-defined type names represented in the section containing user-defined type names, i.e. `.debug_typednames`. It also stores at `*types`, a pointer to a list of `Dwarf_Type` descriptors, one for each of the user-defined type names in the `.debug_typednames` section. The returned results are for the entire section. It returns `DW_DLV_NOCOUNT` on error. It returns `DW_DLV_NO_ENTRY` if the `.debug_typednames` section does not exist.

On a successful return from `dwarf_get_types()`, the `Dwarf_Type` descriptors should be freed using `dwarf_types_dealloc()`. `dwarf_types_dealloc()` is new as of July 15, 2005 and frees all memory allocated by `dwarf_get_types()`.

**Figure 29.** Example1 `dwarf_get_types()`

```
void example1(Dwarf_Debug dbg)  
{  
    Dwarf_Error error = 0;  
    Dwarf_Signed count = 0;  
    Dwarf_Type *types = 0;  
    Dwarf_Signed i = 0;  
    int res = 0;  
  
    res = dwarf_get_types(dbg, &types, &count, &error);  
    if (res == DW_DLV_OK) {  
        for (i = 0; i < count; ++i) {  
            /* use types[i] */  
        }  
        dwarf_types_dealloc(dbg, types, count);  
    }  
}
```

The following code is deprecated as of July 15, 2005 as it does not free all relevant memory. This approach still works as well as it ever did. On a successful return from `dwarf_get_types()`, the `Dwarf_Type` descriptors should be individually freed using `dwarf_dealloc()` with the allocation type `DW_DLA_TYPPENAME_CONTEXT`, (or `DW_DLA_TYPPENAME`, an older name, supported for compatibility) followed by the deallocation of the list itself with the allocation type `DW_DLA_LIST` when the descriptors are no longer of interest.

**Figure 30.** Example1 dwarf\_get\_types() obsolete

```
void example1(Dwarf_Debug dbg)
{
    Dwarf_Error error = 0;
    Dwarf_Signed count = 0;
    Dwarf_Type *types = 0;
    Dwarf_Signed i = 0;
    int res = 0;

    /* OBSOLETE: see dwarf_types_dealloc() above */
    res = dwarf_get_types(dbg, &types, &count, &error);
    if (res == DW_DLV_OK) {
        for (i = 0; i < count; ++i) {
            /* use types[i] */
            dwarf_dealloc(dbg, types[i], DW_DLA_TYPPENAME);
        }
        dwarf_dealloc(dbg, types, DW_DLA_LIST);
    }
}
```

#### 6.16.6.2 dwarf\_typename()

```
int dwarf_typename(
    Dwarf_Type    type,
    char          **return_name,
    Dwarf_Error   *error)
```

The function `dwarf_typename()` returns `DW_DLV_OK` and sets `*return_name` to a pointer to a null-terminated string that names the user-defined type represented by the `Dwarf_Type` descriptor, `type`. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`. On a successful return from this function, the string should be freed using `dwarf_dealloc()`, with the allocation type `DW_DLA_STRING` when no longer of interest.

#### 6.16.6.3 dwarf\_type\_die\_offset()

```
int dwarf_type_die_offset(  
    Dwarf_Type type,  
    Dwarf_Off *return_offset,  
    Dwarf_Error *error)
```

The function `dwarf_type_die_offset()` returns `DW_DLV_OK` and sets `*return_offset` to the offset in the section containing DIEs, i.e. `.debug_info`, of the DIE representing the user-defined type that is described by the `Dwarf_Type` descriptor, `type`. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

#### 6.16.6.4 dwarf\_type\_cu\_offset()

```
int dwarf_type_cu_offset(  
    Dwarf_Type type,  
    Dwarf_Off *return_offset,  
    Dwarf_Error *error)
```

The function `dwarf_type_cu_offset()` returns `DW_DLV_OK` and sets `*return_offset` to the offset in the section containing DIEs, i.e. `.debug_info`, of the compilation-unit header of the compilation-unit that contains the user-defined type described by the `Dwarf_Type` descriptor, `type`. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

#### 6.16.6.5 dwarf\_type\_name\_offsets()

```
int dwarf_type_name_offsets(  
    Dwarf_Type type,  
    char ** returned_name,  
    Dwarf_Off * die_offset,  
    Dwarf_Off * cu_offset,  
    Dwarf_Error *error)
```

The function `dwarf_type_name_offsets()` returns `DW_DLV_OK` and sets `*returned_name` to a pointer to a null-terminated string that gives the name of the user-defined type described by the `Dwarf_Type` descriptor `type`. It also returns in the locations pointed to by `die_offset`, and `cu_offset`, the offsets of the DIE representing the user-defined type, and the DIE representing the compilation-unit containing the user-defined type, respectively. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`. On a successful return from `dwarf_type_name_offsets()` the storage pointed to by `returned_name` should be freed using `dwarf_dealloc()`, with the allocation type `DW_DLA_STRING` when no longer of interest.



## 6.16.7 Accelerated Access varnames

This section is SGI specific and is not part of standard DWARF.

These functions operate on the `.debug_varnames` section of the debugging information. The `.debug_varnames` section contains the names of file-scope static variables, the offsets of the DIEs that represent the definitions of those variables, and the offsets of the compilation-units that contain the definitions of those variables.

### 6.16.7.1 dwarf\_get\_vars()

```
int dwarf_get_vars(  
    Dwarf_Debug dbg,  
    Dwarf_Var **vars,  
    Dwarf_Signed *var_count,  
    Dwarf_Error *error)
```

The function `dwarf_get_vars()` returns `DW_DLV_OK` and sets `*var_count` to the count of file-scope static variable names represented in the section containing file-scope static variable names, i.e. `.debug_varnames`. It also stores, at `*vars`, a pointer to a list of `Dwarf_Var` descriptors, one for each of the file-scope static variable names in the `.debug_varnames` section. The returned results are for the entire section. It returns `DW_DLV_ERROR` on error. It returns `DW_DLV_NO_ENTRY` if the `.debug_varnames` section does not exist.

The following is new as of July 15, 2005. On a successful return from `dwarf_get_vars()`, the `Dwarf_Var` descriptors should be freed using `dwarf_vars_dealloc()`.

**Figure 31.** Exampelen `dwarf_get_vars()`

```
void examplen(Dwarf_Debug dbg)
{
    Dwarf_Error error = 0;
    Dwarf_Signed count = 0;
    Dwarf_Var *vars = 0;
    Dwarf_Signed i = 0;
    int res = 0;
    res = dwarf_get_vars(dbg, &vars, &count, &error);
    if (res == DW_DLV_OK) {
        for (i = 0; i < count; ++i) {
            /* use vars[i] */
        }
        dwarf_vars_dealloc(dbg, vars, count);
    }
}
```

The following code is deprecated as of July 15, 2005 as it does not free all relevant memory. This approach still works as well as it ever did. On a successful return from `dwarf_get_vars()`, the `Dwarf_Var` descriptors should be individually freed using `dwarf_dealloc()` with the allocation type `DW_DLA_VAR_CONTEXT`, (or `DW_DLA_VAR`, an older name, supported for compatibility) followed by the deallocation of the list itself with the allocation type `DW_DLA_LIST` when the descriptors are no longer of interest.

**Figure 32.** Example of `dwarf_get_vars()` obsolete

```
void exampleo(Dwarf_Debug dbg)
{
    Dwarf_Error error = 0;
    Dwarf_Signed count = 0;
    Dwarf_Var *vars = 0;
    Dwarf_Signed i = 0;
    int res = 0;
    res = dwarf_get_vars(dbg, &vars, &count, &error);
    if (res == DW_DLV_OK) {
        /* DO NOT USE: see dwarf_vars_dealloc() above */
        for (i = 0; i < count; ++i) {
            /* use vars[i] */
            dwarf_dealloc(dbg, vars[i], DW_DLA_VAR);
        }
        dwarf_dealloc(dbg, vars, DW_DLA_LIST);
    }
}
```

#### 6.16.7.2 dwarf\_varname()

```
int dwarf_varname(  
    Dwarf_Var var,  
    char **    returned_name,  
    Dwarf_Error *error)
```

The function `dwarf_varname()` returns `DW_DLV_OK` and sets `*returned_name` to a pointer to a null-terminated string that names the file-scope static variable represented by the `Dwarf_Var` descriptor, `var`. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`. On a successful return from this function, the string should be freed using `dwarf_dealloc()`, with the allocation type `DW_DLA_STRING` when no longer of interest.

#### 6.16.7.3 dwarf\_var\_die\_offset()

```
int dwarf_var_die_offset(  
    Dwarf_Var    var,  
    Dwarf_Off    *returned_offset,  
    Dwarf_Error  *error)
```

The function `dwarf_var_die_offset()` returns `DW_DLV_OK` and sets `*returned_offset` to the offset in the section containing DIEs, i.e. `.debug_info`, of the DIE representing the file-scope static variable that is described by the `Dwarf_Var` descriptor, `var`. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

#### 6.16.7.4 dwarf\_var\_cu\_offset()

```
int dwarf_var_cu_offset(  
    Dwarf_Var var,  
    Dwarf_Off *returned_offset,  
    Dwarf_Error *error)
```

The function `dwarf_var_cu_offset()` returns `DW_DLV_OK` and sets `*returned_offset` to the offset in the section containing DIEs, i.e. `.debug_info`, of the compilation-unit header of the compilation-unit that contains the file-scope static variable described by the `Dwarf_Var` descriptor, `var`. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

#### 6.16.7.5 dwarf\_var\_name\_offsets()

```
int dwarf_var_name_offsets(  
    Dwarf_Var var,  
    char      **returned_name,  
    Dwarf_Off *die_offset,  
    Dwarf_Off *cu_offset,  
    Dwarf_Error *error)
```

The function `dwarf_var_name_offsets()` returns `DW_DLV_OK` and sets `*returned_name` to a pointer to a null-terminated string that gives the name of the file-scope static variable described by the `Dwarf_Var` descriptor `var`. It also returns in the locations pointed to by `die_offset`, and `cu_offset`, the offsets of the DIE representing the

representing the compilation-unit containing the file-scope static variable, respectively. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`. On a successful return from `dwarf_var_name_offsets()` the storage pointed to by `returned_name` should be freed using `dwarf_dealloc()`, with the allocation type `DW_DLA_STRING` when no longer of interest.

## 6.17 Names Fast Access (DWARF5) `.debug_names`

The section `.debug_names` section is new in DWARF5 so a new set of functions is defined to access this section. This section replaces `.debug_pubnames` and `.debug_pubtypes` as those older sections were not found to be useful in practice.

### 6.17.1 `dwarf_debugnames_header()`

```
int dwarf_debugnames_header(  
    Dwarf_Debug      dbg,  
    Dwarf_Dnames_Head * dn_out,  
    Dwarf_Unsigned   * dn_index_count_out,  
    Dwarf_Error *error)
```

The function `dwarf_debugnames_header()` allocates an opaque data structure used in all the other `debugnames` calls.

Many of the function calls here let one extract the entire content of the section, which is useful if one wishes to dump the section or to use its data to create one's own internal data structures.

To free space allocated when one has finished with these data structures, call

```
Debug_Dnames_Head dn /* Assume set somehow */;  
...  
dwarf_dealloc(dbg,dn,DW_DLA_DNAMES_HEAD);
```

which will free up all data allocated for `dwarf_debugnames_header()`.

FIXME describe arguments.

```
void exampledebugnames(void)
{
    FIXME need extended example of debugnames use.
}
```

### 6.17.2 dwarf\_debugnames\_sizes()

```
int dwarf_debugnames_sizes(Dwarf_Dnames_Head dn,
    Dwarf_Unsigned    index_number,

    Dwarf_Unsigned * section_offsets,
    Dwarf_Unsigned * version,
    Dwarf_Unsigned * offset_size,

    /* The counts are entry counts, not byte sizes. */
    Dwarf_Unsigned * comp_unit_count,
    Dwarf_Unsigned * local_type_unit_count,
    Dwarf_Unsigned * foreign_type_unit_count,
    Dwarf_Unsigned * bucket_count,
    Dwarf_Unsigned * name_count,

    /* The following are counted in bytes */
    Dwarf_Unsigned * indextable_overall_length,
    Dwarf_Unsigned * abbrev_table_size,
    Dwarf_Unsigned * entry_pool_size,
    Dwarf_Unsigned * augmentation_string_size,

    Dwarf_Error *    error*/)
```

Allows access to fields in a .debug\_names DWARF5 header record. FIXME

### 6.17.3 dwarf\_debugnames\_cu\_entry()

```
int dwarf_debugnames_cu_entry(  
    Dwarf_Dnames_Head dn,  
    Dwarf_Unsigned      index_number,  
    Dwarf_Unsigned      offset_number,  
    Dwarf_Unsigned      * offset_count,  
    Dwarf_Unsigned      * offset,  
    Dwarf_Error *        error)
```

Allows access to fields in cu entry from a .debug\_names DWARF5 compilation unit entry. FIXME

#### **6.17.4 dwarf\_debugnames\_local\_tu\_entry()**

```
int dwarf_debugnames_local_tu_entry(  
    Dwarf_Dnames_Head dn,  
    Dwarf_Unsigned      index_number,  
    Dwarf_Unsigned      offset_number,  
    Dwarf_Unsigned      * offset_count,  
    Dwarf_Unsigned      * offset,  
    Dwarf_Error *        error)
```

FIXME

#### **6.17.5 dwarf\_debugnames\_foreign\_tu\_entry()**

```
int dwarf_debugnames_foreign_tu_entry(  
    Dwarf_Dnames_Head dn,  
    Dwarf_Unsigned      index_number,  
    Dwarf_Unsigned      sig_number,  
    Dwarf_Unsigned      * sig_minimum,  
    Dwarf_Unsigned      * sig_count,  
    Dwarf_Sig8          * signature,  
    Dwarf_Error *        error)
```

FIXME

#### **6.17.6 dwarf\_debugnames\_bucket()**

```
int dwarf_debugnames_bucket(  
    Dwarf_Dnames_Head dn,  
    Dwarf_Unsigned      index_number,  
    Dwarf_Unsigned      bucket_number,  
    Dwarf_Unsigned      * bucket_count,  
    Dwarf_Unsigned      * index_of_name_entry,  
    Dwarf_Error *        error)
```

FIXME

### 6.17.7 dwarf\_debugnames\_name()

```
int dwarf_debugnames_bucket(  
    Dwarf_Dnames_Head dn,  
    Dwarf_Unsigned      index_number,  
    Dwarf_Unsigned      name_entry,  
    Dwarf_Unsigned      * names_count,  
    Dwarf_Sig8          * signature,  
    Dwarf_Unsigned      * offset_to_debug_str,  
    Dwarf_Unsigned      * offset_in_entrypool,  
    Dwarf_Error *        error)
```

FIXME

### 6.17.8 dwarf\_debugnames\_abbrev\_by\_index()

```
int dwarf_debugnames_abbrev_by_index(  
    Dwarf_Dnames_Head dn,  
    Dwarf_Unsigned      index_number,  
    Dwarf_Unsigned      abbrev_entry,  
    Dwarf_Unsigned      * abbrev_code,  
    Dwarf_Unsigned      * tag,  
    Dwarf_Unsigned      * number_of_abbrev,  
    Dwarf_Unsigned      * number_of_attr_form_entries,  
    Dwarf_Error *        error)
```

FIXME

### 6.17.9 dwarf\_debugnames\_abbrev\_by\_code()

```
int dwarf_debugnames_abbrev_by_code(  
    Dwarf_Dnames_Head dn,  
    Dwarf_Unsigned      index_number,  
    Dwarf_Unsigned      abbrev_code,  
    Dwarf_Unsigned      * tag,  
    Dwarf_Unsigned      * index_of_abbrev,  
    Dwarf_Unsigned      * index_of_attr_form_entries,  
    Dwarf_Error *      error)
```

FIXME

#### **6.17.10 dwarf\_debugnames\_form\_by\_index()**

```
int dwarf_debugnames_form_by_index(  
    Dwarf_Dnames_Head dn,  
    Dwarf_Unsigned      index_number,  
    Dwarf_Unsigned      abbrev_entry_index,  
    Dwarf_Unsigned      abbrev_form_index,  
    Dwarf_Unsigned      * name_attr_index,  
    Dwarf_Unsigned      * form,  
    Dwarf_Unsigned      * number_of_attr_form_entries,  
    Dwarf_Error *      error)
```

FIXME

#### **6.17.11 dwarf\_debugnames\_entrypool()**

```
int dwarf_debugnames_entrypool(  
    Dwarf_Dnames_Head dn,  
    Dwarf_Unsigned      index_number,  
    Dwarf_Unsigned      offset_in_entrypool,  
    Dwarf_Unsigned      * abbrev_code,  
    Dwarf_Unsigned      * tag,  
    Dwarf_Unsigned      * value_count,  
    Dwarf_Unsigned      * index_of_abbrev,  
    Dwarf_Unsigned      * offset_of_initial_value,  
    Dwarf_Error *      error)
```

FIXME

#### **6.17.12 dwarf\_debugnames\_entrypool\_values()**



```
int dwarf_debugnames_entrypool_values(  
    Dwarf_Dnames_Head dn,  
    Dwarf_Unsigned      index_number,  
    Dwarf_Unsigned      index_of_abbrev,  
    Dwarf_Unsigned      offset_in_entrypool_of_values,  
    Dwarf_Unsigned      * array_dw_idx_number,  
    Dwarf_Unsigned      * array_form,  
    Dwarf_Unsigned      * array_of_offsets,  
    Dwarf_Sig8          * array_of_signatures,  
    Dwarf_Error *       error)
```

FIXME

## 6.18 Macro Information Operations (DWARF4, DWARF5)

This section refers to DWARF4 and later macro information from the `.debug_macro` section (for DWARF 4 some producers generated `.debug_macro` before its formal standardization in DWARF 5). While standard operations are supported there is as yet no support for implementation-defined extensions. Once someone has defined such things it will make sense to design an interface for extensions.

### 6.18.1 Getting access

The opaque struct pointer `Dwarf_Macro_Context` is allocated by either `dwarf_get_macro_context()` or `dwarf_get_macro_context_by_offset()` and once the context is no longer needed one frees up all its storage by `dwarf_dealloc_macro_context()`.

#### 6.18.1.1 dwarf\_get\_macro\_context()

```
int dwarf_get_macro_context(Dwarf_Die die,  
    Dwarf_Unsigned      * version_out,  
    Dwarf_Macro_Context * macro_context,  
    Dwarf_Unsigned      * macro_unit_offset_out,  
    Dwarf_Unsigned      * macro_ops_count_out,  
    Dwarf_Unsigned      * macro_ops_data_length_out,  
    Dwarf_Error          * error);
```

Given a Compilation Unit (CU) die, on success `dwarf_get_macro_context()` opens a `Dwarf_Macro_Context` and returns a pointer to it and some data from the macro unit for that CU. The `Dwarf_Macro_Context` is used to get at the details of the macros.

The value `version_out` is set to the DWARF version number of the macro data. Version 5 means DWARF5 version information. Version 4 means the DWARF5 format macro data is present as an extension of DWARF4.

The value `macro_unit_offset_out` is set to the offset in the `.debug_macro` section of the first byte of macro data for this CU.

The value `macro_ops_count_out` is set to the number of macro entries in the macro data for this CU. The count includes the final zero entry (which is not really a macro, it is a terminator, a zero byte ending the macro unit).

The value `macro_ops_data_length_out` is set to the number of bytes of data in the macro unit, including the macro unit header.

If `DW_DLV_NO_ENTRY` is returned the CU has no macro data attribute or there is no `.debug_macro` section present.

On error `DW_DLV_ERROR` is returned and the error details are returned through the pointer `error`.

#### 6.18.1.2 `dwarf_get_macro_context_by_offset()`

```
int dwarf_get_macro_context_by_offset(Dwarf_Die die,
    Dwarf_Unsigned      offset,
    Dwarf_Unsigned      * version_out,
    Dwarf_Macro_Context * macro_context,
    Dwarf_Unsigned      * macro_ops_count_out,
    Dwarf_Unsigned      * macro_ops_total_byte_len,
    Dwarf_Error          * error);
```

Given a Compilation Unit (CU) die and the offset of an imported macro unit `dwarf_get_macro_context_by_offset()` opens a `Dwarf_Macro_Context` and returns a pointer to it and some data from the macro unit for that CU on success.

On success the function produces the same output values as `dwarf_get_macro_context()` except there is no offset returned ( the caller provides it).

If `DW_DLV_NO_ENTRY` is returned there is no `.debug_macro` section present.

On error `DW_DLV_ERROR` is returned and the error details are returned through the pointer `error`.

#### 6.18.1.3 `dwarf_dealloc_macro_context()`

```
void dwarf_dealloc_macro_context (Dwarf_Macro_Context macro_context);
```

The function `dwarf_dealloc_macro_context()` cleans up memory allocated by a successful call to `dwarf_get_macro_context()` or `dwarf_get_macro_context_by_offset()`.

**Figure 33.** Examplep5 `dwarf_dealloc_macro_context()`

```
/* This builds an list or some other data structure
   (not defined) to give an import somewhere to list
   the import offset and then later to enquire
   if the list has unexamined offsets.
   A candidate set of hypothetical functions that
   callers would write:
   has_unchecked_import_in_list()
   get_next_import_from_list()
   mark_this_offset_as_examined(macro_unit_offset);
   add_offset_to_list(offset);
*/
void examplep5(Dwarf_Debug dbg, Dwarf_Die cu_die)
{
    int lres = 0;
    Dwarf_Unsigned version = 0;
    Dwarf_Macro_Context macro_context = 0;
    Dwarf_Unsigned macro_unit_offset = 0;
    Dwarf_Unsigned number_of_ops = 0;
    Dwarf_Unsigned ops_total_byte_len = 0;
    Dwarf_Bool is_primary = TRUE;
    unsigned k = 0;
    Dwarf_Error err = 0;

    for(;;) {
        if (is_primary) {
            lres = dwarf_get_macro_context(cu_die,
                                           &version, &macro_context,
                                           &macro_unit_offset,
                                           &number_of_ops,
                                           &ops_total_byte_len,
                                           &err);
            is_primary = FALSE;
        } else {
            if (has_unchecked_import_in_list()) {
                macro_unit_offset = get_next_import_from_list();
            } else {
                /* We are done */
                break;
            }
            lres = dwarf_get_macro_context_by_offset(cu_die,
                                                    macro_unit_offset,
                                                    &version,
                                                    &macro_context,
                                                    &number_of_ops,
                                                    &ops_total_byte_len,
                                                    &err);
        }
    }
}
```

```
        mark_this_offset_as_examined(macro_unit_offset);
    }

    if (lres == DW_DLV_ERROR) {
        /* Something is wrong. */
        return;
    }
    if (lres == DW_DLV_NO_ENTRY) {
        /* We are done. */
        break;
    }
    /* lres == DW_DLV_OK) */
    for (k = 0; k < number_of_ops; ++k) {
        Dwarf_Unsigned section_offset = 0;
        Dwarf_Half      macro_operator = 0;
        Dwarf_Half      forms_count = 0;
        const Dwarf_Small *formcode_array = 0;
        Dwarf_Unsigned line_number = 0;
        Dwarf_Unsigned index = 0;
        Dwarf_Unsigned offset = 0;
        const char      *macro_string = 0;
        int lres = 0;

        lres = dwarf_get_macro_op(macro_context,
                                k, &section_offset, &macro_operator,
                                &forms_count, &formcode_array, &err);
        if (lres != DW_DLV_OK) {
            print_error(dbg,
                       "ERROR from dwarf_get_macro_op()",
                       lres, err);
            dwarf_dealloc_macro_context(macro_context);
            return;
        }
        switch(macro_operator) {
        case 0:
            /* Nothing to do. */
            break;
        case DW_MACRO_end_file:
            /* Do something */
            break;
        case DW_MACRO_define:
        case DW_MACRO_undef:
        case DW_MACRO_define_strp:
        case DW_MACRO_undef_strp:
        case DW_MACRO_define_strx:
        case DW_MACRO_undef_strx:
```

```
case DW_MACRO_define_sup:
case DW_MACRO_undef_sup: {
    lres = dwarf_get_macro_defundef (macro_context,
        k,
        &line_number,
        &index,
        &offset,
        &forms_count,
        &macro_string,
        &err);
    if (lres != DW_DLV_OK) {
        print_error(dbg,
            "ERROR from sup dwarf_get_macro_defundef()",
            lres,err);
        dwarf_dealloc_macro_context (macro_context);
        return;
    }
    /* do something */
}
break;
case DW_MACRO_start_file: {
    lres = dwarf_get_macro_startend_file (macro_context,
        k,&line_number,
        &index,
        &macro_string,&err);
    if (lres != DW_DLV_OK) {
        print_error(dbg,
            "ERROR from dwarf_get_macro_startend_file() (sup)",
            lres,err);
        dwarf_dealloc_macro_context (macro_context);
        return;
    }
    /* do something */
}
break;
case DW_MACRO_import: {
    lres = dwarf_get_macro_import (macro_context,
        k,&offset,&err);
    if (lres != DW_DLV_OK) {
        print_error(dbg,
            "ERROR from dwarf_get_macro_import() (sup)",
            lres,err);
        dwarf_dealloc_macro_context (macro_context);
        return;
    }
    add_offset_to_list (offset);
```

```
    }
    break;
case DW_MACRO_import_sup: {
    lres = dwarf_get_macro_import (macro_context,
                                   k, &offset, &err);
    if (lres != DW_DLV_OK) {
        print_error (dbg,
                    "ERROR from dwarf_get_macro_import() (sup)",
                    lres, err);
        dwarf_dealloc_macro_context (macro_context);
        return;
    }
    /* do something */
}
break;
}
}
dwarf_dealloc_macro_context (macro_context);
macro_context = 0;
}
}
```

## 6.18.2 Getting Macro Unit Header Data

### 6.18.2.1 dwarf\_macro\_context\_head()

```
int dwarf_macro_context_head (Dwarf_Macro_Context macro_context,
                             Dwarf_Half          * version,
                             Dwarf_Unsigned       * mac_offset,
                             Dwarf_Unsigned       * mac_len,
                             Dwarf_Unsigned       * mac_header_len,
                             unsigned             * flags,
                             Dwarf_Bool           * has_line_offset,
                             Dwarf_Unsigned       * line_offset,
                             Dwarf_Bool           * has_offset_size_64,
                             Dwarf_Bool           * has_operands_table,
                             Dwarf_Half          * opcode_count,
                             Dwarf_Error          * error);
```

Given a Dwarf\_Macro\_Context pointer this function returns the basic fields of a macro unit header (Macro Information Header) on success.

The value version is set to the DWARF version number of the macro unit header.

Version 5 means DWARF5 version information. Version 4 means the DWARF5 format macro data is present as an extension of DWARF4.

The value `mac_offset` is set to the offset in the `.debug_macro` section of the first byte of macro data for this CU.

The value `mac_len` is set to the number of bytes of data in the macro unit, including the macro unit header.

The value `mac_header_len` is set to the number of bytes in the macro unit header (not a field that is generally useful).

The value `flags` is set to the value of the `flags` field of the macro unit header.

The value `has_line_offset` is set to non-zero if the `debug_line_offset_flag` bit is set in the `flags` field of the macro unit header. If `has_line_offset` is set then `line_offset` is set to the value of the `debug_line_offset` field in the macro unit header. If `has_line_offset` is not set there is no `debug_line_offset` field present in the macro unit header.

The value `has_offset_size_64` is set non-zero if the `offset_size_flag` bit is set in the `flags` field of the macro unit header and in this case offset fields in this macro unit are 64 bits. If `has_offset_size_64` is not set then offset fields in this macro unit are 32 bits.

The value `has_operands_table` is set to non-zero if the `opcode_operands_table_flag` bit is set in the `flags` field of the macro unit header.

If `has_operands_table` is set non-zero then The value `opcode_count` is set to the number of opcodes in the macro unit header `opcode_operands_table`. See `dwarf_get_macro_op()`.

`DW_DLV_NO_ENTRY` is not returned.

On error `DW_DLV_ERROR` is returned and the error details are returned through the pointer `error`.

#### 6.18.2.2 `dwarf_macro_operands_table()`

```
int dwarf_macro_operands_table(Dwarf_Macro_Context macro_context,
    Dwarf_Half index, /* 0 to opcode_count -1 */
    Dwarf_Half * opcode_number,
    Dwarf_Half * operand_count,
    const Dwarf_Small ** operand_array,
    Dwarf_Error * error);
```

`dwarf_macro_operands_table()` is used to index through the operands table in a macro unit header if the operands table exists in the macro unit header. The operands table provides the mechanism for implementations to add extensions to the macro operations while allowing clients to skip macro operations the client code does not



recognize.

The `macro_context` field passed in identifies the macro unit involved. The `index` field passed in identifies which macro operand to look at. Valid index values are zero through the `opcode_count-1` (returned by `dwarf_macro_context_head()`).

The `opcode_number` value returned through the pointer is the the macro operation code. The operation code could be one of the standard codes or if there are user extensions there would be an extension code in the `DW_MACRO_lo_user` to `DW_MACRO_hi_user` range.

The `operand_count` returned is the number of form codes in the form codes array of unsigned bytes `operand_array`.

`DW_DLV_NO_ENTRY` is not returned.

On error `DW_DLV_ERROR` is returned and the error details are returned through the pointer `error`.

### 6.18.3 Getting Individual Macro Operations Data

#### 6.18.3.1 `dwarf_get_macro_op()`

```
int dwarf_get_macro_op(Dwarf_Macro_Context macro_context,
    Dwarf_Unsigned op_number,
    Dwarf_Unsigned * op_start_section_offset,
    Dwarf_Half     * macro_operator,
    Dwarf_Half     * forms_count,
    const Dwarf_Small ** formcode_array,
    Dwarf_Error    * error);
```

Use `dwarf_get_macro_op()` to access the macro operations of this macro unit.

The `macro_context` field passed in identifies the macro unit involved. The `op_number` field passed in identifies which macro operand to look at. Valid index values are zero through `macro_ops_count_out-1` (field returned by `dwarf_get_macro_context()` or `dwarf_get_macro_context_by_offset()`)

On success the function returns values through the pointers.

The `op_start_section_offset` returned is useful for debugging but otherwise is not normally useful. It is the byte offset of the beginning of this macro operator's data.

The `macro_operator` returned is one of the defined macro operations such as `DW_MACRO_define`. This is the field you will use to choose what call to use to get the data for a macro operator. For example, for `DW_MACRO_undef` one would call `dwarf_get_macro_defundef()` (see below) to get the details about the undefine.

The `forms_count` returned is useful for debugging but otherwise is not normally useful. It is the number of bytes of form numbers in the `formcode_array` of this macro operator's applicable forms.

`DW_DLV_NO_ENTRY` is not returned.

On error `DW_DLV_ERROR` is returned and the error details are returned through the pointer `error`.

### 6.18.3.2 `dwarf_get_macro_defundef()`

```
int dwarf_get_macro_defundef(Dwarf_Macro_Context macro_context,
    Dwarf_Unsigned op_number,
    Dwarf_Unsigned * line_number,
    Dwarf_Unsigned * index,
    Dwarf_Unsigned * offset,
    Dwarf_Half * forms_count,
    const char ** macro_string,
    Dwarf_Error * error);
```

Call `dwarf_get_macro_defundef` for any of the macro define/undefine operators. Which fields are set through the pointers depends on the particular operator.

The `macro_context` field passed in identifies the macro unit involved. The `op_number` field passed in identifies which macro operand to look at. Valid index values are zero through `macro_ops_count_out-1` (field returned by `dwarf_get_macro_context()` or `dwarf_get_macro_context_by_offset()`).

The `line_number` field is set with the source line number of the macro.

The `index` field only set meaningfully if the macro operator is `DW_MACRO_define_strx` or `DW_MACRO_undef_strx`. If set it is an index into an array of offsets in the `.debug_str_offsets` section.

The `offset` field only set meaningfully if the macro operator is `DW_MACRO_define_strx`, `DW_MACRO_undef_strx`, `DW_MACRO_define_strp`, or `DW_MACRO_undef_strp`. If set it is an offset of a string in the `.debug_str` section.

The `forms_count` is set to the number of forms that apply to the macro operator.

The `macro_string` pointer is used to return a pointer to the macro string. If the actual string cannot be found (as when section with the string is in a different object, see `set_tied_dbg()`) the string returned may be "<:No string available>" or "<.debug\_str\_offsets not available>" (without the quotes).

The function returns `DW_DLV_NO_ENTRY` if the macro operation is not one of the define/undef operations.

On error DW\_DLV\_ERROR is returned and the error details are returned through the pointer error.

#### 6.18.3.3 dwarf\_get\_macro\_startend\_file()

```
int dwarf_get_macro_startend_file(Dwarf_Macro_Context macro_context,  
    Dwarf_Unsigned op_number,  
    Dwarf_Unsigned * line_number,  
    Dwarf_Unsigned * name_index_to_line_tab,  
    const char ** src_file_name,  
    Dwarf_Error * error);
```

Call dwarf\_get\_macro\_startend\_file for operators DW\_MACRO\_start\_file or DW\_MACRO\_end\_file.

The macro\_context field passed in identifies the macro unit involved.

The op\_number field passed in identifies which macro operand to look at. Valid index values are zero through macro\_ops\_count\_out-1 (field returned by dwarf\_get\_macro\_context() or dwarf\_get\_macro\_context\_by\_offset()).

For DW\_MACRO\_end\_file none of the following fields are set on successful return, they are only set for DW\_MACRO\_start\_file

The line\_number field is set with the source line number of the macro.

The name\_index\_to\_line\_tab field is set with the index into the file name table of the line table section. For DWARF2, DWARF3, DWARF4 line tables the index value assumes DWARF2 line table header rules (identical to DWARF3, DWARF4 line table header rules). For DWARF5 the index value assumes DWARF5 line table header rules. The src\_file\_name is set with the source file name. If the index seems wrong or the line table is unavailable the name returned is "<no-source-file-name-available>";

The function returns DW\_DLV\_NO\_ENTRY if the macro operation is not one of the start/end operations.

On error DW\_DLV\_ERROR is returned and the error details are returned through the pointer error.

#### 6.18.3.4 dwarf\_get\_macro\_import()

```
int dwarf_get_macro_import(Dwarf_Macro_Context macro_context,  
    Dwarf_Unsigned op_number,  
    Dwarf_Unsigned * target_offset,  
    Dwarf_Error * error);
```

Call dwarf\_get\_macro\_import for operators DW\_MACRO\_import or

DW\_MACRO\_import\_sup.

The `macro_context` field passed in identifies the macro unit involved. The `op_number` field passed in identifies which macro operand to look at. Valid index values are zero through `macro_ops_count_out-1` (field returned by `dwarf_get_macro_context()` or `dwarf_get_macro_context_by_offset()`)

On success the `target_offset` field is set to the offset in the referenced section. For DW\_MACRO\_import the referenced section is the same section as the macro operation referenced here. For DW\_MACRO\_import\_sup the referenced section is in a supplementary object.

The function returns DW\_DLV\_NO\_ENTRY if the macro operation is not one of the import operations.

On error DW\_DLV\_ERROR is returned and the error details are returned through the pointer `error`.

## 6.19 Macro Information Operations (DWARF2, DWARF3, DWARF4)

This section refers to DWARF2, DWARF3, and DWARF4 macro information from the `.debug_macinfo` section. These do not apply to DWARF5 macro data.

### 6.19.1 General Macro Operations

#### 6.19.1.1 `dwarf_find_macro_value_start()`

```
char *dwarf_find_macro_value_start(char * macro_string);
```

Given a macro string in the standard form defined in the DWARF document ("name <space> value" or "name(args)<space>value") this returns a pointer to the first byte of the macro value. It does not alter the string pointed to by `macro_string` or copy the string: it returns a pointer into the string whose address was passed in.

### 6.19.2 Debugger Interface Macro Operations

Macro information is accessed from the `.debug_info` section via the DW\_AT\_macro\_info attribute (whose value is an offset into `.debug_macinfo`).

No Functions yet defined.

### 6.19.3 Low Level Macro Information Operations

#### 6.19.3.1 `dwarf_get_macro_details()`

```
int dwarf_get_macro_details(Dwarf_Debug /*dbg*/,
    Dwarf_Off          macro_offset,
    Dwarf_Unsigned      maximum_count,
    Dwarf_Signed        * entry_count,
    Dwarf_Macro_Details ** details,
    Dwarf_Error *       err);
```

`dwarf_get_macro_details()` returns `DW_DLV_OK` and sets `entry_count` to the number of details records returned through the `details` pointer. The data returned through `details` should be freed by a call to `dwarf_dealloc()` with the allocation type `DW_DLA_STRING`. If `DW_DLV_OK` is returned, the `entry_count` will be at least 1, since a compilation unit with macro information but no macros will have at least one macro data byte of 0.

`dwarf_get_macro_details()` begins at the `macro_offset` offset you supply and ends at the end of a compilation unit or at `maximum_count` detail records (whichever comes first). If `maximum_count` is 0, it is treated as if it were the maximum possible unsigned integer.

`dwarf_get_macro_details()` attempts to set `dmd_fileindex` to the correct file in every details record. If it is unable to do so (or whenever the current file index is unknown, it sets `dmd_fileindex` to -1.

`dwarf_get_macro_details()` returns `DW_DLV_ERROR` on error. It returns `DW_DLV_NO_ENTRY` if there is no more macro information at that `macro_offset`. If `macro_offset` is passed in as 0, a `DW_DLV_NO_ENTRY` return means there is no macro information.

**Figure 34.** Examplep2 `dwarf_get_macro_details()`

```
void examplep2(Dwarf_Debug dbg, Dwarf_Off cur_off)
{
    Dwarf_Error error = 0;
    Dwarf_Signed count = 0;
    Dwarf_Macro_Details *maclist = 0;
    Dwarf_Signed i = 0;
    Dwarf_Unsigned max = 500000; /* sanity limit */
    int errv = 0;

    /* Given an offset from a compilation unit,
       start at that offset (from DW_AT_macroinfo)
       and get its macro details. */
    errv = dwarf_get_macro_details(dbg, cur_off, max,
                                   &count, &maclist, &error);
    if (errv == DW_DLV_OK) {
        for (i = 0; i < count; ++i) {
            Dwarf_Macro_Details * mentry = maclist + i;
            /* example of use */
            Dwarf_Signed lineno = mentry->dmd_lineno;
            functionusingsigned(lineno);
        }
        dwarf_dealloc(dbg, maclist, DW_DLA_STRING);
    }
    /* Loop through all the compilation units macro info from zero.
       This is not guaranteed to work because DWARF does not
       guarantee every byte in the section is meaningful:
       there can be garbage between the macro info
       for CUs. But this loop will sometimes work.
    */
    cur_off = 0;
    while((errv = dwarf_get_macro_details(dbg, cur_off, max,
                                           &count, &maclist, &error)) == DW_DLV_OK) {
        for (i = 0; i < count; ++i) {
            Dwarf_Macro_Details * mentry = maclist + i;
            /* example of use */
            Dwarf_Signed lineno = mentry->dmd_lineno;
            functionusingsigned(lineno);
        }
        cur_off = maclist[count-1].dmd_offset + 1;
        dwarf_dealloc(dbg, maclist, DW_DLA_STRING);
    }
}
```

## 6.20 Low Level Frame Operations

These functions provide information about stack frames to be used to perform stack traces. The information is an abstraction of a table with a row per instruction and a column per register and a column for the canonical frame address (CFA, which corresponds to the notion of a frame pointer), as well as a column for the return address.

From 1993-2006 the interface we'll here refer to as DWARF2 made the CFA be a column in the matrix, but left `DW_FRAME_UNDEFINED_VAL`, and `DW_FRAME_SAME_VAL` out of the matrix (giving them high numbers). As of the DWARF3 interfaces introduced in this document in April 2006, there are *\*two\** interfaces (the original set and a new set). Several frame functions work transparently for either set, we will focus on the ones that are not equally suitable now.

The original DWARF2 interface set still exists (`dwarf_get_fde_info_for_reg()`, `dwarf_get_fde_info_for_cfa_reg()`, and `dwarf_get_fde_info_for_all_regs()`) and works adequately for MIPS/IRIX DWARF2 and ABI/ISA sets that are sufficiently similar to MIPS. These functions not a good choice for non-MIPS architectures nor were they a good design for MIPS either. It's better to switch entirely to the new functions mentioned in the next paragraph. This DWARF2 interface set assumes and uses `DW_FRAME_CFA_COL` and that is assumed when `libdwarf` is configured with `--enable-oldframecol`.

A new DWARF3 interface set of `dwarf_get_fde_info_for_reg3()`, `dwarf_get_fde_info_for_cfa_reg3()`, `dwarf_get_fde_info_for_all_regs3()`, `dwarf_set_frame_rule_table_size()`, `dwarf_set_frame_cfa_value()`, `dwarf_set_frame_same_value()`, `dwarf_set_frame_undefined_value()`, and `dwarf_set_frame_rule_initial_value()` is more flexible and will work for many more architectures. It is also entirely suitable for use with DWARF2 and DWARF4. The setting of the 'frame cfa column number' defaults to `DW_FRAME_CFA_COL3` and it can be set at runtime with `dwarf_set_frame_cfa_value()`.

Mixing use of the DWARF2 interface set with use of the new DWARF3 interface set on a single open `Dwarf_Debug` instance is a mistake. Do not do it.

We will pretend, from here on unless otherwise specified, that `DW_FRAME_CFA_COL3`, `DW_FRAME_UNDEFINED_VAL`, and `DW_FRAME_SAME_VAL` are the synthetic column numbers. These columns may be user-chosen by calls of `dwarf_set_frame_cfa_value()`, `dwarf_set_frame_undefined_value()`, and `dwarf_set_frame_same_value()` respectively.

Each cell in the table contains one of the following:

1. A register + offset(a)(b)
2. A register(c)(d)

3. A marker (DW\_FRAME\_UNDEFINED\_VAL) meaning *register value undefined*

4. A marker (DW\_FRAME\_SAME\_VAL) meaning *register value same as in caller*

(a old DWARF2 interface) When the column is DW\_FRAME\_CFA\_COL: the register number is a real hardware register, not a reference to DW\_FRAME\_CFA\_COL, not DW\_FRAME\_UNDEFINED\_VAL, and not DW\_FRAME\_SAME\_VAL. The CFA rule value should be the stack pointer plus offset 0 when no other value makes sense. A value of DW\_FRAME\_SAME\_VAL would be semi-logical, but since the CFA is not a real register, not really correct. A value of DW\_FRAME\_UNDEFINED\_VAL would imply the CFA is undefined -- this seems to be a useless notion, as the CFA is a means to finding real registers, so those real registers should be marked DW\_FRAME\_UNDEFINED\_VAL, and the CFA column content (whatever register it specifies) becomes unreferenced by anything.

(a new April 2006 DWARF2/3 interface): The CFA is separately accessible and not part of the table. The 'rule number' for the CFA is a number outside the table. So the CFA is a marker, not a register number. See DW\_FRAME\_CFA\_COL3 in libdwarf.h and dwarf\_get\_fde\_info\_for\_cfa\_reg3() and dwarf\_set\_frame\_rule\_cfa\_value().

(b) When the column is not DW\_FRAME\_CFA\_COL3, the 'register' will and must be DW\_FRAME\_CFA\_COL3(COL), implying that to get the final location for the column one must add the offset here plus the DW\_FRAME\_CFA\_COL3 rule value.

(c) When the column is DW\_FRAME\_CFA\_COL3, then the 'register' number is (must be) a real hardware register. (This paragraph does not apply to the April 2006 new interface). If it were DW\_FRAME\_UNDEFINED\_VAL or DW\_FRAME\_SAME\_VAL it would be a marker, not a register number.

(d) When the column is not DW\_FRAME\_CFA\_COL3, the register may be a hardware register. It will not be DW\_FRAME\_CFA\_COL3.

There is no 'column' for DW\_FRAME\_UNDEFINED\_VAL or DW\_FRAME\_SAME\_VAL. Nor for DW\_FRAME\_CFA\_COL3.

Figure 8 is machine dependent and represents MIPS CPU register assignments. The DW\_FRAME\_CFA\_COL define in dwarf.h is historical and really belongs in libdwarf.h, not dwarf.h.



NAME	value	PURPOSE
DW_FRAME_CFA_COL	0	column used for CFA
DW_FRAME_REG1	1	integer register 1
DW_FRAME_REG2	2	integer register 2
---		obvious names and values here
DW_FRAME_REG30	30	integer register 30
DW_FRAME_REG31	31	integer register 31
DW_FRAME_FREG0	32	floating point register 0
DW_FRAME_FREG1	33	floating point register 1
---		obvious names and values here
DW_FRAME_FREG30	62	floating point register 30
DW_FRAME_FREG31	63	floating point register 31
DW_FRAME_RA_COL	64	column recording ra
DW_FRAME_UNDEFINED_VAL	1034	register val undefined
DW_FRAME_SAME_VAL	1035	register same as in caller

**Figure 35.** Frame Information Rule Assignments MIPS

The following table shows SGI/MIPS specific special cell values: these values mean that the cell has the value *undefined* or *same value* respectively, rather than containing a *register* or *register+offset*. It assumes DW\_FRAME\_CFA\_COL is a table rule, which is not readily accomplished or even sensible for some architectures.

NAME	value	PURPOSE
DW_FRAME_UNDEFINED_VAL	1034	means undefined value. Not a column or register value
DW_FRAME_SAME_VAL	1035	means 'same value' as caller had. Not a column or register value
DW_FRAME_CFA_COL	0	means register zero is usurped by the CFA column.

**Figure 36.** Frame Information Special Values any architecture

The following table shows more general special cell values. These values mean that the cell register-number refers to the *cfa-register* or *undefined-value* or *same-value* respectively, rather than referring to a *register in the table*. The generality arises from making DW\_FRAME\_CFA\_COL3 be outside the set of registers and making the cfa rule accessible from outside the rule-table.

NAME	value	PURPOSE
DW_FRAME_UNDEFINED_VAL	1034	means undefined value. Not a column or register value
DW_FRAME_SAME_VAL	1035	means 'same value' as caller had. Not a column or register value
DW_FRAME_CFA_COL3	1436	means 'cfa register' is referred to, not a real register, not a column, but the cfa (the cfa does have a value, but in the DWARF3 libdwarf interface it does not have a 'real register number').

### 6.20.1 dwarf\_get\_frame\_section\_name()

```
int dwarf_get_frame_section_name(Dwarf_Debug dbg,
    const char ** sec_name,
    Dwarf_Error *error)
```

`dwarf_get_string_section_name()` lets consumers access the object string section name. This is useful for applications wanting to print the name, but of course the object section name is not really a part of the DWARF information. Most applications will probably not call this function. It can be called at any time after the `Dwarf_Debug` initialization is done. See also `dwarf_get_frame_section_name_ghnu()`.

The function `dwarf_get_frame_section_name()` operates on the the `.debug_frame` section.

If the function succeeds, `*sec_name` is set to a pointer to a string with the object section name and the function returns `DW_DLV_OK`. Do not free the string whose pointer is returned. For non-Elf objects it is possible the string pointer returned will be `NULL` or will point to an empty string. It is up to the calling application to recognize this possibility and deal with it appropriately.

If the section does not exist the function returns `DW_DLV_NO_ENTRY`.

If there is an internal error detected the function returns `DW_DLV_ERROR` and sets the `*error` pointer.

### 6.20.2 dwarf\_get\_frame\_section\_name\_ghnu()

```
int dwarf_get_frame_section_name_ghnu(Dwarf_Debug dbg
    const char ** sec_name,
    Dwarf_Error *error)
```

`dwarf_get_frame_section_name_ghnu()` lets consumers access the object

string section name. This is useful for applications wanting to print the name, but of course the object section name is not really a part of the DWARF information. Most applications will probably not call this function. It can be called at any time after the Dwarf\_Debug initialization is done. See also dwarf\_get\_frame\_section\_name().

The function dwarf\_get\_frame\_section\_name\_eh\_gnu() operates on the the .eh\_frame section.

If the function succeeds, \*sec\_name is set to a pointer to a string with the object section name and the function returns DW\_DLV\_OK. Do not free the string whose pointer is returned. For non-Elf objects it is possible the string pointer returned will be NULL or will point to an empty string. It is up to the calling application to recognize this possibility and deal with it appropriately.

If the section does not exist the function returns DW\_DLV\_NO\_ENTRY.

If there is an internal error detected the function returns DW\_DLV\_ERROR and sets the \*error pointer.

### 6.20.3 dwarf\_get\_fde\_list()

```
int dwarf_get_fde_list(  
    Dwarf_Debug dbg,  
    Dwarf_Cie **cie_data,  
    Dwarf_Signed *cie_element_count,  
    Dwarf_Fde **fde_data,  
    Dwarf_Signed *fde_element_count,  
    Dwarf_Error *error);
```

dwarf\_get\_fde\_list() stores a pointer to a list of Dwarf\_Cie descriptors in \*cie\_data, and the count of the number of descriptors in \*cie\_element\_count. There is a descriptor for each CIE in the .debug\_frame section. Similarly, it stores a pointer to a list of Dwarf\_Fde descriptors in \*fde\_data, and the count of the number of descriptors in \*fde\_element\_count. There is one descriptor per FDE in the .debug\_frame section. dwarf\_get\_fde\_list() returns DW\_DLV\_ERROR on error. It returns DW\_DLV\_NO\_ENTRY if it cannot find frame entries. It returns DW\_DLV\_OK on a successful return.

On successful return, structures pointed to by a descriptor should be freed using dwarf\_fde\_cie\_list\_dealloc(). This dealloc approach is new as of July 15, 2005.

**Figure 37.** Exampleq dwarf\_get\_fde\_list()

```
void exampleq(Dwarf_Debug dbg)
{
    Dwarf_Cie *cie_data = 0;
    Dwarf_Signed cie_count = 0;
    Dwarf_Fde *fde_data = 0;
    Dwarf_Signed fde_count = 0;
    Dwarf_Error error = 0;
    int fres = 0;

    fres = dwarf_get_fde_list(dbg, &cie_data, &cie_count,
                             &fde_data, &fde_count, &error);
    if (fres == DW_DLV_OK) {
        dwarf_fde_cie_list_dealloc(dbg, cie_data, cie_count,
                                    fde_data, fde_count);
    }
}
```

The following code is deprecated as of July 15, 2005 as it does not free all relevant memory. This approach still works as well as it ever did.

**Figure 38.** Exampleqb dwarf\_get\_fde\_list() obsolete

```
/* OBSOLETE EXAMPLE */
void exampleqb(Dwarf_Debug dbg)
{
    Dwarf_Cie *cie_data = 0;
    Dwarf_Signed cie_count = 0;
    Dwarf_Fde *fde_data = 0;
    Dwarf_Signed fde_count = 0;
    Dwarf_Error error = 0;
    Dwarf_Signed i = 0;
    int fres = 0;

    fres = dwarf_get_fde_list(dbg, &cie_data, &cie_count,
                             &fde_data, &fde_count, &error);
    if (fres == DW_DLV_OK) {
        for (i = 0; i < cie_count; ++i) {
            /* use cie[i] */
            dwarf_dealloc(dbg, cie_data[i], DW_DLA_CIE);
        }
        for (i = 0; i < fde_count; ++i) {
            /* use fde[i] */
            dwarf_dealloc(dbg, fde_data[i], DW_DLA_FDE);
        }
        dwarf_dealloc(dbg, cie_data, DW_DLA_LIST);
        dwarf_dealloc(dbg, fde_data, DW_DLA_LIST);
    }
}
```

#### 6.20.4 dwarf\_get\_fde\_list\_eh()

```
int dwarf_get_fde_list_eh(
    Dwarf_Debug dbg,
    Dwarf_Cie **cie_data,
    Dwarf_Signed *cie_element_count,
    Dwarf_Fde **fde_data,
    Dwarf_Signed *fde_element_count,
    Dwarf_Error *error);
```

dwarf\_get\_fde\_list\_eh() is identical to dwarf\_get\_fde\_list() except that dwarf\_get\_fde\_list\_eh() reads the GNU gcc section named .eh\_frame (C++ exception handling information).

dwarf\_get\_fde\_list\_eh() stores a pointer to a list of Dwarf\_Cie descriptors in \*cie\_data, and the count of the number of descriptors in \*cie\_element\_count. There is a descriptor for each CIE in the .debug\_frame section. Similarly, it stores a

pointer to a list of Dwarf\_Fde descriptors in \*fde\_data, and the count of the number of descriptors in \*fde\_element\_count. There is one descriptor per FDE in the .debug\_frame section. dwarf\_get\_fde\_list() returns DW\_DLV\_ERROR on error. It returns DW\_DLV\_NO\_ENTRY if it cannot find exception handling entries. It returns DW\_DLV\_OK on a successful return.

On successful return, structures pointed to by a descriptor should be freed using dwarf\_fde\_cie\_list\_dealloc(). This dealloc approach is new as of July 15, 2005.

**Figure 39.** Exemplar dwarf\_get\_fde\_list\_eh()

```
void exemplar(Dwarf_Debug dbg,Dwarf_Addr mypcval)
{
    /*  Given a pc value
        for a function find the FDE and CIE data for
        the function.
        Example shows basic access to FDE/CIE plus
        one way to access details given a PC value.
        dwarf_get_fde_n() allows accessing all FDE/CIE
        data so one could build up an application-specific
        table of information if that is more useful.  */
    Dwarf_Signed count = 0;
    Dwarf_Cie *cie_data = 0;
    Dwarf_Signed cie_count = 0;
    Dwarf_Fde *fde_data = 0;
    Dwarf_Signed fde_count = 0;
    Dwarf_Error error = 0;
    int fres = 0;

    fres = dwarf_get_fde_list_eh(dbg,&cie_data,&cie_count,
        &fde_data,&fde_count,&error);
    if (fres == DW_DLV_OK) {
        Dwarf_Fde myfde = 0;
        Dwarf_Addr low_pc = 0;
        Dwarf_Addr high_pc = 0;
        fres = dwarf_get_fde_at_pc(fde_data,mypcval,
            &myfde,&low_pc,&high_pc,
            &error);
        if (fres == DW_DLV_OK) {
            Dwarf_Cie mycie = 0;
            fres = dwarf_get_cie_of_fde(myfde,&mycie,&error);
            if (fres == DW_DLV_OK) {
                /*  Now we can access a range of information
                    about the fde and cie applicable.  */
            }
        }
        dwarf_fde_cie_list_dealloc(dbg, cie_data, cie_count,
            fde_data,fde_count);
    }
    /* ERROR or NO ENTRY. Do something */
}
```

### 6.20.5 dwarf\_get\_cie\_of\_fde()

```
int dwarf_get_cie_of_fde(Dwarf_Fde fde,  
    Dwarf_Cie *cie_returned,  
    Dwarf_Error *error);
```

`dwarf_get_cie_of_fde()` stores a `Dwarf_Cie` into the `Dwarf_Cie` that `cie_returned` points at.

If one has called `dwarf_get_fde_list()` must avoid `dwarf_dealloc`-ing the FDEs and the CIEs for those FDEs individually (see its documentation here). Failing to observe this restriction will cause the FDE(s) not `dealloc`'d to become invalid: an FDE contains (hidden in it) a CIE pointer which will be be invalid (stale, pointing to freed memory) if the CIE is `dealloc`'d. The invalid CIE pointer internal to the FDE cannot be detected as invalid by `libdwarf`. If one later passes an FDE with a stale internal CIE pointer to one of the routines taking an FDE as input the result will be failure of the call (returning `DW_DLV_ERROR`) at best and it is possible a coredump or worse will happen (eventually).

`dwarf_get_cie_of_fde()` returns `DW_DLV_OK` if it is successful (it will be unless `fde` is the `NULL` pointer). It returns `DW_DLV_ERROR` if the `fde` is invalid (`NULL`).

Each `Dwarf_Fde` descriptor describes information about the frame for a particular subroutine or function.

`int dwarf_get_fde_for_die` is SGI/MIPS specific.

### 6.20.6 `dwarf_get_fde_for_die()`

```
int dwarf_get_fde_for_die(  
    Dwarf_Debug dbg,  
    Dwarf_Die die,  
    Dwarf_Fde * return_fde,  
    Dwarf_Error *error)
```

When it succeeds, `dwarf_get_fde_for_die()` returns `DW_DLV_OK` and sets `*return_fde` to a `Dwarf_Fde` descriptor representing frame information for the given `die`. It looks for the `DW_AT_MIPS_fde` attribute in the given `die`. If it finds it, it uses the value of the attribute as the offset in the `.debug_frame` section where the FDE begins. If there is no `DW_AT_MIPS_fde` it returns `DW_DLV_NO_ENTRY`. If there is an error it returns `DW_DLV_ERROR`.

### 6.20.7 `dwarf_get_fde_range()`



```
int dwarf_get_fde_range(  
    Dwarf_Fde fde,  
    Dwarf_Addr *low_pc,  
    Dwarf_Unsigned *func_length,  
    Dwarf_Ptr *fde_bytes,  
    Dwarf_Unsigned *fde_byte_length,  
    Dwarf_Off *cie_offset,  
    Dwarf_Signed *cie_index,  
    Dwarf_Off *fde_offset,  
    Dwarf_Error *error);
```

On success, `dwarf_get_fde_range()` returns `DW_DLV_OK`.

The location pointed to by `low_pc` is set to the low pc value for this function.

The location pointed to by `func_length` is set to the length of the function in bytes. This is essentially the length of the text section for the function.

The location pointed to by `fde_bytes` is set to the address where the FDE begins in the `.debug_frame` section.

The location pointed to by `fde_byte_length` is set to the length in bytes of the portion of `.debug_frame` for this FDE. This is the same as the value returned by `dwarf_get_fde_range`.

The location pointed to by `cie_offset` is set to the offset in the `.debug_frame` section of the CIE used by this FDE.

The location pointed to by `cie_index` is set to the index of the CIE used by this FDE. The index is the index of the CIE in the list pointed to by `cie_data` as set by the function `dwarf_get_fde_list()`. However, if the function `dwarf_get_fde_for_die()` was used to obtain the given `fde`, this index may not be correct.

The location pointed to by `fde_offset` is set to the offset of the start of this FDE in the `.debug_frame` section.

`dwarf_get_fde_range()` returns `DW_DLV_ERROR` on error.

#### **6.20.8 dwarf\_get\_cie\_info()**

```
int dwarf_get_cie_info(  
    Dwarf_Cie      cie,  
    Dwarf_Unsigned *bytes_in_cie,  
    Dwarf_Small    *version,  
    char           **augmenter,  
    Dwarf_Unsigned *code_alignment_factor,  
    Dwarf_Signed   *data_alignment_factor,  
    Dwarf_Half     *return_address_register_rule,  
    Dwarf_Ptr      *initial_instructions,  
    Dwarf_Unsigned *initial_instructions_length,  
    Dwarf_Error     *error);
```

`dwarf_get_cie_info()` is primarily for Internal-level Interface consumers. If successful, it returns `DW_DLV_OK` and sets `*bytes_in_cie` to the number of bytes in the portion of the frames section for the CIE represented by the given `Dwarf_Cie` descriptor, `cie`. The other fields are directly taken from the `cie` and returned, via the pointers to the caller. It returns `DW_DLV_ERROR` on error.

#### 6.20.9 dwarf\_get\_cie\_index()

```
int dwarf_get_cie_index(  
    Dwarf_Cie cie,  
    Dwarf_Signed *cie_index,  
    Dwarf_Error *error);
```

On success, `dwarf_get_cie_index()` returns `DW_DLV_OK`. On error this function returns `DW_DLV_ERROR`.

The location pointed to by `cie_index` is set to the index of the CIE of this FDE. The index is the index of the CIE in the list pointed to by `cie_data` as set by the function `dwarf_get_fde_list()`.

So one must have used `dwarf_get_fde_list()` or `dwarf_get_fde_list_eh()` to get a `cie` list before this is meaningful.

This function is occasionally useful, but is little used.

#### 6.20.10 dwarf\_get\_fde\_instr\_bytes()

```
int dwarf_get_fde_instr_bytes(  
    Dwarf_Fde fde,  
    Dwarf_Ptr *outinstrs,  
    Dwarf_Unsigned *outlen,  
    Dwarf_Error *error);
```

`dwarf_get_fde_instr_bytes()` returns `DW_DLV_OK` and sets `*outinstrs` to a pointer to a set of bytes which are the actual frame instructions for this fde. It also sets `*outlen` to the length, in bytes, of the frame instructions. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`. The intent is to allow low-level consumers like a dwarf-dumper to print the bytes in some fashion. The memory pointed to by `outinstrs` must not be changed and there is nothing to free.

### 6.20.11 `dwarf_get_fde_info_for_reg()`

This interface is suitable for DWARF2 but is not sufficient for DWARF3. See `int dwarf_get_fde_info_for_reg3`.

```
int dwarf_get_fde_info_for_reg(  
    Dwarf_Fde fde,  
    Dwarf_Half table_column,  
    Dwarf_Addr pc_requested,  
    Dwarf_Signed *offset_relevant,  
    Dwarf_Signed *register_num,  
    Dwarf_Signed *offset,  
    Dwarf_Addr *row_pc,  
    Dwarf_Error *error);
```

`dwarf_get_fde_info_for_reg()` returns `DW_DLV_OK` and sets `*offset_relevant` to non-zero if the offset is relevant for the row specified by `pc_requested` and column specified by `table_column`, for the FDE specified by `fde`. The intent is to return the rule for the given pc value and register. The location pointed to by `register_num` is set to the register value for the rule. The location pointed to by `offset` is set to the offset value for the rule. If offset is not relevant for this rule, `*offset_relevant` is set to zero. Since more than one pc value will have rows with identical entries, the user may want to know the earliest pc value after which the rules for all the columns remained unchanged. Recall that in the virtual table that the frame information represents there may be one or more table rows with identical data (each such table row at a different pc value). Given a `pc_requested` which refers to a pc in such a group of identical rows, the location pointed to by `row_pc` is set to the lowest pc value within the group of identical rows. The value put in `*register_num` any of the `DW_FRAME_*` table columns values specified in `libdwarf.h` or `dwarf.h`.

`dwarf_get_fde_info_for_reg` returns `DW_DLV_ERROR` if there is an error.

It is usable with either `dwarf_get_fde_n()` or `dwarf_get_fde_at_pc()`.

`dwarf_get_fde_info_for_reg()` is tailored to MIPS, please use `dwarf_get_fde_info_for_reg3()` instead for all architectures.

### 6.20.12 `dwarf_get_fde_info_for_all_regs()`

```
int dwarf_get_fde_info_for_all_regs(  
    Dwarf_Fde fde,  
    Dwarf_Addr pc_requested,  
    Dwarf_Regtable *reg_table,  
    Dwarf_Addr *row_pc,  
    Dwarf_Error *error);
```

`dwarf_get_fde_info_for_all_regs()` returns `DW_DLV_OK` and sets `*reg_table` for the row specified by `pc_requested` for the FDE specified by `fde`.

The intent is to return the rules for decoding all the registers, given a pc value. `reg_table` is an array of rules, one for each register specified in `dwarf.h`. The rule for each register contains three items - `dw_regnum` which denotes the register value for that rule, `dw_offset` which denotes the offset value for that rule and `dw_offset_relevant` which is set to zero if offset is not relevant for that rule. See `dwarf_get_fde_info_for_reg()` for a description of `row_pc`.

`dwarf_get_fde_info_for_all_regs` returns `DW_DLV_ERROR` if there is an error.

`int dwarf_get_fde_info_for_all_regs` is tailored to SGI/MIPS, please use `dwarf_get_fde_info_for_all_regs3()` instead for all architectures.

### 6.20.13 `dwarf_fde_section_offset()`

```
int dwarf_fde_section_offset(  
    Dwarf_Debug /*dbg*/,  
    Dwarf_Fde /*in_fde*/,  
    Dwarf_Off * /*fde_off*/,  
    Dwarf_Off * /*cie_off*/,  
    Dwarf_Error *error);
```

On success `dwarf_fde_section_offset()` returns the `.dwarf_line` section offset of the fde passed in and also the offset of its CIE.

It returns `DW_DLV_ERROR` if there is an error.

It returns `DW_DLV_ERROR` if there is an error.

when such want to print the offsets of CIEs and FDEs.

### 6.20.14 dwarf\_cie\_section\_offset()

```
int dwarf_cie_section_offset (
    Dwarf_Debug /*dbg*/,
    Dwarf_Cie /*in_cie*/,
    Dwarf_Off * /*cie_off*/,
    Dwarf_Error * /*err*/);
Dwarf_Error *error);
```

On success `dwarf_cie_section_offset()` returns the `.dwarf_line` section offset of the cie passed in.

It returns `DW_DLV_ERROR` if there is an error.

when such want to print the offsets of CIEs.

### 6.20.15 dwarf\_set\_frame\_rule\_table\_size()

This allows consumers to set the size of the (internal to libdwarf) rule table when using the 'reg3' interfaces (these interfaces are strongly preferred over the older 'reg' interfaces). It should be at least as large as the number of real registers in the ABI which is to be read in for the `dwarf_get_fde_info_for_reg3()` or `dwarf_get_fde_info_for_all_regs3()` functions to work properly.

The frame rule table size must be less than the marker values `DW_FRAME_UNDEFINED_VAL`, `DW_FRAME_SAME_VAL`, `DW_FRAME_CFA_COL3` (`dwarf_set_frame_rule_undefined_value()`, `dwarf_set_frame_same_value()`, `dwarf_set_frame_cfa_value()` effectively set these markers so the frame rule table size can actually be any value regardless of the macro values in `libdwarf.h` as long as the table size does not overlap these markers).

```
Dwarf_Half
dwarf_set_frame_rule_table_size(Dwarf_Debug dbg,
    Dwarf_Half value);
```

`dwarf_set_frame_rule_table_size()` sets the value value as the size of libdwarf-internal rules tables of dbg.

The function returns the previous value of the rules table size setting (taken from the `dbg` structure).

### 6.20.16 dwarf\_set\_frame\_rule\_initial\_value()

This allows consumers to set the initial value for rows in the frame tables. By default it is taken from `libdwarf.h` and is `DW_FRAME_REG_INITIAL_VALUE` (which itself is

either DW\_FRAME\_SAME\_VAL or DW\_FRAME\_UNDEFINED\_VAL). The MIPS/IRIX default is DW\_FRAME\_SAME\_VAL. Consumer code should set this appropriately and for many architectures (but probably not MIPS) DW\_FRAME\_UNDEFINED\_VAL is an appropriate setting. Note: an earlier spelling of dwarf\_set\_frame\_rule\_initial\_value() is still supported as an interface, but please change to use the new correctly spelled name.

```
Dwarf_Half  
dwarf_set_frame_rule_initial_value(Dwarf_Debug dbg,  
    Dwarf_Half value);
```

dwarf\_set\_frame\_rule\_initial\_value() sets the value value as the initial value for this dbg when initializing rules tables.

The function returns the previous value of initial value (taken from the dbg structure).

#### 6.20.17 dwarf\_set\_frame\_cfa\_value()

This allows consumers to set the number of the CFA register for rows in the frame tables. By default it is taken from libdwarf.h and is DW\_FRAME\_CFA\_COL. Consumer code should set this appropriately and for nearly all architectures DW\_FRAME\_CFA\_COL3 is an appropriate setting.

```
Dwarf_Half  
dwarf_set_frame_rule_cfa_value(Dwarf_Debug dbg,  
    Dwarf_Half value);
```

dwarf\_set\_frame\_rule\_cfa\_value() sets the value value as the number of the cfa 'register rule' for this dbg when initializing rules tables.

The function returns the previous value of the pseudo-register (taken from the dbg structure).

#### 6.20.18 dwarf\_set\_frame\_same\_value()

This allows consumers to set the number of the pseudo-register when DW\_CFA\_same\_value is the operation. By default it is taken from libdwarf.h and is DW\_FRAME\_SAME\_VAL. Consumer code should set this appropriately, though for many architectures DW\_FRAME\_SAME\_VAL is an appropriate setting.

```
Dwarf_Half  
dwarf_set_frame_rule_same_value(Dwarf_Debug dbg,  
    Dwarf_Half value);
```

dwarf\_set\_frame\_rule\_same\_value() sets the value value as the number of the register that is the pseudo-register set by the DW\_CFA\_same\_value frame operation.

The function returns the previous value of the pseudo-register (taken from the dbg

structure).

### 6.20.19 dwarf\_set\_frame\_undefined\_value()

This allows consumers to set the number of the pseudo-register when DW\_CFA\_undefined\_value is the operation. By default it is taken from libdwarf.h and is DW\_FRAME\_UNDEFINED\_VAL. Consumer code should set this appropriately, though for many architectures DW\_FRAME\_UNDEFINED\_VAL is an appropriate setting.

```
Dwarf_Half  
dwarf_set_frame_rule_undefined_value(Dwarf_Debug dbg,  
                                     Dwarf_Half value);
```

dwarf\_set\_frame\_rule\_undefined\_value() sets the value value as the number of the register that is the pseudo-register set by the DW\_CFA\_undefined\_value frame operation.

The function returns the previous value of the pseudo-register (taken from the dbg structure).

### 6.20.20 dwarf\_set\_default\_address\_size()

This allows consumers to set a default address size. When one has an object where the default address\_size does not match the frame address size where there is no debug\_info available to get a frame-specific address-size, this function is useful. For example, if an Elf64 object has a .debug\_frame whose real address\_size is 4 (32 bits). This a very rare situation.

```
Dwarf_Small  
dwarf_set_default_address_size(Dwarf_Debug dbg,  
                               Dwarf_Small value);
```

dwarf\_set\_default\_address\_size() sets the value value as the default address size for this activation of the reader, but only if value is greater than zero (otherwise the default address size is not changed).

The function returns the previous value of the default address size (taken from the dbg structure).

### 6.20.21 dwarf\_get\_fde\_info\_for\_reg3()

This interface is suitable for DWARF2 and later. It returns the values for a particular real register (Not for the CFA virtual register, see dwarf\_get\_fde\_info\_for\_cfa\_reg3() below). If the application is going to retrieve the value for more than a few table\_column

values at this `pc_requested` (by calling this function multiple times) it is much more efficient to call `dwarf_get_fde_info_for_all_regs3()` (in spite of the additional setup that requires of the caller).

```
int dwarf_get_fde_info_for_reg3(  
    Dwarf_Fde fde,  
    Dwarf_Half table_column,  
    Dwarf_Addr pc_requested,  
    Dwarf_Small *value_type,  
    Dwarf_Signed *offset_relevant,  
    Dwarf_Signed *register_num,  
    Dwarf_Signed *offset_or_block_len,  
    Dwarf_Ptr *block_ptr,  
    Dwarf_Addr *row_pc,  
    Dwarf_Error *error);
```

See also the nearly identical function `dwarf_get_fde_info_for_reg3_b()`.

`dwarf_get_fde_info_for_reg3()` returns `DW_DLV_OK` on success. It sets `*value_type` to one of `DW_EXPR_OFFSET` (0), `DW_EXPR_VAL_OFFSET`(1), `DW_EXPR_EXPRESSION`(2) or `DW_EXPR_VAL_EXPRESSION`(3). On call, `table_column` must be set to the register number of a real register. Not the `cfa` 'register' or `DW_FRAME_SAME_VALUE` or `DW_FRAME_UNDEFINED_VALUE`.

if `*value_type` has the value `DW_EXPR_OFFSET` (0) then:

It sets `*offset_relevant` to non-zero if the offset is relevant for the row specified by `pc_requested` and column specified by `table_column` or, for the FDE specified by `fde`. In this case the `*register_num` will be set to `DW_FRAME_CFA_COL3` (. This is an offset(N) rule as specified in the DWARF3/2 documents.

Adding the value of `*offset_or_block_len` to the value of the CFA register gives the address of a location holding the previous value of register `table_column`.

If offset is not relevant for this rule, `*offset_relevant` is set to zero. `*register_num` will be set to the number of the real register holding the value of the `table_column` register. This is the register(R) rule as specified in DWARF3/2 documents.

The intent is to return the rule for the given `pc` value and register. The location pointed to by `register_num` is set to the register value for the rule. The location pointed to by `offset` is set to the offset value for the rule. Since more than one `pc` value will have rows with identical entries, the user may want to know the earliest `pc` value after which the rules for all the columns remained unchanged. Recall that in the virtual table that the frame information



represents there may be one or more table rows with identical data (each such table row at a different pc value). Given a `pc_requested` which refers to a pc in such a group of identical rows, the location pointed to by `row_pc` is set to the lowest pc value within the group of identical rows.

If `*value_type` has the value `DW_EXPR_VAL_OFFSET` (1) then:

This will be a `val_offset(N)` rule as specified in the DWARF3/2 documents so `*offset_relevant` will be non zero. The calculation is identical to the `DW_EXPR_OFFSET` (0) calculation with `*offset_relevant` non-zero, but the value resulting is the actual `table_column` value (rather than the address where the value may be found).

If `*value_type` has the value `DW_EXPR_EXPRESSION` (1) then:

`*offset_or_block_len` is set to the length in bytes of a block of memory with a DWARF expression in the block. `*block_ptr` is set to point at the block of memory. The consumer code should evaluate the block as a DWARF-expression. The result is the address where the previous value of the register may be found. This is a DWARF3/2 `expression(E)` rule.

If `*value_type` has the value `DW_EXPR_VAL_EXPRESSION` (1) then:

The calculation is exactly as for `DW_EXPR_EXPRESSION` (1) but the result of the DWARF-expression evaluation is the value of the `table_column` (not the address of the value). This is a DWARF3/2 `val_expression(E)` rule.

`dwarf_get_fde_info_for_reg` returns `DW_DLV_ERROR` if there is an error and if there is an error only the `error` pointer is set, none of the other output arguments are touched.

It is usable with either `dwarf_get_fde_n()` or `dwarf_get_fde_at_pc()`.

### 6.20.22 `dwarf_get_fde_info_for_reg3_b()`

This interface is suitable for DWARF2 and later. It returns the values for a particular real register (Not for the CFA virtual register, see `dwarf_get_fde_info_for_cfa_reg3_b()` below). If the application is going to retrieve the value for more than a few `table_column` values at this `pc_requested` (by calling this function multiple times) it is much more efficient to call `dwarf_get_fde_info_for_all_regs3()` (in spite of the additional setup that requires of the caller).

```
int dwarf_get_fde_info_for_reg3_b(  
    Dwarf_Fde fde,  
    Dwarf_Half table_column,  
    Dwarf_Addr pc_requested,  
    Dwarf_Small *value_type,  
    Dwarf_Signed *offset_relevant,  
    Dwarf_Signed *register_num,  
    Dwarf_Signed *offset_or_block_len,  
    Dwarf_Ptr *block_ptr,  
    Dwarf_Addr *row_pc,  
    Dwarf_Bool *has_more_rows,  
    Dwarf_Addr *subsequent_pc,  
    Dwarf_Error *error);
```

This is identical to `dwarf_get_fde_info_for_reg3()` except for the new arguments `has_more_rows` and `subsequent_pc` which allow the caller to know if there are more rows in the frame table and what the next pc value in the frame table for this fde is. The two new arguments may be passed in as NULL if their values are not needed by the caller.

### 6.20.23 dwarf\_get\_fde\_info\_for\_cfa\_reg3()

```
int dwarf_get_fde_info_for_cfa_reg3(Dwarf_Fde fde,  
    Dwarf_Addr pc_requested,  
    Dwarf_Small *value_type,  
    Dwarf_Signed* offset_relevant,  
    Dwarf_Signed* register_num,  
    Dwarf_Signed* offset_or_block_len,  
    Dwarf_Ptr *block_ptr ,  
    Dwarf_Addr *row_pc_out,  
    Dwarf_Error *error)
```

This is identical to `dwarf_get_fde_info_for_reg3()` except the returned values are for the CFA rule. So register number `*register_num` will be set to a real register, not one of the pseudo registers (which are usually `DW_FRAME_CFA_COL3`, `DW_FRAME_SAME_VALUE`, or `DW_FRAME_UNDEFINED_VALUE`).

Applications like `dwarfdump` which access the register rules for every pc value in a function may find the following function a slight performance improvement if the new arguments are used appropriately. See `dwarfdump` for an example of use.

### 6.20.24 dwarf\_get\_fde\_info\_for\_cfa\_reg3\_b()

```
int dwarf_get_fde_info_for_cfa_reg3_b(Dwarf_Fde fde,
    Dwarf_Addr      pc_requested,
    Dwarf_Small *   value_type,
    Dwarf_Signed*   offset_relevant,
    Dwarf_Signed*   register_num,
    Dwarf_Signed*   offset_or_block_len,
    Dwarf_Ptr *     block_ptr ,
    Dwarf_Addr *    row_pc_out,
    Dwarf_Bool *    has_more_rows,
    Dwarf_Addr *    subsequent_pc,
    Dwarf_Error *   error)
```

This is identical to `dwarf_get_fde_info_for_cfa_reg3()` except for the new arguments `has_more_rows` and `subsequent_pc` which allow the caller to know if there are more rows in the frame table and what the next pc value is. The two new arguments may be passed in as NULL if their values are not needed by the caller.

For a tool just wanting the frame information for a single pc\_value this interface is no more useful or efficient than `dwarf_get_fde_info_for_cfa_reg3()`.

The essential difference is that when using `dwarf_get_fde_info_for_cfa_reg3()` for all pc values for a function the caller has no idea what is the next pc value that might have new frame data and iterating through pc values (calling `dwarf_get_fde_info_for_cfa_reg3()` on each) is a waste of cpu cycles. With `dwarf_get_fde_info_for_cfa_reg3_b()` the `has_more_rows` and `subsequent_pc` arguments let the caller know whether there are further rows and if so at what pc value.

If `has_more_rows` is non-null then 1 is returned through the pointer if, for the `pc_requested` there is frame data for addresses after `pc_requested` in the frame. And if there are no more rows in the frame data then 0 is set through the `has_more_rows` pointer.

If `subsequent_pc` is non-null then the pc-value which has the next frame operator is returned through the pointer. If no more rows are present zero is returned through the pointer, but please use `has_more_rows` to determine if there are more rows.

### 6.20.25 dwarf\_get\_fde\_info\_for\_all\_regs3()

```
int dwarf_get_fde_info_for_all_regs3(
    Dwarf_Fde fde,
    Dwarf_Addr pc_requested,
    Dwarf_Regtable3 *reg_table,
    Dwarf_Addr *row_pc,
    Dwarf_Error *error)
```

`dwarf_get_fde_info_for_all_regs3()` returns `DW_DLV_OK` and sets `*reg_table` for the row specified by `pc_requested` for the FDE specified by `fde`. The intent is to return the rules for decoding all the registers, given a `pc` value. `reg_table` is an array of rules, the array size specified by the caller. plus a rule for the CFA. The rule for the cfa returned in `*reg_table` defines the CFA value at `pc_requested`. The rule for each register contains several values that enable the consumer to determine the previous value of the register (see the earlier documentation of `Dwarf_Regtable3`). `dwarf_get_fde_info_for_reg3()` and the `Dwarf_Regtable3` documentation above for a description of the values for each row.

`dwarf_get_fde_info_for_all_regs3` returns `DW_DLV_ERROR` if there is an error.

It is up to the caller to allocate space for `*reg_table` and initialize it properly.

#### 6.20.26 `dwarf_get_fde_n()`

```
int    dwarf_get_fde_n(  
        Dwarf_Fde *fde_data,  
        Dwarf_Unsigned fde_index,  
        Dwarf_Fde      *returned_fde  
        Dwarf_Error *error)
```

`dwarf_get_fde_n()` returns `DW_DLV_OK` and sets `returned_fde` to the `Dwarf_Fde` descriptor whose index is `fde_index` in the table of `Dwarf_Fde` descriptors pointed to by `fde_data`. The index starts with 0. The table pointed to by `fde_data` is required to contain at least one entry. If the table has no entries at all the error checks may refer to uninitialized memory. Returns `DW_DLV_NO_ENTRY` if the index does not exist in the table of `Dwarf_Fde` descriptors. Returns `DW_DLV_ERROR` if there is an error. This function cannot be used unless the block of `Dwarf_Fde` descriptors has been created by a call to `dwarf_get_fde_list()`.

#### 6.20.27 `dwarf_get_fde_at_pc()`

```
int    dwarf_get_fde_at_pc(  
        Dwarf_Fde *fde_data,  
        Dwarf_Addr pc_of_interest,  
        Dwarf_Fde *returned_fde,  
        Dwarf_Addr *lopc,  
        Dwarf_Addr *hipc,  
        Dwarf_Error *error)
```

`dwarf_get_fde_at_pc()` returns `DW_DLV_OK` and sets `returned_fde` to a `Dwarf_Fde` descriptor for a function which contains the `pc` value specified by `pc_of_interest`. In addition, it sets the locations pointed to by `lopc` and `hipc` to the low address and the high address covered by this FDE, respectively. The table pointed to by `fde_data` is required to contain at least one entry. If the table has no entries at all the error checks may refer to uninitialized memory. It returns `DW_DLV_ERROR` on error. It returns `DW_DLV_NO_ENTRY` if `pc_of_interest` is not in any of the FDEs represented by the block of `Dwarf_Fde` descriptors pointed to by `fde_data`. This function cannot be used unless the block of `Dwarf_Fde` descriptors has been created by a call to `dwarf_get_fde_list()`.

### 6.20.28 `dwarf_expand_frame_instructions()`

```
int dwarf_expand_frame_instructions(  
    Dwarf_Cie cie,  
    Dwarf_Ptr instruction,  
    Dwarf_Unsigned i_length,  
    Dwarf_Frame_Op **returned_op_list,  
    Dwarf_Signed * returned_op_count,  
    Dwarf_Error *error);
```

`dwarf_expand_frame_instructions()` is a High-level interface function which expands a frame instruction byte stream into an array of `Dwarf_Frame_Op` structures. To indicate success, it returns `DW_DLV_OK`. The address where the byte stream begins is specified by `instruction`, and the length of the byte stream is specified by `i_length`. The location pointed to by `returned_op_list` is set to point to a table of `returned_op_count` pointers to `Dwarf_Frame_Op` which contain the frame instructions in the byte stream. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`. After a successful return, the array of structures should be freed using `dwarf_dealloc()` with the allocation type `DW_DLA_FRAME_BLOCK` (when they are no longer of interest).

Not all CIEs have the same address-size, so it is crucial that a CIE pointer to the frame's CIE be passed in.

**Figure 40.** Examples `dwarf_expand_frame_instructions()`

```
void examples(Dwarf_Debug dbg,Dwarf_Cie cie,
              Dwarf_Ptr instruction,Dwarf_Unsigned len)
{
    Dwarf_Signed count = 0;
    Dwarf_Frame_Op *frameops = 0;
    Dwarf_Error error = 0;
    int res = 0;

    res = dwarf_expand_frame_instructions(cie,instruction,len,
                                         &frameops,&count, &error);
    if (res == DW_DLV_OK) {
        Dwarf_Signed i = 0;

        for (i = 0; i < count; ++i) {
            /* use frameops[i] */
        }
        dwarf_dealloc(dbg, frameops, DW_DLA_FRAME_BLOCK);
    }
}
```

#### **6.20.29 dwarf\_get\_fde\_exception\_info()**

```
int dwarf_get_fde_exception_info(
    Dwarf_Fde fde,
    Dwarf_Signed * offset_into_exception_tables,
    Dwarf_Error * error);
```

dwarf\_get\_fde\_exception\_info() is an IRIX specific function which returns an exception table signed offset through offset\_into\_exception\_tables. The function never returns DW\_DLV\_NO\_ENTRY. If DW\_DLV\_NO\_ENTRY is NULL the function returns DW\_DLV\_ERROR. For non-IRIX objects the offset returned will always be zero. For non-C++ objects the offset returned will always be zero. The meaning of the offset and the content of the tables is not defined in this document. The applicable CIE augmentation string (see above) determines whether the value returned has meaning.

### **6.21 Location Expression Evaluation**

An "interpreter" which evaluates a location expression is required in any debugger. There is no interface defined here at this time.

One problem with defining an interface is that operations are machine dependent: they depend on the interpretation of register numbers and the methods of getting values from the environment the expression is applied to.

It would be desirable to specify an interface.

## 6.21.1 Location List Internal-level Interface

### 6.21.1.1 dwarf\_get\_loclist\_entry()

```
int dwarf_get_loclist_entry(  
    Dwarf_Debug dbg,  
    Dwarf_Unsigned offset,  
    Dwarf_Addr *hipc_offset,  
    Dwarf_Addr *lopc_offset,  
    Dwarf_Ptr *data,  
    Dwarf_Unsigned *entry_len,  
    Dwarf_Unsigned *next_entry,  
    Dwarf_Error *error)
```

This function is ill suited to use with 21st century DWARF as there is just not enough data provided in the interface. Do not use this interface.

The function reads a location list entry starting at `offset` and returns through pointers (when successful) the high pc `hipc_offset`, low pc `lopc_offset`, a pointer to the location description data `data`, the length of the location description data `entry_len`, and the offset of the next location description entry `next_entry`.

This function will often work correctly (meaning with most objects compiled for DWARF3 or DWARF3) but will not work correctly (and can crash an application calling it) if either some location list applies to a compilation unit with an `address_size` different from the overall `address_size` of the object file being read or if the `.debug_loc` section being read has random padding bytes between loclists. Neither of these characteristics necessarily represents a bug in the compiler/linker toolset that produced the object file being read. The DWARF standard allows both characteristics.

`dwarf_dwarf_get_loclist_entry()` returns `DW_DLV_OK` if successful. `DW_DLV_NO_ENTRY` is returned when the offset passed in is beyond the end of the `.debug_loc` section (expected if you start at offset zero and proceed through all the entries). `DW_DLV_ERROR` is returned on error.

The `hipc_offset`, low pc `lopc_offset` are offsets from the beginning of the current procedure, not genuine pc values.

**Figure 41.** Examples `dwarf_get_loclist_entry()`

```
void examplet(Dwarf_Debug dbg,Dwarf_Unsigned offset)
{
    /* Looping through the dwarf_loc section finding loclists:
       an example. */
    int res;
    Dwarf_Unsigned next_entry = 0;
    Dwarf_Addr hipc_off = 0;
    Dwarf_Addr lowpc_off = 0;
    Dwarf_Ptr data = 0;
    Dwarf_Unsigned entry_len = 0;
    Dwarf_Error err = 0;

    for(;;) {
        res = dwarf_get_loclist_entry(dbg,offset,&hipc_off,
                                     &lowpc_off, &data, &entry_len,&next_entry,&err);
        if (res == DW_DLV_OK) {
            /* A valid entry. */
            offset = next_entry;
            continue;
        } else if (res ==DW_DLV_NO_ENTRY) {
            /* Done! */
            break;
        } else {
            /* Error! */
            break;
        }
    }
}
```

## 6.22 Abbreviations access

These are Internal-level Interface functions. Debuggers can ignore this.

### 6.22.1 dwarf\_get\_abbrev()

```
int dwarf_get_abbrev(
    Dwarf_Debug dbg,
    Dwarf_Unsigned offset,
    Dwarf_Abbrev *returned_abbrev,
    Dwarf_Unsigned *length,
    Dwarf_Unsigned *attr_count,
    Dwarf_Error *error)
```



The function `dwarf_get_abbrev()` returns `DW_DLV_OK` and sets `*returned_abbrev` to Dwarf\_Abbrev descriptor for an abbreviation at offset `*offset` in the abbreviations section (i.e `.debug_abbrev`) on success. The user is responsible for making sure that a valid abbreviation begins at `offset` in the abbreviations section. The location pointed to by `length` is set to the length in bytes of the abbreviation in the abbreviations section. The location pointed to by `attr_count` is set to the number of attributes in the abbreviation. An abbreviation entry with a length of 1 is the 0 byte of the last abbreviation entry of a compilation unit.

`dwarf_get_abbrev()` returns `DW_DLV_ERROR` on error. If the call succeeds, the storage pointed to by `*returned_abbrev` should be freed, using `dwarf_dealloc()` with the allocation type `DW_DLA_ABBREV` when no longer needed.

### 6.22.2 dwarf\_get\_abbrev\_tag()

```
int dwarf_get_abbrev_tag(  
    Dwarf_abbrev abbrev,  
    Dwarf_Half *return_tag,  
    Dwarf_Error *error);
```

If successful, `dwarf_get_abbrev_tag()` returns `DW_DLV_OK` and sets `*return_tag` to the *tag* of the given abbreviation. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

### 6.22.3 dwarf\_get\_abbrev\_code()

```
int dwarf_get_abbrev_code(  
    Dwarf_abbrev abbrev,  
    Dwarf_Unsigned *return_code,  
    Dwarf_Error *error);
```

If successful, `dwarf_get_abbrev_code()` returns `DW_DLV_OK` and sets `*return_code` to the abbreviation code of the given abbreviation. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

### 6.22.4 dwarf\_get\_abbrev\_children\_flag()

```
int dwarf_get_abbrev_children_flag(  
    Dwarf_Abbrev abbrev,  
    Dwarf_Signed *returned_flag,  
    Dwarf_Error *error)
```

The function `dwarf_get_abbrev_children_flag()` returns `DW_DLV_OK` and sets `returned_flag` to `DW_children_no` (if the given abbreviation indicates that a die with that abbreviation has no children) or `DW_children_yes` (if the given abbreviation indicates that a die with that abbreviation has a child). It returns `DW_DLV_ERROR` on error.

### 6.22.5 `dwarf_get_abbrev_entry_b()`

```
int dwarf_get_abbrev_entry_b(Dwarf_Abbrev abbrev,  
    Dwarf_Unsigned index,  
    Dwarf_Bool      filter_outliers,  
    Dwarf_Unsigned * returned_attr_num,  
    Dwarf_Unsigned * returned_form,  
    Dwarf_Signed   * returned_implicit_const,  
    Dwarf_Off      * offset,  
    Dwarf_Error    * error)
```

`dwarf_get_abbrev_entry_b()` is new in August 2019. It should be used in place of `dwarf_get_abbrev_entry()` as `dwarf_get_abbrev_entry()` cannot return the DWARF5 implicit const value and `dwarf_get_abbrev_entry()` can hide some instances of corrupt uleb abbreviation values.

While the `returned_attr_num` and `returned_form` are only correct if they each fit in a `Dwarf_Half` value, we return larger values in certain cases (see next paragraph).

If `filter_outliers` is passed in zero then erroneous `returned_attr_num` or `returned_form` are returned whether their values are sensible or not and `DW_DLV_OK` is the returned value. This is useful for `dwarfdump` as `dwarfdump` checks abbreviation values quite thoroughly and reports errors in detail (`dwarfdump -kb`).

If `filter_outliers` is passed in non-zero then `DW_DLV_OK` is returned only if `returned_attr_num` and `returned_form` are both legitimate values.

If successful, `dwarf_get_abbrev_entry_b()` returns `DW_DLV_OK` and sets `*attr_num` to the attribute code of the attribute whose index is specified by `index` in the given abbreviation.

The index starts at 0.

The location pointed to by `returned_attr_num` is set to the attribute number (example: `DW_AT_name`). The location pointed to by `returned_form` is set to the form of the attribute (example: `DW_FORM_string`). The location pointed to by `returned_implicit_const` is set to the implicit const value if and only if the

FORM returned is DW\_FORM\_implicit\_const

The location pointed to by `offset` is set to the byte offset of the attribute in the abbreviations section.

The function returns DW\_DLV\_NO\_ENTRY if the index specified is outside the range of attributes in this abbreviation.

The function returns DW\_DLV\_ERROR on error and sets `*error` to an error value instance.

### 6.22.6 dwarf\_get\_abbrev\_entry()

```
int dwarf_get_abbrev_entry(  
    Dwarf_Abbrev abbrev,  
    Dwarf_Signed index,  
    Dwarf_Half   *attr_num,  
    Dwarf_Signed *form,  
    Dwarf_Off   *offset,  
    Dwarf_Error  *error)
```

This function cannot return DW\_FORM\_implicit\_const const values. When convenient all callers should switch to using the `dwarf_get_abbrev_entry_b()` function.

If successful, `dwarf_get_abbrev_entry()` returns DW\_DLV\_OK and sets `*attr_num` to the attribute code of the attribute whose index is specified by `index` in the given abbreviation. The index starts at 0. The location pointed to by `form` is set to the form of the attribute. The location pointed to by `offset` is set to the byte offset of the attribute in the abbreviations section.

It returns DW\_DLV\_NO\_ENTRY if the index specified is outside the range of attributes in this abbreviation.

It returns DW\_DLV\_ERROR on error.

## 6.23 String Section Operations

The `.debug_str` section contains only strings. Debuggers need never use this interface: it is only for debugging problems with the string section itself.

### 6.23.1 dwarf\_get\_string\_section\_name()

```
int dwarf_get_string_section_name(Dwarf_Debug dbg,  
    const char ** sec_name,  
    Dwarf_Error  *error)
```

`dwarf_get_string_section_name()` lets consumers access the object string

section name. This is useful for applications wanting to print the name, but of course the object section name is not really a part of the DWARF information. Most applications will probably not call this function. It can be called at any time after the Dwarf\_Debug initialization is done. See also `dwarf_get_die_section_name_b()`.

The function `dwarf_get_string_section_name()` operates on the the `.debug_string[.dwo]` section.

If the function succeeds, `*sec_name` is set to a pointer to a string with the object section name and the function returns `DW_DLV_OK`. Do not free the string whose pointer is returned. For non-Elf objects it is possible the string pointer returned will be NULL or will point to an empty string. It is up to the calling application to recognize this possibility and deal with it appropriately.

If the section does not exist the function returns `DW_DLV_NO_ENTRY`.

If there is an internal error detected the function returns `DW_DLV_ERROR` and sets the `*error` pointer.

### 6.23.2 dwarf\_get\_str()

```
int dwarf_get_str(  
    Dwarf_Debug    dbg,  
    Dwarf_Off      offset,  
    char           **string,  
    Dwarf_Signed   *returned_str_len,  
    Dwarf_Error    *error)
```

The function `dwarf_get_str()` returns `DW_DLV_OK` and sets `*returned_str_len` to the length of the string, not counting the null terminator, that begins at the offset specified by `offset` in the `.debug_str` section. The location pointed to by `string` is set to a pointer to this string. The next string in the `.debug_str` section begins at the previous `offset + 1 + *returned_str_len`. A zero-length string is NOT the end of the section. If there is no `.debug_str` section, `DW_DLV_NO_ENTRY` is returned. If there is an error, `DW_DLV_ERROR` is returned. If we are at the end of the section (that is, `offset` is one past the end of the section) `DW_DLV_NO_ENTRY` is returned. If the `offset` is some other too-large value then `DW_DLV_ERROR` is returned.

## 6.24 String Offsets Section Operations

The `.debug_str_offsets` section contains only table arrays (with headers) and Debuggers should never need to use this interface. The normal string access functions use the section tables transparently. The functions here are only intended to allow `dwarfdump` (or the like) print the section completely and to help compiler developers look for bugs in the section.

**Figure 42.** examplestringoffsets dwarf\_open\_str\_offsets\_table\_access() etc

```
void examplestringoffsets(Dwarf_Debug dbg)
{
    int res = 0;
    Dwarf_Str_Offsets_Table sot = 0;
    Dwarf_Unsigned wasted_byte_count = 0;
    Dwarf_Unsigned table_count = 0;
    Dwarf_Error error = 0;

    res = dwarf_open_str_offsets_table_access(dbg, &sot,&error);
    if(res == DW_DLV_NO_ENTRY) {
        /* No such table */
        return;
    }
    if(res == DW_DLV_ERROR) {
        /* Something is very wrong. Print the error? */
        return;
    }
    for(;;) {
        Dwarf_Unsigned unit_length =0;
        Dwarf_Unsigned unit_length_offset =0;
        Dwarf_Unsigned table_start_offset =0;
        Dwarf_Half      entry_size = 0;
        Dwarf_Half      version =0;
        Dwarf_Half      padding =0;
        Dwarf_Unsigned table_value_count =0;
        Dwarf_Unsigned i = 0;
        Dwarf_Unsigned table_entry_value = 0;

        res = dwarf_next_str_offsets_table(sot,
            &unit_length, &unit_length_offset,
            &table_start_offset,
            &entry_size,&version,&padding,
            &table_value_count,&error);
        if (res == DW_DLV_NO_ENTRY) {
            /* We have dealt with all tables */
            break;
        }
        if (res == DW_DLV_ERROR) {
            /* Something badly wrong. Do something. */
            return;
        }
        /* One could call dwarf_str_offsets_statistics to
           get the wasted bytes so far, but we do not do that
           in this example. */
        /* Possibly print the various table-related values
           returned just above. */
    }
}
```

```
    for (i=0; i < table_value_count; ++i) {
        res = dwarf_str_offsets_value_by_index(sot,i,
            &table_entry_value,&error);
        if (res != DW_DLV_OK) {
            /* Something is badly wrong. Do something. */
            return;
        }
        /* Do something with the table_entry_value
           at this index. Maybe just print it.
           It is an offset in .debug_str. */
    }
}
res = dwarf_str_offsets_statistics(sot,&wasted_byte_count,
    &table_count,&error);
if (res == DW_DLV_OK) {
    /* The wasted byte count is set. Print it or something.
       One hopes zero bytes are wasted.
       Print the table count if one is interested. */
}
res = dwarf_close_str_offsets_table_access(sot,&error);
/* There is little point in checking the return value
   as little can be done about any error. */
sot = 0;
}
```

### 6.24.1 dwarf\_open\_str\_offsets\_table\_access()

```
int dwarf_open_str_offsets_table_access(
    Dwarf_Debug dbg,
    Dwarf_Str_Offsets_Table * table_data,
    Dwarf_Error * error);
```

`dwarf_open_str_offsets_table_access()` creates an opaque struct and returns a pointer to it on success. That struct pointer is used in all subsequent operations on the table. Through the function `dwarf_next_str_offsets_table()` the caller can iterate through each of the per-CU offset tables.

If there is no such section, or if the section is empty the function returns `DW_DLV_NO_ENTRY`.

If there is an error (such as out-of-memory) the function returns `DW_DLV_ERROR` and sets an error value through the error pointer.

### 6.24.2 dwarf\_close\_str\_offsets\_table\_access()

```
int
dwarf_close_str_offsets_table_access(
    Dwarf_Str_Offsets_Table table_data,
    Dwarf_Error * error);
```

On success, `dwarf_close_str_offsets_table_access()` frees any allocated data associated with the struct pointed to by `table_data` and returns `DW_DLV_OK`. It is up to the caller to set the `table_data` pointer to `NULL` if desired. The pointer is unusable at that point and any other calls to `libdwarf` using that pointer will fail.

It returns `DW_DLV_OK` on error. Any error suggests there is memory corruption or an error in the call. Something serious happened.

It never returns `DW_DLV_NO_ENTRY`, but if it did there would be nothing the caller could do anyway..

If one forgets to call this function the memory allocated will be freed automatically by to call to `dwarf_finish()`, as is true of all other data allocated by `libdwarf`.

### 6.24.3 dwarf\_next\_str\_offsets\_table()

```
int dwarf_next_str_offsets_table(
    Dwarf_Str_Offsets_Table table,
    Dwarf_Unsigned *unit_length_out,
    Dwarf_Unsigned *unit_length_offset_out,
    Dwarf_Unsigned *table_start_offset_out,
    Dwarf_Half *entry_size_out,
    Dwarf_Half *version_out,
    Dwarf_Half *padding_out,
    Dwarf_Unsigned *table_value_count_out,
    Dwarf_Error * error);
```

Each call to `dwarf_next_str_offsets_table()` returns the next String Offsets table in the `.debug_str_offsets` section. Typically there would be one such table for each CU in `.debug_info[dwo]` contributing to `.debug_str_offsets`. The `table` contains (internally, hidden) the section offset of the next table.

On success it returns `DW_DLV_OK` and sets various fields representing data about the current table (fields described below).

If there are no more tables it returns `DW_DLV_NO_ENTRY`.

On error it returns `DW_DLV_ERROR` and passes back error details through the `error` pointer.

The returned values are intended to let the caller understand the table header and the table



in detail. These pointers are only used if the call returned DW\_DLV\_OK.

`unit_length_out` is set to the `unit_length` of a String Offsets Table Header. Which means it gives the length, in bytes, of the data following the length value that belongs to this table.

`unit_length_offset_out` is set to the section offset of the table header.

`table_start_offset_out` is set to the section offset of the array of offsets in this table.

`entry_size_out` is set to the size of a table entry. Which is 4 for 32-bit offsets in this table and 8 for 64-bit offsets in this table.

`version_out` is set to the version number in the table header. The only current valid value is 5.

`padding_out` is set to the 16-bit padding value in the table header. In a correct table header the value is zero.

`table_value_count_out` is set to the number of entries in the array of offsets in this table. Each entry is `entry_size_out` bytes long. Use this value in calling `dwarf_str_offsets_value_by_index()`.

#### **6.24.4 dwarf\_str\_offsets\_value\_by\_index()**

```
int dwarf_str_offsets_value_by_index(  
    Dwarf_Str_Offsets_Table sot,  
    Dwarf_Unsigned index,  
    Dwarf_Unsigned *stroffset,  
    Dwarf_Error *error);
```

On success, `dwarf_str_offsets_value_by_index()` returns DW\_DLV\_OK and sets the offset from the array of string offsets in the current table at the input `index`.

Valid index values are zero through `table_value_count_out - 1`

A function is used instead of simply letting callers use pointers as `libdwarf` correctly handles endianness differences (between the system running `libdwarf` and the object file being inspected) so offsets can be reported properly.

DW\_DLV\_ERROR is returned on error.

DW\_DLV\_NO\_ENTRY is never returned.

#### **6.24.5 dwarf\_str\_offsets\_statistics()**

```
int dwarf_str_offsets_statistics(  
    Dwarf_Str_Offsets_Table table_data,  
    Dwarf_Unsigned * wasted_byte_count,  
    Dwarf_Unsigned * table_count,  
    Dwarf_Error * error);
```

Normally called after all tables have been inspected to return (through a pointer) the count of apparently-wasted bytes in the section. It can be called at any point that the Dwarf\_Str\_Offsets\_Table pointer is valid.

On error it returns DW\_DLV\_ERROR and sets an error value through the pointer.

DW\_DLV\_NO\_ENTRY is never returned.

On success it returns DW\_DLV\_OK and sets values through the two pointers. Calling just after each table is accessed by dwarf\_next\_str\_offsets\_table() will reveal the sum of all wasted bytes at that point in iterating through the section.

table\_count is the count of table headers encountered so far.

By wasted bytes we mean bytes in between tables. libdwarf has no idea whether any apparently-valid table data is in fact useless.

## 6.25 Address Range Operations

These functions provide information about address ranges. The content is in the .debug\_aranges section. Address ranges map ranges of pc values to the corresponding compilation-unit die that covers the address range. In the DWARF2,3,4 Standards this is described under "Accelerated Access" "Lookup by Address".

### 6.25.1 dwarf\_get\_aranges\_section\_name()

```
int dwarf_get_aranges_section_name(Dwarf_Debug dbg,  
    const char ** sec_name,  
    Dwarf_Error *error)
```

\*dwarf\_get\_aranges\_section\_name() retrieves the object file section name of the applicable aranges section. This is useful for applications wanting to print the name, but of course the object section name is not really a part of the DWARF information. Most applications will probably not call this function. It can be called at any time after the Dwarf\_Debug initialization is done.

If the function succeeds, \*sec\_name is set to a pointer to a string with the object section name and the function returns DW\_DLV\_OK. Do not free the string whose pointer is returned. For non-Elf objects it is possible the string pointer returned will be NULL or will point to an empty string. It is up to the calling application to recognize this possibility and deal with it appropriately.

If the section does not exist the function returns DW\_DLV\_NO\_ENTRY.

If there is an internal error detected the function returns DW\_DLV\_ERROR and sets the \*error pointer.

### 6.25.2 dwarf\_get\_aranges()

```
int dwarf_get_aranges(  
    Dwarf_Debug dbg,  
    Dwarf_Arange **aranges,  
    Dwarf_Signed * returned_arange_count,  
    Dwarf_Error *error)
```

The function dwarf\_get\_aranges() returns DW\_DLV\_OK and sets \*returned\_arange\_count to the count of the number of address ranges in the .debug\_aranges section (for all compilation units). It sets \*aranges to point to a block of Dwarf\_Arange descriptors, one for each address range. It returns DW\_DLV\_ERROR on error. It returns DW\_DLV\_NO\_ENTRY if there is no .debug\_aranges section.

This not only reads all the ranges, it also reads the per-compilation-unit headers in .debug\_aranges and verifies they make sense.

**Figure 43.** Exampleu dwarf\_get\_aranges()

```
void exampleu(Dwarf_Debug dbg)  
{  
    Dwarf_Signed count = 0;  
    Dwarf_Arange *arang = 0;  
    int res = 0;  
    Dwarf_Error error = 0;  
  
    res = dwarf_get_aranges(dbg, &arang,&count, &error);  
    if (res == DW_DLV_OK) {  
        Dwarf_Signed i = 0;  
  
        for (i = 0; i < count; ++i) {  
            /* use arang[i] */  
            dwarf_dealloc(dbg, arang[i], DW_DLA_ARANGE);  
        }  
        dwarf_dealloc(dbg, arang, DW_DLA_LIST);  
    }  
}
```

### 6.25.3 dwarf\_get\_arange()

```
int dwarf_get_arange(  
    Dwarf_Arange *aranges,  
    Dwarf_Unsigned arange_count,  
    Dwarf_Addr address,  
    Dwarf_Arange *returned_arange,  
    Dwarf_Error *error);
```

The function `dwarf_get_arange()` takes as input a pointer to a block of `Dwarf_Arange` pointers, and a count of the number of descriptors in the block. It then searches for the descriptor that covers the given address. If it finds one, it returns `DW_DLV_OK` and sets `*returned_arange` to the descriptor. It returns `DW_DLV_ERROR` on error. It returns `DW_DLV_NO_ENTRY` if there is no `.debug_aranges` entry covering that address.

### 6.25.4 dwarf\_get\_cu\_die\_offset()

```
int dwarf_get_cu_die_offset(  
    Dwarf_Arange arange,  
    Dwarf_Off *returned_cu_die_offset,  
    Dwarf_Error *error);
```

The function `dwarf_get_cu_die_offset()` takes a `Dwarf_Arange` descriptor as input, and if successful returns `DW_DLV_OK` and sets `*returned_cu_die_offset` to the offset in the `.debug_info` section of the compilation-unit DIE for the compilation-unit represented by the given address range. It returns `DW_DLV_ERROR` on error.

### 6.25.5 dwarf\_get\_arange\_cu\_header\_offset()

```
int dwarf_get_arange_cu_header_offset(  
    Dwarf_Arange arange,  
    Dwarf_Off *returned_cu_header_offset,  
    Dwarf_Error *error)
```

The function `dwarf_get_arange_cu_header_offset()` takes a `Dwarf_Arange` descriptor as input, and if successful returns `DW_DLV_OK` and sets `*returned_cu_header_offset` to the offset in the `.debug_info` section of the compilation-unit header for the compilation-unit represented by the given address range.

It returns DW\_DLV\_ERROR on error.

This function added Rev 1.45, June, 2001.

This function is declared as 'optional' in libdwarf.h on IRIX systems so the \_MIPS\_SYMBOL\_PRESENT predicate may be used at run time to determine if the version of libdwarf linked into an application has this function.

### 6.25.6 dwarf\_get\_arange\_info\_b()

```
int dwarf_get_arange_info_b(  
    Dwarf_Arange arange,  
    Dwarf_Unsigned *segment,  
    Dwarf_Unsigned *segment_entry_size;  
    Dwarf_Addr *start,  
    Dwarf_Unsigned *length,  
    Dwarf_Off *cu_die_offset,  
    Dwarf_Error *error)
```

The function dwarf\_get\_arange\_info\_b() returns DW\_DLV\_OK and stores segment number of the FIXME FIXME The the starting value of the address range in the location pointed to by start, the length of the address range in the location pointed to by length, and the offset in the .debug\_info section of the compilation-unit DIE for the compilation-unit represented by the address range. It returns DW\_DLV\_ERROR on error.

### 6.25.7 dwarf\_get\_arange\_info()

```
int dwarf_get_arange_info(  
    Dwarf_Arange arange,  
    Dwarf_Addr *start,  
    Dwarf_Unsigned *length,  
    Dwarf_Off *cu_die_offset,  
    Dwarf_Error *error)
```

This is the same as dwarf\_get\_arange\_info\_b() except that this earlier function does not have a way to return the segment information.

## 6.26 General Low Level Operations

This function is low-level and intended for use only by programs such as dwarf-dumpers.

### 6.26.1 dwarf\_get\_offset\_size()

```
int dwarf_get_offset_size(Dwarf_Debug dbg,
                          Dwarf_Half *offset_size,
                          Dwarf_Error *error)
```

The function `dwarf_get_offset_size()` returns `DW_DLV_OK` on success and sets the `*offset_size` to the size in bytes of an offset. In case of error, it returns `DW_DLV_ERROR` and does not set `*offset_size`.

The offset size returned is the overall address size, which can be misleading if different compilation units have different address sizes. Many ABIs have only a single address size per executable, but differing address sizes are becoming more common.

### 6.26.2 dwarf\_get\_address\_size()

```
int dwarf_get_address_size(Dwarf_Debug dbg,
                           Dwarf_Half *addr_size,
                           Dwarf_Error *error)
```

The function `dwarf_get_address_size()` returns `DW_DLV_OK` on success and sets the `*addr_size` to the size in bytes of an address. In case of error, it returns `DW_DLV_ERROR` and does not set `*addr_size`.

The address size returned is the overall address size, which can be misleading if different compilation units have different address sizes. Many ABIs have only a single address size per executable, but differing address sizes are becoming more common.

Use `dwarf_get_die_address_size()` instead whenever possible.

### 6.26.3 dwarf\_get\_die\_address\_size()

```
int dwarf_get_die_address_size(Dwarf_Die die,
                               Dwarf_Half *addr_size,
                               Dwarf_Error *error)
```

The function `dwarf_get_die_address_size()` returns `DW_DLV_OK` on success and sets the `*addr_size` to the size in bytes of an address. In case of error, it returns `DW_DLV_ERROR` and does not set `*addr_size`.

The address size returned is the address size of the compilation unit owning the `die`

This is the preferred way to get address size when the `Dwarf_Die` is known.

## 6.27 Ranges Operations DWARF5 (.debug\_rnglists)

These functions provide information about the address ranges indicated by a `DW_AT_ranges` attribute of a DIE. The ranges are recorded in the `.debug_rnglists` section.

The section requires that each group of ranges has a header and the compilation unit may have a `DW_AT_ranges_base` attribute that must be added to the `DW_AT_ranges` attribute value to get the true ranges offset.

(A compiler generating `DW_AT_ranges_base` will add a relocation for that attribute value but will not have to make the `DW_AT_ranges` attributes relocatable and will thus save space in the object (ie, .o) file and save link time.)

See DWARF5 Section 2.17.3 Non-Contiguous Address Ranges and Section 7.28 Range List Table.

Section 7.28 describes the header fields for a Range List Table. There will usually be many such tables, in some sequence, in the `.debug_rnglists` section. Here we call each header `Dwarf_Rnglists_Head` (a pointer to an opaque struct).

### 6.27.1 Getting rnglists data for a DIE

This set of interfaces provides access to the DWARF5 `.debug_rnglists` entries for a particular DIE. Here is an example using the functions described below:

**Figure 44.** Example .debug\_rnglist for attribute

```
int example_rnglist_for_attribute(Dwarf_Attribute attr,
Dwarf_Unsigned attrvalue, Dwarf_Error *error)
{
    /* attrvalue must be the DW_AT_ranges
       DW_FORM_rnglistx or DW_FORM_sec_offset value
       extracted from attr. */
    int res = 0;
    Dwarf_Half theform = 0;
    Dwarf_Unsigned entries_count;
    Dwarf_Unsigned global_offset_of_rle_set;
    Dwarf_Rnglists_Head rnglhead = 0;
    Dwarf_Unsigned i = 0;

    res = dwarf_rnglists_get_rle_head(attr,
        theform,
        attrvalue,
        &rnglhead,
        &entries_count,
        &global_offset_of_rle_set,
        error);
    if (res != DW_DLV_OK) {
        return res;
    }
    for (i = 0; i < entries_count; ++i) {
        unsigned entrylen = 0;
        unsigned code = 0;
        Dwarf_Unsigned lowpc = 0;
        Dwarf_Unsigned highpc = 0;

        /* Actual addresses are most likely what one
           wants to know, not the lengths/offsets
           recorded in .debug_rnglists. So we pass
           NULLs to avoid dealing with values we
           do not wish to see. */
        res = dwarf_get_rnglists_entry_fields(rnglhead,
            i, &entrylen, &code,
            0, 0,
            &lowpc, &highpc, error);
        if (res != DW_DLV_OK) {
            dwarf_dealloc_rnglists_head(rnglhead);
            return res;
        }
        if (code == DW_RLE_end_of_list) {
            /* we are done */
            break;
        }
    }
}
```



```
    }
    if (code == DW_RLE_base_addressx ||
        code == DW_RLE_base_address) {
        /* We do not need to use these, they
           have been accounted for already. */
        continue;
    }
    /* Here do something with lowpc and highpc, these
       are real addresses */
}
dwarf_dealloc_rnglists_head(rnglhead);
return DW_DLV_OK;
}
```

#### 6.27.1.1 dwarf\_rnglists\_get\_rle\_head()

This function is used to enable access to the specific set of rnglist entries applying to a specific DW\_AT\_ranges attribute.

```
int dwarf_rnglists_get_rle_head( Dwarf_Attribute attr,
    Dwarf_Half theform,
    Dwarf_Unsigned attr_val,
    Dwarf_Rnglists_Head *head_out,
    Dwarf_Unsigned *entries_count_out,
    Dwarf_Unsigned *global_offset_of_rle_set,
    Dwarf_Error *error);
```

Given a DW\_AT\_ranges Dwarf\_Attribute, the FORM from that attribute, and the value of the the attribute (which might be an index from DW\_FORM\_rnglistx or a section offset from DW\_FORM\_sec\_offset the function determines which Dwarf\_Rnglists\_Head applies and returns the pointer on success (meaning it returned . DW\_DLV\_OK). And on success it also returns the global offset of a set of rnglist entries within that particular Dwarf\_Rnglists\_Head (not needed except to show it to users) as well as the count of entries in that set (which is crucial to iterate through the rnglist entries applicable).

If not successful none of the pointers head\_out, entries\_count\_out, global\_offset will not be touched by the function.

If there is some problem with the section it will return DW\_DLV\_ERROR and return the error informatio through. \*error.

There is, currently, no situation in which it will return DW\_DLV\_NO\_ENTRY.

See dwarf\_dealloc\_rnglists\_head() below to release the storage allocated by a successful call here.

### 6.27.1.2 dwarf\_get\_rnglist\_head\_basics()

```
int dwarf_get_rnglist_head_basics(  
    Dwarf_Rnglists_Head head,  
    Dwarf_Unsigned * rle_count,  
    Dwarf_Unsigned * rle_version,  
    Dwarf_Unsigned * rnglists_index_returned,  
    Dwarf_Unsigned * bytes_total_in_rle,  
    unsigned * offset_size,  
    unsigned * address_size,  
    unsigned * segment_selector_size,  
    Dwarf_Unsigned * overall_offset_of_this_context,  
    Dwarf_Unsigned * total_length_of_this_context,  
    Dwarf_Bool * rnglists_base_present,  
    Dwarf_Unsigned * rnglists_base,  
    Dwarf_Bool * rnglists_base_address_present,  
    Dwarf_Unsigned * rnglists_base_address,  
    Dwarf_Bool * rnglists_debug_addr_base_present,  
    Dwarf_Unsigned * rnglists_debug_addr_base,  
    Dwarf_Error *error)
```

The function `dwarf_get_rnglist_head_basics()` allows caller to print or display the fields of the `Dwarf_Rnglists_Head` that might be of interest for understanding the section data for that `Dwarf_Rnglists_Head`.

It is not needed to access the rnglist data. It currently returns only `DW_DLV_OK`.

### 6.27.1.3 dwarf\_get\_rnglists\_entry\_fields()

```
int dwarf_get_rnglists_entry_fields(  
    Dwarf_Rnglists_Head head,  
    Dwarf_Unsigned entrynum,  
    unsigned *entrylen,  
    unsigned *code,  
    Dwarf_Unsigned *raw1,  
    Dwarf_Unsigned *raw2,  
    Dwarf_Unsigned *cooked1,  
    Dwarf_Unsigned *cooked2,  
    Dwarf_Error *err)
```

This is the function to access the rnglist entries for this `Dwarf_Rnglists_Head`. Call this with `entrynum` in the normal iteration "`i = 0; i < entries_count; ++i`".

where `entries_count` was returned by `dwarf_rnglists_get_rle_head()` through a pointer.

#### 6.27.1.4 `dwarf_dealloc_rnglists_head()`

```
int dwarf_dealloc_rnglists_head(Dwarf_Rnglists_Head /*head*/);
```

This frees the storage allocated by the `dwarf_rnglists_get_rle_head()` call that created the `Dwarf_Rnglists_Head` pointer.

It only returns `DW_DLV_OK`.

### 6.27.2 Getting raw `.debug_rnglists` entries

This set of interfaces is to read the (entire) `.debug_rnglists` section without reference to any DIE. As such these can only present the raw data from the file. There is no way in these interfaces to get actual addresses. These might be of interest if you want to know exactly what the compiler output in the `.debug_rnglists` section. "dwarfdump ---print-raw-rnglists" (try adding `-v` or `-vvv`) makes these calls.

Here is an example using all the following calls.

example\_rngl

**Figure 45.** Example of `dwarf_get_ranges_a()`

```
int example_raw_rnglist(Dwarf_Debug dbg,Dwarf_Error *error)
{
    Dwarf_Unsigned count = 0;
    int res = 0;
    Dwarf_Unsigned i = 0;

    res = dwarf_load_rnglists(dbg,&count,error);
    if (res != DW_DLV_OK) {
        return res;
    }
    for(i=0 ; i < count ; ++i) {
        Dwarf_Unsigned header_offset = 0;
        Dwarf_Small    offset_size = 0;
        Dwarf_Small    extension_size = 0;
        unsigned        version = 0; /* 5 */
        Dwarf_Small    address_size = 0;
        Dwarf_Small    segment_selector_size = 0;
        Dwarf_Unsigned offset_entry_count = 0;
        Dwarf_Unsigned offset_of_offset_array = 0;
        Dwarf_Unsigned offset_of_first_rangeentry = 0;
        Dwarf_Unsigned offset_past_last_rangeentry = 0;

        res = dwarf_get_rnglist_context_basics(dbg,i,
            &header_offset,&offset_size,&extension_size,
            &version,&address_size,&segment_selector_size,
            &offset_entry_count,&offset_of_offset_array,
            &offset_of_first_rangeentry,
            &offset_past_last_rangeentry,error);
        if (res != DW_DLV_OK) {
            return res;
        }
        {
            Dwarf_Unsigned e = 0;
            unsigned colmax = 4;
            unsigned col = 0;
            Dwarf_Unsigned global_offset_of_value = 0;

            for ( ; e < offset_entry_count; ++e) {
                Dwarf_Unsigned value = 0;
                int resc = 0;

                resc = dwarf_get_rnglist_offset_index_value(dbg,
                    i,e,&value,
                    &global_offset_of_value,error);
                if (resc != DW_DLV_OK) {
                    return resc;
                }
            }
        }
    }
}
```

```
    }
    /* Do something */
    col++;
    if (col == colmax) {
        col = 0;
    }
}

}
{
    Dwarf_Unsigned curoffset = offset_of_first_rangeentry;
    Dwarf_Unsigned endoffset = offset_past_last_rangeentry;
    int rese = 0;
    Dwarf_Unsigned ct = 0;

    for ( ; curoffset < endoffset; ++ct ) {
        unsigned entrylen = 0;
        unsigned code = 0;
        Dwarf_Unsigned v1 = 0;
        Dwarf_Unsigned v2 = 0;
        rese = dwarf_get_rnglist_rle(dbg,i,
                                     curoffset,endoffset,
                                     &entrylen,
                                     &code,&v1,&v2,error);
        if (rese != DW_DLV_OK) {
            return rese;
        }
        curoffset += entrylen;
        if (curoffset > endoffset) {
            return DW_DLV_ERROR;
        }
    }
}
}
return DW_DLV_OK;
}
```

#### 6.27.2.1 dwarf\_load\_rnglists()

```
int dwarf_load_rnglists(
    Dwarf_Debug dbg,
    Dwarf_Unsigned *rnglists_count,
    Dwarf_Error *error)
```

On a successful call to `dwarf_load_rnglists()` the function returns `DW_DLV_OK`, sets `*rnglists_count` (if and only if `rnglists_count` is non-null) to the number of distinct section contents that exist. A small amount of data for each Range Line Table is recorded in `dbg` as a side effect. Normally `libdwarf` will have already called this, but if an application never requests any `.debug_info` data the section might not be loaded. If the section is loaded this returns very quickly and will set `*rnglists_count` just as described in this paragraph.

If there is no `.debug_rnglists` section in the object file this function returns `DW_DLV_NO_ENTRY`.

If something is malformed it returns `DW_DLV_ERROR` and sets `*error` to the applicable error pointer describing the problem.

There is no `dealloc` call. Calling `dwarf_finish()` releases the modest amount of memory recorded for this section as a side effect.

#### 6.27.2.2 `dwarf_get_rnglist_context_basics()`

```
int dwarf_get_rnglist_context_basics(Dwarf_Debug dbg,
    Dwarf_Unsigned context_index,
    Dwarf_Unsigned * header_offset,
    Dwarf_Small    * offset_size,
    Dwarf_Small    * extension_size,
    unsigned       * version, /* 5 */
    Dwarf_Small    * address_size,
    Dwarf_Small    * segment_selector_size,
    Dwarf_Unsigned * offset_entry_count,
    Dwarf_Unsigned * offset_of_offset_array,
    Dwarf_Unsigned * offset_of_first_rangeentry,
    Dwarf_Unsigned * offset_past_last_rangeentry,
    Dwarf_Error *   /*err*/);
```

On success this returns `DW_DLV_OK` and returns values through the pointer arguments (other than `dbg` or `error`)

A call to `dwarf_load_rnglists()` that succeeds gets you the count of contexts and `dwarf_get_rnglist_context_basics()` for any "`i`  $\geq 0$  and `i`  $<$  count" gets you the context values relevant to `.debug_rnglists`.

Any of the pointer-arguments for returning context values can be passed in as 0 (in which case they will be skipped).

You will want `*offset_entry_count` so you can call `dwarf_get_rnglist_offset_index_value()` usefully.

If the `context_index` passed in is out of range the function returns `DW_DLV_NO_ENTRY`

At the present time DW\_DLV\_ERROR is never returned.

#### 6.27.2.3 dwarf\_get\_rnglist\_offset\_index\_value()

```
int dwarf_get_rnglist_offset_index_value(Dwarf_Debug dbg,
    Dwarf_Unsigned context_index,
    Dwarf_Unsigned offsetentry_index,
    Dwarf_Unsigned * offset_value_out,
    Dwarf_Unsigned * global_offset_value_out,
    Dwarf_Error *error)
```

On success dwarf\_get\_rnglist\_offset\_index\_value() returns DW\_DLV\_OK, sets \* offset\_value\_out to the value in the Range List Table offset array, and sets \* global\_offset\_value\_out to the section offset (in .debug\_addr) of the offset value.

Pass in context\_index exactly as the same field passed to dwarf\_get\_rnglist\_context\_basics().

Pass in offset\_entry\_index based on the return field offset\_entry\_count from dwarf\_get\_rnglist\_context\_basics(), meaning for that context\_index an offset\_entry\_index >=0 and < offset\_entry\_count.

Pass in offset\_entry\_count exactly as the same field passed to dwarf\_get\_rnglist\_context\_basics().

If one of the indexes passed in is out of range DW\_DLV\_NO\_ENTRY will be returned and no return arguments touched.

If there is some corruption of DWARF5 data then DW\_DLV\_ERROR might be returned and \*error set to the error details.

#### 6.27.2.4 dwarf\_get\_rnglist\_rle()

```
int dwarf_get_rnglist_rle(  
    Dwarf_Debug dbg,  
    Dwarf_Unsigned contextnumber,  
    Dwarf_Unsigned entry_offset,  
    Dwarf_Unsigned endoffset,  
    unsigned *entrylen,  
    unsigned *entry_kind,  
    Dwarf_Unsigned *entry_operand1,  
    Dwarf_Unsigned *entry_operand2,  
    Dwarf_Error *error)
```

On success it returns a single DW\_RLE\* record (see dwarf.h) fields.

contextnumber is the number of the current rnglist context.

entry\_offset is the section offset (section-global offset) of the next record.

endoffset is one past the last entry in this rle context.

\*entrylen returns the length in the .debug\_rnglists section of the particular record returned. It's used to increment to the next record within this rnglist context.

\*entrykind returns is the DW\_RLE\* number.

Some record kinds have 1 or 0 operands, most have two operands (the records describing ranges).

If the contextnumber is out of range it will return DW\_DLV\_NO\_ENTRY.

If the .debug\_rnglists section is malformed or the entry\_offset is incorrect it may return DW\_DLV\_ERROR.

## 6.28 Ranges Operations DWARF3,4 (.debug\_ranges)

These functions provide information about the address ranges indicated by a DW\_AT\_ranges attribute (the ranges are recorded in the .debug\_ranges section) of a DIE. These functions apply to DWARF3 and DWARF4. Each call of dwarf\_get\_ranges\_a() or dwarf\_get\_ranges() returns a an array of Dwarf\_Ranges structs, each of which represents a single ranges entry. The struct is defined in libdwarf.h.

New in DWARF3, for DWARF3, and DWARF4 the section contains just ranges. The ranges are referenced by DW\_AT\_ranges attributes in various DIEs.

For DWARF5 the section requires that each group of ranges has a header and the compilation unit may have a DW\_AT\_ranges\_base attribute that must be added to the DW\_AT\_ranges attribute value to get the true ranges offset.

(A compiler generating DW\_AT\_ranges\_base will add a relocation for that attribute value but will not have to make the DW\_AT\_ranges attributes relocatable and will thus save space in the object (ie, .o) file and link time.)



### 6.28.1 dwarf\_get\_ranges\_section\_name()

```
int dwarf_get_ranges_section_name(Dwarf_Debug dbg,
    const char ** sec_name,
    Dwarf_Error *error)
```

`*dwarf_get_ranges_section_name()` retrieves the object file section name of the applicable ranges section. This is useful for applications wanting to print the name, but of course the object section name is not really a part of the DWARF information. Most applications will probably not call this function. It can be called at any time after the `Dwarf_Debug` initialization is done.

If the function succeeds, `*sec_name` is set to a pointer to a string with the object section name and the function returns `DW_DLV_OK`. Do not free the string whose pointer is returned. For non-Elf objects it is possible the string pointer returned will be `NULL` or will point to an empty string. It is up to the calling application to recognize this possibility and deal with it appropriately.

If the section does not exist the function returns `DW_DLV_NO_ENTRY`.

If there is an internal error detected the function returns `DW_DLV_ERROR` and sets the `*error` pointer.

### 6.28.2 dwarf\_get\_ranges()

This is the original call and it will work fine when all compilation units have the same `address_size`. There is no `die` argument to this original version of the function. Other arguments (and deallocation) match the use of `dwarf_get_ranges_a()` (described next).

### 6.28.3 dwarf\_get\_ranges\_a()

```
int dwarf_get_ranges_a(
    Dwarf_Debug dbg,
    Dwarf_Off  offset,
    Dwarf_Die  die,
    Dwarf_Ranges **ranges,
    Dwarf_Signed * returned_ranges_count,
    Dwarf_Unsigned * returned_byte_count,
    Dwarf_Error *error)
```

The function `dwarf_get_ranges_a()` returns `DW_DLV_OK` and sets `*returned_ranges_count` to the count of the number of address ranges in the group of ranges in the `.debug_ranges` section at offset `offset` (which ends with a pair of zeros of pointer-size). This function is new as of 27 April 2009.

This function is normally used when one has a DIE with the `DW_AT_ranges` attribute (whose value is the offset needed). The ranges thus apply to the DIE involved. This function never sees the per-compilation-unit headers in the `.debug_ranges` section so of course any errors in those headers are ignored.

See also `dwarf_get_aranges()`,

The `offset` argument should be the value of a `DW_AT_ranges` attribute of a Debugging Information Entry.

The `die` argument should be the value of a `Dwarf_Die` pointer of a `Dwarf_Die` with the attribute containing this range set offset. Because each compilation unit has its own `address_size` field this argument is necessary to to correctly read ranges. (Most executables have the same `address_size` in every compilation unit, but some ABIs allow multiple address sized in an executable). If a `NULL` pointer is passed in `libdwarf` assumes a single `address_size` is appropriate for all ranges records.

The call sets `*ranges` to point to a block of `Dwarf_Ranges` structs, one for each address range. It returns `DW_DLV_ERROR` on error. It returns `DW_DLV_NO_ENTRY` if there is no `.debug_ranges` section or if `offset` is past the end of the `.debug_ranges` section.

If the `*returned_byte_count` pointer is passed as non-`NULL` the number of bytes that the returned ranges were taken from is returned through the pointer (for example if the `returned_ranges_count` is 2 and the pointer-size is 4, then `returned_byte_count` will be 8). If the `*returned_byte_count` pointer is passed as `NULL` the parameter is ignored. The `*returned_byte_count` is only of use to certain dumper applications, most applications will not use it.

**Figure 46.** Example `dwarf_get_ranges_a()`

```
void examplev(Dwarf_Debug dbg,Dwarf_Unsigned offset,Dwarf_Die die)
{
    Dwarf_Signed count = 0;
    Dwarf_Ranges *ranges = 0;
    Dwarf_Unsigned bytes = 0;
    Dwarf_Error error = 0;
    int res = 0;
    res = dwarf_get_ranges_a(dbg,offset,die,
        &ranges,&count,&bytes,&error);
    if (res == DW_DLV_OK) {
        Dwarf_Signed i;
        for( i = 0; i < count; ++i ) {
            Dwarf_Ranges *cur = ranges+i;
            /* Use cur. */
            functionusingrange(cur);
        }
        dwarf_ranges_dealloc(dbg,ranges,count);
    }
}
```

#### 6.28.4 dwarf\_ranges\_dealloc()

```
int dwarf_ranges_dealloc(
    Dwarf_Debug dbg,
    Dwarf_Ranges *ranges,
    Dwarf_Signed range_count,
    );
```

The function `dwarf_ranges_dealloc()` takes as input a pointer to a block of `Dwarf_Ranges` array and the number of structures in the block. It frees all the data in the array of structures.

### 6.29 Gdb Index operations

These functions get access to the fast lookup tables defined by gdb and gcc and stored in the `.gdb_index` section. The section is of sufficient complexity that a number of function interfaces are needed. For additional information see "[https://sourceware.org/gdb/onlinedocs/gdb/\"Index-Section-Format.html#Index-Section-Format](https://sourceware.org/gdb/onlinedocs/gdb/\)". (We split the url to two pieces so it can fit on the printed page join the pieces to make a usable url).

### 6.29.1 dwarf\_gdbindex\_header()

```
int dwarf_gdbindex_header(Dwarf_Debug dbg,
    Dwarf_Gdbindex * gdbindexptr,
    Dwarf_Unsigned * version,
    Dwarf_Unsigned * cu_list_offset,
    Dwarf_Unsigned * types_cu_list_offset,
    Dwarf_Unsigned * address_area_offset,
    Dwarf_Unsigned * symbol_table_offset,
    Dwarf_Unsigned * constant_pool_offset,
    Dwarf_Unsigned * section_size,
    Dwarf_Unsigned * unused_reserved,
    const char ** section_name,
    Dwarf_Error * error);
```

The function `dwarf_gdbindex_header()` takes as input a pointer to a `Dwarf_Debug` structure and returns fields through various pointers.

If the function returns `DW_DLV_NO_ENTRY` there is no `.gdb_index` section and none of the return-pointer argument values are set.

If the function returns `DW_DLV_ERROR` `error` is set to indicate the specific error, but no other return-pointer arguments are touched.

If successful, the function returns `DW_DLV_OK` and other values are set. The other values are set as follows:

The field `*gdbindexptr` is set to an opaque pointer to a `libdwarf_internal` structure used as an argument to other `.gdbindex` functions below.

The remaining fields are set to values that are mostly of interest to a pretty-printer application. See the detailed layout specification for specifics. The values returned are recorded in the `Dwarf_Gdbindex` opaque structure for the other `gdbindex` functions documented below.

The field `*version` is set to the version of the `gdb` index header (2)..

The field `*cu_list_offset` is set to the offset (in the `.gdb_index` section) of the `cu-list`.

The field `*types_cu_list_offset` is set to the offset (in the `.gdb_index` section) of the `types-list`.

The field `*address_area_offset` is set to the offset (in the `.gdb_index` section) of the `address area`.

The field `*symbol_table_offset` is set to the offset (in the `.gdb_index` section) of the `symbol table`.

The field `*constant_pool_offset` is set to the offset (in the `.gdb_index` section) of the `constant pool`.

The field `*section_size` is set to the length of the `.gdb_index` section.

The field `*unused_reserved` is set to zero.

The field `*section_name` is set to the Elf object file section name (`.gdb_index`). If a non-Elf object file has such a section the value set might be `NULL` or might point to an empty string (NUL terminated), so code to account for `NULL` or empty.

The field `*error` is not set.

Here we show a use of the set of `cu_list` functions (using all the functions in one example makes it rather too long).

**Figure 47.** Examplew `dwarf_get_gdbindex_header()`

```
void examplew(Dwarf_Debug dbg)
{
    Dwarf_Gdbindex gindexptr = 0;
    Dwarf_Unsigned version = 0;
    Dwarf_Unsigned cu_list_offset = 0;
    Dwarf_Unsigned types_cu_list_offset = 0;
    Dwarf_Unsigned address_area_offset = 0;
    Dwarf_Unsigned symbol_table_offset = 0;
    Dwarf_Unsigned constant_pool_offset = 0;
    Dwarf_Unsigned section_size = 0;
    Dwarf_Unsigned reserved = 0;
    Dwarf_Error error = 0;
    const char * section_name = 0;
    int res = 0;
    res = dwarf_gdbindex_header(dbg, &gindexptr,
                                &version, &cu_list_offset, &types_cu_list_offset,
                                &address_area_offset, &symbol_table_offset,
                                &constant_pool_offset, &section_size,
                                &reserved, &section_name, &error);
    if (res == DW_DLV_NO_ENTRY) {
        return;
    } else if (res == DW_DLV_ERROR) {
        return;
    }
    {
        /* do something with the data */
        Dwarf_Unsigned length = 0;
        Dwarf_Unsigned typeslength = 0;
        Dwarf_Unsigned i = 0;
        res = dwarf_gdbindex_culist_array(gindexptr,
                                          &length, &error);
        /* Example actions. */
        if (res == DW_DLV_OK) {
            for(i = 0; i < length; ++i) {
                Dwarf_Unsigned cuoffset = 0;
                res = dwarf_gdbindex_culist_entry(gindexptr,
                                                  i, &cuoffset, &culength, &error);
                if (res == DW_DLV_OK) {
                    /* Do something with cuoffset, culength */
                }
            }
        }
        res = dwarf_gdbindex_types_culist_array(gindexptr,
                                                  &typeslength, &error);
        if (res == DW_DLV_OK) {
            for(i = 0; i < typeslength; ++i) {
```

```
        Dwarf_Unsigned cuoffset = 0;
        Dwarf_Unsigned tuoffset = 0;
        Dwarf_Unsigned culength = 0;
        Dwarf_Unsigned type_signature = 0;
        res = dwarf_gdbindex_types_culist_entry(gindexptr,
            i, &cuoffset, &tuoffset, &type_signature, &error);
        if (res == DW_DLV_OK) {
            /* Do something with cuoffset etc. */
        }
    }
}
dwarf_gdbindex_free(gindexptr);
}
```

### 6.29.2 dwarf\_gdbindex\_culist\_array()

```
int dwarf_gdbindex_culist_array(Dwarf_Gdbindex gdbindexptr,
    Dwarf_Unsigned * list_length,
    Dwarf_Error * error);
```

The function `dwarf_gdbindex_culist_array()` takes as input valid `Dwarf_Gdbindex` pointer.

While currently only `DW_DLV_OK` is returned one should test for `DW_DLV_NO_ENTRY` and `DW_DLV_ERROR` and do something sensible if either is returned.

If successful, the function returns `DW_DLV_OK` and returns the number of entries in the culist through the `list_length` pointer.

### 6.29.3 dwarf\_gdbindex\_culist\_entry()

```
int dwarf_gdbindex_culist_entry(Dwarf_Gdbindex gdbindexptr,
    Dwarf_Unsigned entryindex,
    Dwarf_Unsigned * cu_offset,
    Dwarf_Unsigned * cu_length,
    Dwarf_Error * error);
```

The function `dwarf_gdbindex_culist_entry()` takes as input valid `Dwarf_Gdbindex` pointer and an index into the culist array. Valid indexes are 0 through

`list_length -1.`

If it returns `DW_DLV_NO_ENTRY` there is a coding error. If it returns `DW_DLV_ERROR` there is an error of some kind and the error is indicated by the value returned through the `error` pointer.

On success it returns `DW_DLV_OK` and returns the `cu_offset` (the section global offset of the CU in `.debug_info`) and `cu_length` (the length of the CU in `.debug_info`) values through the pointers.

#### 6.29.4 `dwarf_gdbindex_types_culist_array()`

```
int dwarf_gdbindex_types_culist_array(Dwarf_Gdbindex /*gdbindexptr*/,
    Dwarf_Unsigned /*types_list_length*/,
    Dwarf_Error /*error*/);
```

The function `dwarf_gdbindex_types_culist_array()` takes as input valid `Dwarf_Gdbindex` pointer.

While currently only `DW_DLV_OK` is returned one should test for `DW_DLV_NO_ENTRY` and `DW_DLV_ERROR` and do something sensible if either is returned.

If successful, the function returns `DW_DLV_OK` and returns the number of entries in the types culist through the `list_length`

#### 6.29.5 `dwarf_gdbindex_types_culist_entry()`

```
int dwarf_gdbindex_types_culist_entry(
    Dwarf_Gdbindex gdbindexptr,
    Dwarf_Unsigned entryindex,
    Dwarf_Unsigned * cu_offset,
    Dwarf_Unsigned * tu_offset,
    Dwarf_Unsigned * type_signature,
    Dwarf_Error * error);
```

The function `dwarf_gdbindex_types_culist_entry()` takes as input valid `Dwarf_Gdbindex` pointer and an index into the types culist array. Valid indexes are 0 through `types_list_length -1.`

If it returns `DW_DLV_NO_ENTRY` there is a coding error. If it returns `DW_DLV_ERROR` there is an error of some kind. and the error is indicated by the value returned through the `error` pointer.

On success it returns `DW_DLV_OK` and returns the `tu_offset` (the section global offset of the CU in `.debug_types`) and `tu_length` (the length of the CU in



.debug\_types) values through the pointers. It also returns the type signature (a 64bit value) through the `type_signature` pointer.

#### 6.29.6 dwarf\_gdbindex\_addressarea()

```
int dwarf_gdbindex_addressarea(Dwarf_Gdbindex /*gdbindexptr*/,
    Dwarf_Unsigned /*addressarea_list_length*/,
    Dwarf_Error /*error*/);
```

The function `dwarf_addressarea()` takes as input valid `Dwarf_Gdbindex` pointer and returns the length of the address area through `addressarea_list_length`.

If it returns `DW_DLV_NO_ENTRY` there is a coding error. If it returns `DW_DLV_ERROR` there is an error of some kind. and the error is indicated by the value returned through the `error` pointer.

If successful, the function returns `DW_DLV_OK` and returns the number of entries in the address area through the `addressarea_list_length` pointer.

#### 6.29.7 dwarf\_gdbindex\_addressarea\_entry()

```
int dwarf_gdbindex_addressarea_entry(
    Dwarf_Gdbindex gdbindexptr,
    Dwarf_Unsigned entryindex,
    Dwarf_Unsigned * low_address,
    Dwarf_Unsigned * high_address,
    Dwarf_Unsigned * cu_index,
    Dwarf_Error * error);
```

The function `dwarf_addressarea_entry()` takes as input valid `Dwarf_Gdbindex` pointer and an index into the address area (valid indexes are zero through `addressarea_list_length - 1`).

If it returns `DW_DLV_NO_ENTRY` there is a coding error. If it returns `DW_DLV_ERROR` there is an error of some kind. and the error is indicated by the value returned through the `error` pointer.

If successful, the function returns `DW_DLV_OK` and returns The `low_address` `high_address` and `cu_index` through the pointers.

Given an open `Dwarf_Gdbindex` one uses the function as follows:

**Figure 48.** Example `gdbindex_dwarf_gdbindex_addressarea()`

```
void examplewgdindex(Dwarf_Gdbindex gdbindex)
{
    Dwarf_Unsigned list_len = 0;
    Dwarf_Unsigned i = 0;
    int res = 0;
    Dwarf_Error err = 0;

    res = dwarf_gdbindex_addressarea(gdbindex, &list_len, &err);
    if (res != DW_DLV_OK) {
        /* Something wrong, ignore the addressarea */
    }
    /* Iterate through the address area. */
    for( i = 0; i < list_len; i++) {
        Dwarf_Unsigned lowpc = 0;
        Dwarf_Unsigned highpc = 0;
        Dwarf_Unsigned cu_index = 0;
        res = dwarf_gdbindex_addressarea_entry(gdbindex, i,
            &lowpc, &highpc,
            &cu_index,
            &err);
        if (res != DW_DLV_OK) {
            /* Something wrong, ignore the addressarea */
            return;
        }
        /* We have a valid address area entry, do something
           with it. */
    }
}
```

### 6.29.8 `dwarf_gdbindex_symboltable_array()`

```
int dwarf_gdbindex_symboltable_array(Dwarf_Gdbindex gdbindexptr,
    Dwarf_Unsigned *symtab_list_length,
    Dwarf_Error *error);
```

One can look at the symboltable as a two-level table (with The outer level indexes through symbol names and the inner level indexes through all the compilation units that define that symbol (each symbol having a different number of compilation units, this is not a simple rectangular table).

The function `dwarf_gdbindex_symboltable_array()` takes as input valid `Dwarf_Gdbindex` pointer.

If it returns `DW_DLV_NO_ENTRY` there is a coding error. If it returns

DW\_DLV\_ERROR there is an error of some kind. and the error is indicated by the value returned through the `error` pointer.

If successful, the function returns `DW_DLV_OK` and returns The `syntab_list_length` through the pointer.

Given a valid `Dwarf_Gdbindex` pointer, one can access the entire symbol table as follows (using 'return' here to indicate we are giving up due to a problem while keeping the example code fairly short):

**Figure 49.** Example `dwarf_gdbindex_symboltable_array()`

```
void examplex(Dwarf_Gdbindex gdbindex)
{
    Dwarf_Unsigned symtab_list_length = 0;
    Dwarf_Unsigned i = 0;
    Dwarf_Error err = 0;
    int res = 0;

    res = dwarf_gdbindex_symboltable_array(gdbindex,
        &symtab_list_length, &err);
    if (res != DW_DLV_OK) {
        return;
    }
    for( i = 0; i < symtab_list_length; i++) {
        Dwarf_Unsigned symnameoffset = 0;
        Dwarf_Unsigned cuvecoffset = 0;
        Dwarf_Unsigned cuvec_len = 0;
        Dwarf_Unsigned ii = 0;
        const char *name = 0;
        res = dwarf_gdbindex_symboltable_entry(gdbindex, i,
            &symnameoffset, &cuvecoffset,
            &err);
        if (res != DW_DLV_OK) {
            return;
        }
        res = dwarf_gdbindex_string_by_offset(gdbindex,
            symnameoffset, &name, &err);
        if(res != DW_DLV_OK) {
            return;
        }
        res = dwarf_gdbindex_cuvector_length(gdbindex,
            cuvecoffset, &cuvec_len, &err);
        if( res != DW_DLV_OK) {
            return;
        }
        for(ii = 0; ii < cuvec_len; ++ii ) {
            Dwarf_Unsigned attributes = 0;
            Dwarf_Unsigned cu_index = 0;
            Dwarf_Unsigned reserved1 = 0;
            Dwarf_Unsigned symbol_kind = 0;
            Dwarf_Unsigned is_static = 0;

            res = dwarf_gdbindex_cuvector_inner_attributes(
                gdbindex, cuvecoffset, ii,
                &attributes, &err);
            if( res != DW_DLV_OK) {
```

```
        return;
    }
    /* 'attributes' is a value with various internal
       fields so we expand the fields. */
    res = dwarf_gdbindex_cuvector_instance_expand_value(gdbindex,
        attributes, &cu_index,&reserved1,&symbol_kind, &is_static,
        &err);
    if( res != DW_DLV_OK) {
        return;
    }
    /* Do something with the attributes. */
}
}
```

### 6.29.9 dwarf\_gdbindex\_symboltable\_entry()

```
int dwarf_gdbindex_symboltable_entry(
    Dwarf_Gdbindex  gdbindexptr,
    Dwarf_Unsigned  entryindex,
    Dwarf_Unsigned * string_offset,
    Dwarf_Unsigned * cu_vector_offset,
    Dwarf_Error    * error);
```

The function `dwarf_gdbindex_symboltable_entry()` takes as input valid `Dwarf_Gdbindex` pointer and an entry index(valid index values being zero through `syntab_list_length -1`).

If it returns `DW_DLV_NO_ENTRY` there is a coding error. If it returns `DW_DLV_ERROR` there is an error of some kind. and the error is indicated by the value returned through the `error` pointer.

If successful, the function returns `DW_DLV_OK` and returns The `string_offset` and `cu_vector_offset` through the pointers. See the example above which uses this function.

### 6.29.10 dwarf\_gdbindex\_cuvector\_length()

```
int dwarf_gdbindex_cuvector_length(
    Dwarf_Gdbindex  gdbindex,
    Dwarf_Unsigned  cuvector_offset,
    Dwarf_Unsigned * innercount,
    Dwarf_Error    * error);
```

The function `dwarf_gdbindex_cuvector_length()` takes as input valid `Dwarf_Gdbindex` pointer and an a cu vector offset.

If it returns `DW_DLV_NO_ENTRY` there is a coding error. If it returns `DW_DLV_ERROR` there is an error of some kind. and the error is indicated by the value returned through the `error` pointer.

If successful, the function returns `DW_DLV_OK` and returns the `inner_count` through the pointer. The `inner_count` is the number of compilation unit vectors for this array of vectors. See the example above which uses this function.

### 6.29.11 `dwarf_gdbindex_cuvector_inner_attributes()`

```
int dwarf_gdbindex_cuvector_inner_attributes(  
    Dwarf_Gdbindex gdbindex,  
    Dwarf_Unsigned cuvector_offset,  
    Dwarf_Unsigned innerindex,  
    /* The attr_value is a field of bits. For expanded version  
       use dwarf_gdbindex_cuvector_expand_value() */  
    Dwarf_Unsigned * attr_value,  
    Dwarf_Error * error);
```

The function `dwarf_gdbindex_cuvector_inner_attributes()` takes as input valid `Dwarf_Gdbindex` pointer and an a cu vector offset and a `inner_index` (valid `inner_index` values are zero through `inner_count - 1`).

If it returns `DW_DLV_NO_ENTRY` there is a coding error. If it returns `DW_DLV_ERROR` there is an error of some kind. and the error is indicated by the value returned through the `error` pointer.

If successful, the function returns `DW_DLV_OK` and returns The `attr_value` through the pointer. The `attr_value` is actually composed of several fields, see the next function which expands the value. See the example above which uses this function.

### 6.29.12 `dwarf_gdbindex_cuvector_instance_expand_value()`

```
int dwarf_gdbindex_cuvector_instance_expand_value(  
    Dwarf_Gdbindex  gdbindex,  
    Dwarf_Unsigned  attr_value,  
    Dwarf_Unsigned * cu_index,  
    Dwarf_Unsigned * reserved1,  
    Dwarf_Unsigned * symbol_kind,  
    Dwarf_Unsigned * is_static,  
    Dwarf_Error    * error);
```

The function `dwarf_gdbindex_cuvector_instance_expand_value()` takes as input valid `Dwarf_Gdbindex` pointer and an `attr_value`.

If it returns `DW_DLV_NO_ENTRY` there is a coding error. If it returns `DW_DLV_ERROR` there is an error of some kind. and the error is indicated by the value returned through the `error` pointer.

If successful, the function returns `DW_DLV_OK` and returns the following values through the pointers:

The `cu_index` field is the index in the applicable CU list of a compilation unit. For the purpose of indexing the CU list and the types CU list form a single array so the `cu_index` can be indicating either list.

The `symbol_kind` field is a small integer with the symbol kind( zero is reserved, one is a type, 2 is a variable or enum value, etc).

The `reserved1` field should have the value zero and is the value of a bit field defined as reserved for future use.

The `is_static` field is zero if the CU indexed is global and one if the CU indexed is static.

See the example above which uses this function.

### 6.29.13 dwarf\_gdbindex\_string\_by\_offset()

```
int dwarf_gdbindex_string_by_offset(  
    Dwarf_Gdbindex  gdbindexptr,  
    Dwarf_Unsigned  stringoffset,  
    const char      ** string_ptr,  
    Dwarf_Error     * error);
```

The function `dwarf_gdbindex_string_by_offset()` takes as input valid `Dwarf_Gdbindex` pointer and a `stringoffset`. If it returns `DW_DLV_NO_ENTRY` there is a coding error. If it returns `DW_DLV_ERROR` there is an error of some kind. and

the error is indicated by the value returned through the `error` pointer.

If it succeeds, the call returns a pointer to a string from the 'constant pool' through the `string_ptr`. The string pointed to must never be `free()`d.

See the example above which uses this function.

## 6.30 GNU linking (`.gnu_debuglink`, `.note.gnu.build-id`) operations

This section deals with the way GNU tools allow creation of DWARF separated from the executable file involved. See <https://sourceware.org/gdb/onlinedocs/gdb/Separate-Debug-Files.html> for more information (the line break is just to make this url print nicely, there is no space). The function here is new in September 2019, revised in October 2019. An example of use follows the description of arguments.

### 6.30.1 `dwarf_gnu_debuglink()`

```
int dwarf_gnu_debuglink(Dwarf_Debug dbg,
    char      **debuglink_path_returned,
    unsigned char **crc_returned,
    char      **debuglink_fullpath_returned,
    unsigned *buildid_type_returned,
    char      **buildid_returned,
    unsigned *buildid_length_returned,
    char      ***paths_returned,
    unsigned *paths_count_returned,
    Dwarf_Error* error);
```

This returns `DW_DLV_NO_ENTRY` if there is neither a `.gnu_debuglink` object-file section nor a `.note.gnu.build-id` section in the object file.

If there is an error it returns `DW_DLV_ERROR` and sets `*error` to point to the error value.

On success it returns `DW_DLV_OK` and sets the fields through the pointers as described below. Two fields must be `free()`d to avoid a memory leak. None of the other fields should be freed.

If there is a `.gnu_debuglink` section the first four fields will be set.

`*debuglink_path_returned` points to the null-terminated string in the section. It must not be `free()`d.

`*crc_returned` points to a 4-byte CRC value. The bytes pointed to are not a string.

`*debuglink_fullpath_returned` points to a full pathname derived from the `*debuglink_fullpath_returned` string. And then `*debuglink_fullpath_strlen` is set to the length of `*debuglink_fullpath_returned` just as `strlen()` would count the length.



Callers must free() `*debuglink_fullpath_returned`.

If there is a `.note.gnu.build-id` section the buildid fields will be set through the pointers.

`*buildid_type_returned` will be set to the value 3.

`*buildid_owner_name_returned` will be set to point to the null-terminated string which will be "GNU". Do not free() this.

`*buildid_returned` will be set to point to the group of bytes of length `*buildid_length_returned`. This is not a string and is not null-terminated. It is normally a 20-byte field to be used in its ascii-hex form. Do not free() this.

If `*paths_returned` is passed as NULL then no paths calculation will be made and `*paths_count_returned` is not referenced by libdwarf.

If `*paths_returned` is passed in non-NULL then `*paths_returned` and `*paths_count_returned` provide an array of pointers-to-strings (with the actual strings following the array) and the count of the pointers in the array. When the strings are no longer needed free() `*paths_returned`. The number of paths returned will depend on which (of the two) sections exist and on how many global paths have been set by `dwarf_add_debuglink_global_path()`. and defined by the rules described in the web page mentioned above. The default global path is `"/usr/lib/debug"` and that is set by libdwarf as `paths_returned[0]`.

An example of calling this function follows

**Figure 50.** Example debuglink ()

```
void exampledebuglink(Dwarf_Debug dbg)
{
    int      res = 0;
    char      *debuglink_path = 0;
    unsigned char *crc = 0;
    char      *debuglink_fullpath = 0;
    unsigned debuglink_fullpath_strlen = 0;
    unsigned buildid_type = 0;
    char *    buildidowner_name = 0;
    unsigned char *buildid_itself = 0;
    unsigned buildid_length = 0;
    char **   paths = 0;
    unsigned paths_count = 0;
    Dwarf_Error error = 0;
    unsigned i = 0;

    /* This is just an example if one knows
       of another place full-DWARF objects
       may be. "/usr/lib/debug" is automatically
       set. */
    res = dwarf_add_debuglink_global_path(dbg,
        "/some/path/debug",&error);
    if (res != DW_DLV_OK) {
        /* Something is wrong, but we'll ignore
           that. */
    }

    res = dwarf_gnu_debuglink(dbg,
        &debuglink_path,
        &crc,
        &debuglink_fullpath,
        &debuglink_fullpath_strlen,
        &buildid_type,
        &buildidowner_name,
        &buildid_itself,
        &buildid_length,
        &paths,
        &paths_count,
        &error);
    if (res == DW_DLV_ERROR) {
        /* Do something with the error */
        return;
    }
    if (res == DW_DLV_NO_ENTRY) {
        /* No such sections as .note.gnu.build-id
```

```
        or .gnu_debuglink */
    return;
}
if (debuglink_fullpath_strlen) {
    printf("debuglink      path: %s\n", debuglink_path);
    printf("crc length      : %u  crc: ", 4);
    for (i = 0; i < 4; ++i) {
        printf("%02x", crc[i]);
    }
    printf("\n");
    printf("debuglink fullpath: %s\n", debuglink_fullpath);
}
if (buildid_length) {
    printf("buildid type      : %u\n", buildid_type);
    printf("Buildid owner      : %s\n", buildidowner_name);
    printf("buildid byte count: %u\n", buildid_length);
    printf(" ");
    /* buildid_length should be 20. */
    for (i = 0; i < buildid_length; ++i) {
        printf("%02x", buildid_itself[i]);
    }
    printf("\n");
}
printf("Possible paths count %u\n", paths_count);
for ( ; i < paths_count; ++i) {
    printf("%2u: %s\n", i, paths[i]);
}
free(debuglink_fullpath);
free(paths);
return;
}
```

### 6.30.2 dwarf\_add\_debuglink\_global\_path()

```
int dwarf_add_debuglink_global_path(Dwarf_Debug dbg,
    const char * path,
    Dwarf_Error* error);
```

This is unlikely to return DW\_DLV\_ERROR unless one passes in a NULL instead of an open Dwarf\_Debug. It cannot return DW\_DLV\_NO\_ENTRY.

On success it returns DW\_DLV\_OK after adding the path to the global list recorded in the Dwarf\_Debug.

## 6.31 Debug Fission (.debug\_tu\_index, .debug\_cu\_index) operations

We name things "xu" as these sections have the same format so we let "x" stand for either section. The DWARF5 standard refers to Split Dwarf while libdwarf tends to refer to this as "Fission".

These functions get access to the index functions needed to access and print the contents of an object file which is an aggregate of .dwo objects. These sections are implemented in gcc/gdb and are DWARF5. The idea is that much debug information can be separated off into individual .dwo Elf objects and then aggregated simply into a single .dwp object so the executable need not have the complete debug information in it at runtime yet allow good debugging.

For additional information, see "<https://gcc.gnu.org/wiki/DebugFissionDWP>", "<https://gcc.gnu.org/wiki/DebugFission>", and "<http://www.bayarea.net/~cary/dwarf/Accelerated%20Access%20Diagram.png>" and as of 17 February 2017, the DWARF5 standard.

There are FORM access functions related to Debug Fission (Split Dwarf). See `dwarf_formaddr()` and `dwarf_get_debug_addr_index()` and `dwarf_get_debug_str_index()`.

The FORM with the hash value (for a reference to a type unit ) is `DW_FORM_ref_sig8`.

In a compilation unit of Debug Fission object (or a .dwp Package File) `DW_AT_dwo_id` the hash is expected to be `DW_FORM_data8`.

The DWARF5 standard defines the hash as an 8 byte value which we could use `Dwarf_Unsigned`. Instead (and mostly for type safety) we define the value as a structure whose type name is `Dwarf_Sig8`.

To look up a name in the hash (to find which CU(s) it exists in). use `dwarf_get_debugfission_for_key()` fP, defined below.

The second group of interfaces here beginning with `dwarf_get_xu_index_header()` are useful if one wants to print a `.debug_tu_index` or `.debug_cu_index` section.

To access DIE, macro, etc information the support is built into DIE, Macro, etc operations so applications usually won't need to use these operations at all.

### 6.31.1 Dwarf\_Debug\_Fission\_Per\_CU

```
#define DW_FISSION_SECT_COUNT 12
struct Dwarf_Debug_Fission_Per_CU_s {
    /* Do not free the string. It contains "cu" or "tu". */
    /* If this is not set (ie, not a CU/TU in DWP Package File)
       then pcu_type will be NULL. */
    const char * pcu_type;
    /* pcu_index is the index (range 1 to N )
       into the tu/cu table of offsets and the table
       of sizes. 1 to N as the zero index is reserved
       for special purposes. Not a value one
       actually needs. */
    Dwarf_Unsigned pcu_index;
    Dwarf_Sig8 pcu_hash; /* 8 byte */
    /* [0] has offset and size 0.
       [1]-[8] are DW_SECT_* indexes and the
       values are the offset and size
       of the respective section contribution
       of a single .dwo object. When pcu_size[n] is
       zero the corresponding section is not present. */
    Dwarf_Unsigned pcu_offset[DW_FISSION_SECT_COUNT];
    Dwarf_Unsigned pcu_size[DW_FISSION_SECT_COUNT];
    Dwarf_Unsigned unused1;
    Dwarf_Unsigned unused2;
};
```

The structure is used to return data to callers with the data from either `.debug_tu_index` or `.debug_cu_index` that is applicable to a single compilation unit or type unit.

Callers to the applicable functions (see below) should allocate the structure and zero all the bytes in it. The structure has a few fields that are presently unused. These are reserved for future use since it is impossible to alter the structure without breaking binary compatibility.

### 6.31.2 dwarf\_die\_from\_hash\_signature()

```
int dwarf_die_from_hash_signature(Dwarf_Debug dbg,
    Dwarf_Sig8 * hash_sig,
    const char * sig_type,
    Dwarf_Die* returned_die,
    Dwarf_Error* error);
```

The function `dwarf_die_from_hash_signature()` is the most direct way to go from the hash data from a `DW_FORM_ref_sig8` or a `DW_AT_dwo_id` (from `DW_FORM_data8`) to a DIE from a .dwp package file or a .dwo object file ( .dwo access not supported yet).

The caller passes in `dbg` which should be `Dwarf_Debug` open/initialized on a .dwp

package file (or a .dwo object file).

The caller also passes in `hash_sig`, a pointer to the hash signature for which the caller wishes to find a DIE.

The caller also passes in `sig_type` which must contain either "tu" (identifying the hash referring to a type unit) or "cu" (identifying the hash as referring to a compilation unit).

On success the function returns `DW_DLV_OK` and sets `*returned_die` to be a pointer to a valid DIE for the compilation unit or type unit. If the type is "tu" the DIE returned is the specific type DIE that the hash refers to. If the type is "cu" the DIE returned is the compilation unit DIE of the compilation unit referred to.

When appropriate the caller should free the space of the returned DIE by a call something like

```
dwarf_dealloc(dbg,die,DW_DLA_DIE);
```

If there is no DWP Package File section or the hash cannot be found the function returns `DW_DLV_NO_ENTRY` and leaves `returned_die` untouched. Only .dwo objects and .dwp package files have the package file index sections.

If there is an error of some sort the function returns `DW_DLV_ERROR`, leaves `returned_die` untouched, and sets `*error` to indicate the precise error encountered.

### 6.31.3 dwarf\_get\_debugfission\_for\_die()

```
int dwarf_get_debugfission_for_die(Dwarf_Die die,  
    Dwarf_Debug_Fission_Per_CU *percu_out,  
    Dwarf_Error * error);
```

The function `dwarf_get_debugfission_for_die()` returns the debug fission for the compilation unit the DIE is a part of. Any DIE in the compilation (or type) unit will get the same result.

On a call to this function ensure the pointed-to space is fully initialized.

On success the function returns `DW_DLV_OK` and fills in the fields of `*percu_out` for which it has data.

If there is no DWP Package File section the function returns `DW_DLV_NO_ENTRY` and leaves `*percu_out` untouched. Only .dwp package files have the package file index sections.

If there is an error of some sort the function returns `DW_DLV_ERROR`, leaves `*percu_out` untouched, and sets `*error` to indicate the precise error encountered.

### 6.31.4 dwarf\_get\_debugfission\_for\_key()

```
int dwarf_get_debugfission_for_key(Dwarf_Debug dbg,
    Dwarf_Sig8 *      key,
    const char *      key_type ,
    Dwarf_Debug_Fission_Per_CU * percu_out,
    Dwarf_Error *      error);
```

The function `dwarf_get_debugfission_for_key()` returns the debug fission data for the compilation unit in a .dwp package file.

If there is no DWP Package File section the function returns `DW_DLV_NO_ENTRY` and leaves `*percu_out` untouched. Only .dwp package files have the package file index sections.

If there is an error of some sort the function returns `DW_DLV_ERROR`, leaves `*percu_out` untouched, and sets `*error` to indicate the precise error encountered.

### 6.31.5 dwarf\_get\_xu\_index\_header()

```
int dwarf_get_xu_index_header(Dwarf_Debug dbg,
    const char * section_type, /* "tu" or "cu" */
    Dwarf_Xu_Index_Header * xuhdr,
    Dwarf_Unsigned * version_number,
    Dwarf_Unsigned * offsets_count /* L*/,
    Dwarf_Unsigned * units_count /* N*/,
    Dwarf_Unsigned * hash_slots_count /* M*/,
    const char ** sect_name,
    Dwarf_Error * err);
```

The function `dwarf_get_xu_index_header()` takes as input a valid `Dwarf_Debug` pointer and an `section_type` value, which must one of the strings `tu` or `cu`.

It returns `DW_DLV_NO_ENTRY` if the section requested is not in the object file.

It returns `DW_DLV_ERROR` there is an error of some kind. and the error is indicated by the value returned through the `error` pointer.

If successful, the function returns `DW_DLV_OK` and returns the following values through the pointers:

The `xuhdr` field is a pointer usable in other operations (see below).

The `version_number` field is a the index version number. For gcc before DWARF5 the version number is 2. For DWARF5 the version number is 5.

The `offsets_count` field is a the number of columns in the table of section offsets.

Sometimes known as L.

The `units_count` field is a the number of compilation units or type units in the index. Sometimes known as N.

The `hash_slots_count` field is a the number of slots in the hash table. Sometimes known as M.

The `sect_name` field is the name of the section in the object file. Because non-Elf objects may not use section names callers must recognize that the `sect_name` may be set to NULL (zero) or to point to the empty string and this is not considered an error.

An example of initializing and disposing of a `Dwarf_Xu_Index_Header` follows.

**Figure 51.** Example `dwarf_get_xu_index_header()`

```
void exampley(Dwarf_Debug dbg, const char *type)
{
    /* type is "tu" or "cu" */
    int res = 0;
    Dwarf_Xu_Index_Header xuhdr = 0;
    Dwarf_Unsigned version_number = 0;
    Dwarf_Unsigned offsets_count = 0; /* L */
    Dwarf_Unsigned units_count = 0; /* M */
    Dwarf_Unsigned hash_slots_count = 0; /* N */
    Dwarf_Error err = 0;
    const char * section_name = 0;

    res = dwarf_get_xu_index_header(dbg,
        type,
        &xuhdr,
        &version_number,
        &offsets_count,
        &units_count,
        &hash_slots_count,
        &section_name,
        &err);
    if (res == DW_DLV_NO_ENTRY) {
        /* No such section. */
        return;
    }
    if (res == DW_DLV_ERROR) {
        /* Something wrong. */
        return;
    }
    /* Do something with the xuhdr here . */
    dwarf_xu_header_free(xuhdr);
}
```



### 6.31.6 dwarf\_get\_xu\_index\_section\_type()

```
int dwarf_get_xu_index_section_type(  
    Dwarf_Xu_Index_Header xuhdr,  
    const char ** typename,  
    const char ** sectionname,  
    Dwarf_Error * error);
```

The function `dwarf_get_xu_section_type()` takes as input a valid `Dwarf_Xu_Index_Header`. It is only useful when one already has an open `xuhdr` but one does not know if this is a type unit or compilation unit index section.

If it returns `DW_DLV_NO_ENTRY` something is wrong (should never happen). If it returns `DW_DLV_ERROR` something is wrong and the `error` field is set to indicate a specific error.

If successful, the function returns `DW_DLV_OK` and sets the following arguments through the pointers:

`typename` is set to the string `tu` or `cu` to indicate the index is of a type unit or a compilation unit, respectively.

`sectionname` is set to name of the object file section. Because non-Elf objects may not use section names callers must recognize that the `sect_name` may be set to `NULL` (zero) or to point to the empty string and this is not considered an error.

Neither string should be `free()`d.

### 6.31.7 dwarf\_get\_xu\_header\_free()

```
void dwarf_xu_header_free(Dwarf_Xu_Index_Header xuhdr);
```

The function `dwarf_get_xu_header_free()` takes as input a valid `Dwarf_Xu_Index_Header` and frees all the special data allocated for this access type. Once called, any pointers returned by use of the `xuhdr` should be considered stale and unusable.

### 6.31.8 dwarf\_get\_xu\_hash\_entry()

```
int dwarf_get_xu_hash_entry(  
    Dwarf_Xu_Index_Header xuhdr,  
    Dwarf_Unsigned      index,  
    Dwarf_Sig8 *        hash_value,  
    Dwarf_Unsigned *    index_to_sections,  
    Dwarf_Error *        error);
```

The function `dwarf_get_xu_hash_entry()` takes as input a valid `Dwarf_Xu_Index_Header` and an `index` of a hash slot entry (valid hash slot index values are zero (0) through `hash_slots_count - 1` (M-1)).

If it returns `DW_DLV_NO_ENTRY` something is wrong

If it returns `DW_DLV_ERROR` something is wrong and the `error` field is set to indicate a specific error.

If successful, the function returns `DW_DLV_OK` and sets the following arguments through the pointers:

`hash_value` is set to the 64bit hash of the symbol name.

`index_to_sections` is set to the index into offset-size tables of this hash entry.

If both `hash_value` and `index_to_sections` are zero (0) then the hash slot is unused. `index_to_sections` is used in calls to the function `dwarf_get_xu_section_offset()` as the `row_index`.

An example of use follows.

**Figure 52.** Exemplez dwarf\_get\_xu\_hash\_entry()

```
void exemplez( Dwarf_Xu_Index_Header xuhdr,
              Dwarf_Unsigned hash_slots_count)
{
    /* hash_slots_count returned by
       dwarf_get_xu_index_header(), see above. */
    static Dwarf_Sig8 zerohashval;

    Dwarf_Error err = 0;
    Dwarf_Unsigned h = 0;

    for( h = 0; h < hash_slots_count; h++) {
        Dwarf_Sig8 hashval;
        Dwarf_Unsigned index = 0;
        int res = 0;

        res = dwarf_get_xu_hash_entry(xuhdr,h,
                                     &hashval,&index,&err);
        if (res == DW_DLV_ERROR) {
            /* Oops. hash_slots_count wrong. */
            return;
        } else if (res == DW_DLV_NO_ENTRY) {
            /* Impossible */
            return;
        } else if (!memcmp(&hashval,&zerohashval,
                           sizeof(Dwarf_Sig8))
                   && index == 0 ) {
            /* An unused hash slot */
            continue;
        }
        /* Here, hashval and index (a row index into
           offsets and lengths) are valid.
           But the row to be passed into
           various functions here is index-1. */
    }
}
```

### 6.31.9 dwarf\_get\_xu\_section\_names()

```
int dwarf_get_xu_section_names(  
    Dwarf_Xu_Index_Header xuhdr,  
    Dwarf_Unsigned      column_index,  
    Dwarf_Unsigned*     number,  
    const char **       name,  
    Dwarf_Error *       err);
```

The function `dwarf_get_xu_section_names()` takes as input a valid `Dwarf_Xu_Index_Header` and a `column_index` of a hash slot entry (valid `column_index` values are zero (0) through `offsets_count - 1` (L-1)).

If it returns `DW_DLV_NO_ENTRY` something is wrong

If it returns `DW_DLV_ERROR` something is wrong and the `error` field is set to indicate a specific error.

If successful, the function returns `DW_DLV_OK` and sets the following arguments through the pointers:

`number` is set to a number identifying which section this column applies to. For example, if the value is `DW_SECT_INFO` (1) the column came from a `.debug_info.dwo` section. See the table of `DW_SECT_` identifiers and assigned numbers in DWARF5.

`name` is set to the applicable spelling of the section identifier, for example `DW_SECT_INFO`.

### 6.31.10 dwarf\_get\_xu\_section\_offset()

```
int dwarf_get_xu_section_offset(  
    Dwarf_Xu_Index_Header xuhdr,  
    Dwarf_Unsigned      row_index,  
    Dwarf_Unsigned      column_index,  
    Dwarf_Unsigned*     sec_offset,  
    Dwarf_Unsigned*     sec_size,  
    Dwarf_Error *       error);
```

The function `dwarf_get_xu_section_offset()` takes as input a valid `Dwarf_Xu_Index_Header` and a `row_index` (see `dwarf_get_xu_hash_entry()` above) and a `column_index`.

Valid `row_index` values are zero (0) through `units_count-1` (N) but one uses `dwarf_get_xu_hash_entry()` (above) to get row index and it returns a 1-origin index as that is what the DWARF5 standard specifies. Since a zero index from `dwarf_get_xu_hash_entry()` means this is not an actual entry such must be skipped.

Hence it makes (some) sense to subtract one making a zero-origin as that is the sense of all but the first row of the offsets table.

Valid `column_index` values are zero (0) through `offsets_count - 1` (L-1).

If it returns `DW_DLV_NO_ENTRY` something is wrong.

If it returns `DW_DLV_ERROR` something is wrong and the `error` field is set to indicate a specific error.

If successful, the function returns `DW_DLV_OK` and sets the following arguments through the pointers:

`sec_offset`, (`base_offset`) is set to the base offset of the initial compilation-unit-header section taken from a .dwo object. The `base_offset` is the data from a single section of a .dwo object.

`sec_size` is set to the length of the original section taken from a .dwo object. This is the length in the applicable section in the .dwp over which the base offset applies.

An example of use of `dwarf_get_xu_section_names()` and `dwarf_get_xu_section_offset()` follows.

**Figure 53.** Exampleza dwarf\_get\_xu\_section\_names()

```
void exampleza(Dwarf_Xu_Index_Header xuhdr,
    Dwarf_Unsigned offsets_count, Dwarf_Unsigned index )
{
    Dwarf_Error err = 0;
    Dwarf_Unsigned col = 0;
    /* We use 'offsets_count' returned by
       a dwarf_get_xu_index_header() call.
       We use 'index' returned by a
       dwarf_get_xu_hash_entry() call. */
    for (col = 0; col < offsets_count; col++) {
        Dwarf_Unsigned off = 0;
        Dwarf_Unsigned len = 0;
        const char * name = 0;
        Dwarf_Unsigned num = 0;
        int res = 0;

        res = dwarf_get_xu_section_names(xuhdr,
            col, &num, &name, &err);
        if (res != DW_DLV_OK) {
            break;
        }
        res = dwarf_get_xu_section_offset(xuhdr,
            index-1, col, &off, &len, &err);
        if (res != DW_DLV_OK) {
            break;
        }
        /* Here we have the DW_SECT_ name and number
           and the base offset and length of the
           section data applicable to the hash
           that got us here.
           Use the values.*/
    }
}
```

## 6.32 TAG ATTR etc names as strings

These functions turn a value into a string. So applications wanting the string "DW\_TAG\_compile\_unit" given the value 0x11 (the value defined for this TAG) can do so easily.

The general form is

```
int dwarf_get_<something>_name(
    unsigned value,
    char **s_out,
);
```

If the value passed in is known, the function returns DW\_DLV\_OK and places a pointer to the appropriate string into `*s_out`. The string is in static storage and applications must never free the string. If the value is not known, DW\_DLV\_NO\_ENTRY is returned and `*s_out` is not set. DW\_DLV\_ERROR is never returned.

Libdwarf generates these functions at libdwarf build time by reading dwarf.h.

All these follow this pattern rigidly, so the details of each are not repeated for each function.

The choice of 'unsigned' for the value type argument (the code value) argument is somewhat arbitrary, 'int' could have been used.

The library simply assumes the value passed in is applicable. So, for example, passing a TAG value code to `dwarf_get_ACCESS_name()` is a coding error which libdwarf will process as if it was an accessibility code value. Examples of bad and good usage are:

**Figure 54.** Example of `dwarf_get_TAG_name()`

```
void examplezb(void)
{
    const char * out = 0;
    int res = 0;

    /* The following is wrong, do not do it! */
    res = dwarf_get_ACCESS_name(DW_TAG_entry_point,&out);
    /* Nothing one does here with 'res' or 'out'
       is meaningful. */

    /* The following is meaningful.*/
    res = dwarf_get_TAG_name(DW_TAG_entry_point,&out);
    if( res == DW_DLV_OK) {
        /* Here 'out' is a pointer one can use which
           points to the string "DW_TAG_entry_point". */
    } else {
        /* Here 'out' has not been touched, it is
           uninitialized. Do not use it. */
    }
}
```

### **6.32.1 dwarf\_get\_ACCESS\_name()**

Returns an accessibility code name through the `s_out` pointer.

### **6.32.2 dwarf\_get\_AT\_name()**

Returns an attribute code name through the `s_out` pointer.

### **6.32.3 dwarf\_get\_ATE\_name()**

Returns a base type encoding name through the `s_out` pointer.

### **6.32.4 dwarf\_get\_ADDR\_name()**

Returns an address type encoding name through the `s_out` pointer. As of this writing only `DW_ADDR_none` is defined in `dwarf.h`.

### **6.32.5 dwarf\_get\_ATCF\_name()**

Returns a SUN code flag encoding name through the `s_out` pointer. This code flag is entirely a DWARF extension.

### **6.32.6 dwarf\_get\_CHILDREN\_name()**

Returns a child determination name (which is seen in the abbreviations section data) through the `s_out` pointer. The only value this recognizes for a 'yes' value is 1. As a flag value this is not quite correct (any non-zero value means yes) but dealing with this is left up to client code (normally compilers really do emit a value of 1 for a flag).

### **6.32.7 dwarf\_get\_children\_name()**

Returns a child determination name through the `s_out` pointer, though this version is really a libdwarf artifact. The standard function is `dwarf_get_CHILDREN_name()` which appears just above. As a flag value this is not quite correct (any non-zero value means yes) but dealing with this is left up to client code (normally compilers really do emit a value of 1 for a flag).

### **6.32.8 dwarf\_get\_CC\_name()**

Returns a calling convention case code name through the `s_out` pointer.

### **6.32.9 dwarf\_get\_CFA\_name()**

Returns a call frame information instruction name through the `s_out` pointer.

### **6.32.10 dwarf\_get\_DS\_name()**

Returns a decimal sign code name through the `s_out` pointer.

### **6.32.11 dwarf\_get\_DSC\_name()**

Returns a discriminant descriptor code name through the `s_out` pointer.

### **6.32.12 dwarf\_get\_EH\_name()**

Returns a GNU exception header code name through the `s_out` pointer.



### **6.32.13 dwarf\_get\_END\_name()**

Returns an endian code name through the `s_out` pointer.

### **6.32.14 dwarf\_get\_FORM\_name()**

Returns an form code name through the `s_out` pointer.

### **6.32.15 dwarf\_get\_FRAME\_name()**

Returns a frame code name through the `s_out` pointer. These are dependent on the particular ABI, so unless the `dwarf.h` used to generate `libdwarf` matches your ABI these names are unlikely to be very useful and certainly won't be entirely appropriate.

### **6.32.16 dwarf\_get\_ID\_name()**

Returns an identifier case code name through the `s_out` pointer.

### **6.32.17 dwarf\_get\_INL\_name()**

Returns an inline code name through the `s_out` pointer.

### **6.32.18 dwarf\_get\_LANG\_name()**

Returns a language code name through the `s_out` pointer.

### **6.32.19 dwarf\_get\_LLE\_name()**

Returns a split-dwarf loclist code name through the `s_out` pointer.

### **6.32.20 dwarf\_get\_LNE\_name()**

Returns a line table extended opcode code name through the `s_out` pointer.

### **6.32.21 dwarf\_get\_LNS\_name()**

Returns a line table standard opcode code name through the `s_out` pointer.

### **6.32.22 dwarf\_get\_MACINFO\_name()**

Returns a macro information `macinfo` code name through the `s_out` pointer.

### **6.32.23 dwarf\_get\_MACRO\_name()**

Returns a DWARF5 macro information `macro` code name through the `s_out` pointer.

### **6.32.24 dwarf\_get\_OP\_name()**

Returns a DWARF expression operation code name through the `s_out` pointer.

### **6.32.25 dwarf\_get\_ORD\_name()**

Returns an array ordering code name through the `s_out` pointer.

### **6.32.26 dwarf\_get\_TAG\_name()**

Returns a TAG name through the `s_out` pointer.

### **6.32.27 dwarf\_get\_VIRTUALITY\_name()**

Returns a virtuality code name through the `s_out` pointer.

### **6.32.28 dwarf\_get\_VIS\_name()**

Returns a visibility code name through the `s_out` pointer.

## **6.33 Section Operations**

In checking DWARF in linkonce sections for correctness it has been found useful to have certain section-oriented operations when processing object files. Normally these operations are not needed or useful in a fully-linked executable or shared library.

While the code is written with Elf sections in mind, it is quite possible to process non-Elf objects with code that implements certain function pointers (see `struct Dwarf_Obj_Access_interface_s`).

So far no one with such non-elf code has come forward to open-source it.

### **6.33.1 dwarf\_get\_section\_count()**

```
int dwarf_get_section_count(  
    Dwarf_Debug dbg)
```

Returns a count of the number of object sections found.

If there is an incomplete or damaged `dbg` passed in this can return -1;

### **6.33.2 dwarf\_get\_section\_info\_by\_name()**

```
int dwarf_get_section_info_by_name(  
    const char *section_name,  
    Dwarf_Addr *section_addr,  
    Dwarf_Unsigned *section_size,  
    Dwarf_Error *error)
```

The function `dwarf_get_section_info_by_name()` returns `DW_DLV_OK` if the section given by `section_name` was seen by `libdwarf`. On success it sets `*section_addr` to the virtual address assigned to the section by the linker or compiler and `*section_size` to the size of the object section.

It returns `DW_DLV_ERROR` on error.

### 6.33.3 dwarf\_get\_section\_info\_by\_index()

```
int dwarf_get_section_info_by_index(  
    int section_index,  
    const char **section_name,  
    Dwarf_Addr *section_addr,  
    Dwarf_Unsigned *section_size,  
    Dwarf_Error *error)
```

The function `dwarf_get_section_info_by_index()` returns `DW_DLV_OK` if the section given by `section_index` was seen by `libdwarf`. `*section_addr` to the virtual address assigned to the section by the linker or compiler and `*section_size` to the size of the object section.

No free or deallocate of information returned should be done by callers.

## 6.34 Utility Operations

These functions aid in the management of errors encountered when using functions in the *libdwarf* library and releasing memory allocated as a result of a *libdwarf* operation.

For clients that wish to encode LEB numbers two interfaces are provided to the producer code's internal LEB function.

### 6.34.1 dwarf\_errno()

```
Dwarf_Unsigned dwarf_errno(  
    Dwarf_Error error)
```

The function `dwarf_errno()` returns the error number corresponding to the error specified by `error`.

### 6.34.2 dwarf\_errmsg()

```
const char* dwarf_errmsg(  
    Dwarf_Error error)
```

The function `dwarf_errmsg()` returns a pointer to a null-terminated error message string corresponding to the error specified by `error`. The string should not be deallocated using `dwarf_dealloc()`.

The string should be considered to be a temporary string. That is, the returned pointer may become stale if you do `libdwarf` calls on the `Dwarf_Debug` instance other than `dwarf_errmsg()` or `dwarf_errno()`. So copy the `errmsg` string ( or print it) but

do not depend on the pointer remaining valid past other libdwarf calls to the Dwarf\_Debug instance that detected an error.

### 6.34.3 dwarf\_errmsg\_by\_number()

```
const char* dwarf_errmsg_by_number(  
    Dwarf_Unside errcode)
```

The function dwarf\_errmsg\_by\_number() returns a pointer to a null-terminated error message string corresponding to the error number specified by errcode. The string should not be deallocated or freed. If the errcode is too large for the table of static error strings a string reflecting that fact is returned.

For some places in the code a Dwarf\_Error() is inconvenient and this function lets dwarfdump report better information in those cases.

Function new December 19, 2018.

### 6.34.4 dwarf\_get\_endian\_copy\_function()

```
void (*dwarf_get_endian_copy_function(Dwarf_Debug /*dbg*/))  
    (void *, const void * /*src*/, unsigned long /*srcclen*/)
```

When reader client code wants to extract endian-dependent integers from dwarf and the existing interfaces won't do that (for example in printing frame instructions as done by dwarfdump) dwarf\_get\_endian\_copy\_function helps by returning the proper copy function needed, the one libdwarf itself uses. The client code needs a bit of glue to finish the job, as demonstrated by the ASNAR macro in dwarfdump/print\_frames.c

On success this returns a pointer to the correct copy function.

On failure it returns the null pointer. It's up to the client code to decide how to deal with the situation. In no reasonable case will the null pointer be returned.

New December 2018.

### 6.34.5 dwarf\_get\_harmless\_error\_list()

```
int dwarf_get_harmless_error_list(Dwarf_Debug dbg,  
    unsigned count,  
    const char ** errmsg_ptrs_array,  
    unsigned * newerr_count);
```

The harmless errors are not denoted by error returns from the other libdwarf functions. Instead, this function returns strings of any harmless errors that have been seen in the

current object. Clients never need call this, but if a client wishes to report any such errors it may call.

Only a fixed number of harmless errors are recorded. It is a circular list, so if more than the current maximum is encountered older harmless error messages are lost.

The caller passes in a pointer to an array of pointer-to-char as the argument `errmsg_ptrs_array`. The caller must provide this array, `libdwarf` does not provide it. The caller need not initialize the array elements.

The caller passes in the number of elements of the array of pointer-to-char thru `count`. Since the

If there are no unreported harmless errors the function returns `DW_DLV_NO_ENTRY` and the function arguments are ignored. Otherwise the function returns `DW_DLV_OK` and uses the arguments.

`libdwarf` assigns error strings to the `errmsg_ptrs_array`. The `MINIMUM(count-1, number of messages recorded)` pointers are assigned to the array. The array is terminated with a `NULL` pointer. (That is, one array entry is reserved for a `NULL` pointer). So if `count` is 5 up to 4 strings may be returned through the array, and one array entry is set to `NULL`.

Because the list is circular and messages may have been dropped the function also returns the actual error count of harmless errors encountered through `newerr_count` (unless the argument is `NULL`, in which case it is ignored).

Each call to this function resets the circular error buffer and the error count. So think of this call as reporting harmless errors since the last call to it.

The pointers returned through `errmsg_ptrs_array` are only valid till the next call to `libdwarf`. Do not save the pointers, they become invalid. Copy the strings if you wish to save them.

Calling this function neither allocates any space in memory nor frees any space in memory.

#### **6.34.6 dwarf\_insert\_harmless\_error()**

```
void dwarf_insert_harmless_error(Dwarf_Debug dbg,  
    char * newerror);
```

This function is used to test `dwarf_get_harmless_error_list`. It simply adds a harmless error string. There is little reason client code should use this function. It exists

so that the harmless error functions can be easily tested for correctness and leaks.

#### **6.34.7 dwarf\_set\_harmless\_error\_list\_size()**

```
unsigned dwarf_set_harmless_error_list_size(Dwarf_Debug dbg,  
      unsigned maxcount)
```

`dwarf_set_harmless_error_list_size` returns the number of harmless error strings the library is currently set to hold. If `maxcount` is non-zero the library changes the maximum it will record to be `maxcount`.

It is extremely unwise to make `maxcount` large because `libdwarf` allocates space for `maxcount` strings immediately.

The set of errors enumerated in Figure 8 below were defined in Dwarf 1. These errors are not used by the `libdwarf` implementation for Dwarf 2 or later.

<b>SYMBOLIC NAME</b>	<b>DESCRIPTION</b>
DW_DLE_NE	No error (0)
DW_DLE_VMM	Version of DWARF information newer than libdwarf
DW_DLE_MAP	Memory map failure
DW_DLE_LEE	Propagation of libelf error
DW_DLE_NDS	No debug section
DW_DLE_NLS	No line section
DW_DLE_ID	Requested information not associated with descriptor
DW_DLE_IOF	I/O failure
DW_DLE_MAF	Memory allocation failure
DW_DLE_IA	Invalid argument
DW_DLE_MDE	Mangled debugging entry
DW_DLE_MLE	Mangled line number entry
DW_DLE_FNO	File descriptor does not refer to an open file
DW_DLE_FNR	File is not a regular file
DW_DLE_FWA	File is opened with wrong access
DW_DLE_NOB	File is not an object file
DW_DLE_MOF	Mangled object file header
DW_DLE_EOLL	End of location list entries
DW_DLE_NOLL	No location list section
DW_DLE_BADOFF	Invalid offset
DW_DLE_EOS	End of section
DW_DLE_ATRUNC	Abbreviations section appears truncated
DW_DLE_BADBITC	Address size passed to dwarf bad

**Figure 55.** Dwarf Error Codes

The set of errors returned by `Libdwarf` functions is listed below. The list does lengthen: the ones listed here are far from a complete list. Some of the errors are SGI specific. See `libdwarf/dwarf_errmsg_list.h` for the complete list.

<b>SYMBOLIC NAME (description not shown here)</b>
DW_DLE_DBG_ALLOC
DW_DLE_FSTAT_ERROR
DW_DLE_FSTAT_MODE_ERROR
DW_DLE_INIT_ACCESS_WRONG
DW_DLE_ELF_BEGIN_ERROR
DW_DLE_ELF_GETEHDR_ERROR
DW_DLE_ELF_GETSHDR_ERROR
DW_DLE_ELF_STRPTR_ERROR
DW_DLE_DEBUG_INFO_DUPLICATE
DW_DLE_DEBUG_INFO_NULL
DW_DLE_DEBUG_ABBREV_DUPLICATE
DW_DLE_DEBUG_ABBREV_NULL
DW_DLE_DEBUG_ARANGES_DUPLICATE
DW_DLE_DEBUG_ARANGES_NULL
DW_DLE_DEBUG_LINE_DUPLICATE
DW_DLE_DEBUG_LINE_NULL
DW_DLE_DEBUG_LOC_DUPLICATE
DW_DLE_DEBUG_LOC_NULL
DW_DLE_DEBUG_MACINFO_DUPLICATE
DW_DLE_DEBUG_MACINFO_NULL
DW_DLE_DEBUG_PUBNAMES_DUPLICATE
DW_DLE_DEBUG_PUBNAMES_NULL
DW_DLE_DEBUG_STR_DUPLICATE
DW_DLE_DEBUG_STR_NULL
DW_DLE_CU_LENGTH_ERROR
DW_DLE_VERSION_STAMP_ERROR
DW_DLE_ABBREV_OFFSET_ERROR
DW_DLE_ADDRESS_SIZE_ERROR
DW_DLE_DEBUG_INFO_PTR_NULL
DW_DLE_DIE_NULL
DW_DLE_STRING_OFFSET_BAD
DW_DLE_DEBUG_LINE_LENGTH_BAD
DW_DLE_LINE_PROLOG_LENGTH_BAD
DW_DLE_LINE_NUM_OPERANDS_BAD
DW_DLE_LINE_SET_ADDR_ERROR

**Figure 56.** Dwarf 2 and later Error Codes

This list of errors is not complete; additional errors have been added. Some of the above errors may be unused. Errors may not have the same meaning in different releases. Since most error codes are returned from only one place (or a very small number of places) in the source it is normally very useful to simply search the `libdwarf` source to find out where a particular error code is generated. See `libdwarf/dwarf_errmsg_list.h` for the complete message set with short descriptions.



### 6.34.8 dwarf\_dealloc()

```
void dwarf_dealloc(  
    Dwarf_Debug dbg,  
    void* space,  
    Dwarf_Unsigned type)
```

The function `dwarf_dealloc` frees the dynamic storage pointed to by `space`, and allocated to the given `Dwarf_Debug`. The argument `type` is an integer code that specifies the allocation type of the region pointed to by the `space`. Refer to section 4 for details on *libdwarf* memory management.

### 6.34.9 dwarf\_encode\_leb128()

```
int dwarf_encode_leb128(Dwarf_Unsigned val,  
    int * nbytes,  
    char * space,  
    int splen);
```

The function `dwarf_encode_leb128` encodes the value `val` in the caller-provided buffer that `space` points to. The caller-provided buffer must be at least `splen` bytes long.

The function returns `DW_DLV_OK` if the encoding succeeds. If `splen` is too small to encode the value, `DW_DLV_ERROR` will be returned.

If the call succeeds, the number of bytes of `space` that are used in the encoding are returned through the pointer `nbytes`

### 6.34.10 dwarf\_encode\_signed\_leb128()

```
int dwarf_encode_signed_leb128(Dwarf_Signed val,  
    int * nbytes,  
    char * space,  
    int splen);
```

The function `dwarf_encode_signed_leb128` is the same as `dwarf_encode_leb128` except that the argument `val` is signed.

## 6.35 Finding Memory Leaks

If you are using `dwarf_set_de_alloc_flag(0)` to turn off the garbage collection `dwarffinish()` does and you find memory leaks there are a couple specific tools provided that may ease the process of tracking down the errors you have made.

This chapter is new as of 26 March 2020.

### 6.35.1 Compiling libdwarf -DDEBUG=1

The first tool is to build libdwarf with options `-g -O0 -DDEBUG=1`. The `-O0` is simply to help a debugger, valgrind or other too identify source lines accurately. The `-DDEBUG=1` Turns on printf statements in `dwarf_alloc.c` and `dwarf_error.c` that emit lines like

```
libdwarfdetector ALLOC ret 0x... size
libdwarfdetector DEALLOC ret 0x... size
libdwarfdetector ALLOC creating error string
libdwarfdetector DEALLOC Now destruct error string
```

at each point of particular interest.

The first two relate to actually malloc/free. The `ret 0x...` will be a hex address of the pointer your code is presented for allocations inside libdwarf.

The second two relate to allocation/free of a string in Dwarf\_Error record when an error record with variable descriptive error information is being built/freed.

### 6.35.2 Making use of the output of -DDEBUG=1

A small Python 3 program (`alloctrack.py`) in the libdwarf regressiontests on SourceForge.net will read through a file with libdwarfdetector lines and report on mismatches in the alloc/dealloc counts for each memory-blob libdwarf created. All other lines are skipped.

This has been found very useful.

Since the regression tests are large and you won't otherwise need them a copy of `alloctrack.py` follows so you need not clone the test code.

```
#!/usr/bin/env python3
# Copyright 2020 David Anderson
# This Python code is hereby placed into the public domain
# for use by anyone for any purpose.

# Useful for finding the needle of
# a single leaking allocation
# in the haystack of all the libdwarfdetector
# lines libdwarf can emit if compiled -DDEBUG=1
import sys
import os

def trackallocs(fi, valdict):
    line = 0
    while True:
        line = int(line)+1
        try:
            recf = fi.readline()
        except EOFError:
            break
        if len(recf) < 1:
            # eof
            break
        rec = recf.strip()
        if rec.find("ALLOC") != -1:
            if rec.find("libdwarfdetector ALLOC ret 0x") != -1:
                wds = rec.split()
                off = wds[3]
                if off in valdict:
                    (allo, deallo) = valdict[off]
                    if int(allo) == 0:
                        r = (1, deallo)
                        valdict[off] = r
                    else:
                        print("Duplicate use of ", off, "line", line)
                        r = (int(allo)+1, deallo)
                        valdict[off] = r
                else:
                    allo = 1
                    deallo = 0
                    r = (allo, deallo)
                    valdict[off] = r
            continue

        if rec.find("libdwarfdetector DEALLOC ret 0x") != -1:
            wds = rec.split()
```

```
    off = wds[3]
    if off in valdict:
        (allo,deallo) = valdict[off]
        if int(deallo) == 0:
            r = (allo,1)
            valdict[off] = r
        else:
            print("Duplicate use of ",off,"line",line)
            r = (allo,int(deallo)+1)
            valdict[off] = r
    else:
        allo = 0
        deallo = 1
        r=(allo,deallo)
        valdict[off] = r
    continue

if __name__ == '__main__':
    if len(sys.argv) > 1:
        fname = sys.argv[1]
        try:
            file = open(fname,"r")
        except IOError as message:
            print("File could not be opened: ", fname, " ", message)
            sys.exit(1)
        else:
            file = sys.stdin

    vals = {}
    trackallocs(file,vals)
    for s in vals:
        (allo,deallo) = vals[s]
        if int(allo) != int(deallo):
            print("Mismatch on ",s," a vs d: ",allo,deallo)
        if int(allo) > 1:
            print("Reuse of ",s," a vs d: ",allo,deallo)
```



## CONTENTS

1. INTRODUCTION .....	1
1.1 Copyright .....	1
1.2 Purpose and Scope .....	1
1.3 Document History .....	2
1.4 Definitions .....	2
1.5 Overview .....	2
1.6 Items Changed .....	3
1.7 Items Removed .....	7
1.8 Revision History .....	7
2. Types Definitions .....	8
2.1 General Description .....	8
2.2 Scalar Types .....	8
2.3 Aggregate Types .....	9
2.3.1 Location Record .....	10
2.3.2 Location Description .....	10
2.3.3 Data Block .....	11
2.3.4 Frame Operation Codes: DWARF 2 .....	12
2.3.5 Frame Regtable: DWARF 2 .....	13
2.3.6 Frame Operation Codes: DWARF 3 (for DWARF2 and later ) .....	13
2.3.7 Frame Regtable: DWARF 3 (for DWARF2 and later) .....	14
2.3.8 Macro Details Record .....	15
2.4 Opaque Types .....	16
3. UTF-8 strings .....	20
4. Error Handling .....	20
4.1 Returned values in the functional interface .....	23
5. Memory Management .....	24
5.1 Read-only Properties .....	24
5.2 Storage Deallocation .....	24
5.2.1 dwarf_dealloc() .....	25
5.2.2 dwarf_dealloc_die() .....	25
5.2.3 dwarf_dealloc_error() .....	25
5.2.4 dwarf_dealloc_attribute() .....	26
6. Functional Interface .....	28

6.1	Initialization Operations .....	28
6.1.1	dwarf_init_path() .....	29
6.1.2	dwarf_init_b() .....	30
6.1.3	dwarf_init() .....	31
6.1.4	dwarf_set_de_alloc_flag() .....	32
6.1.5	Dwarf_Handler function .....	32
6.1.6	dwarf_elf_init_b() [deprecated 2019] .....	33
6.1.7	dwarf_elf_init() [deprecated 2019] .....	33
6.1.8	dwarf_get_elf() .....	34
6.1.9	dwarf_set_tied_dbg() .....	34
6.1.10	dwarf_get_tied_dbg() .....	35
6.1.11	dwarf_finish() .....	36
6.1.12	dwarf_set_stringcheck() .....	36
6.1.13	dwarf_set_reloc_application() .....	36
6.1.14	dwarf_record_cmdline_options() .....	37
6.1.15	dwarf_object_init_b() .....	37
6.1.16	dwarf_object_init() .....	38
6.1.17	dwarf_get_real_section_name() .....	38
6.1.18	dwarf_package_version() .....	39
6.2	Object Type Detectors .....	39
6.2.1	dwarf_object_detector_path() .....	39
6.2.2	dwarf_object_detector_fd() .....	40
6.3	Section Group Operations .....	41
6.3.1	dwarf_sec_group_sizes() .....	41
6.3.2	dwarf_sec_group_map() .....	42
6.4	Section size operations .....	45
6.4.1	dwarf_get_section_max_offsets_b() .....	45
6.4.2	dwarf_get_section_max_offsets() .....	45
6.5	Printf Callbacks .....	46
6.5.1	dwarf_register_printf_callback .....	46
6.5.2	Dwarf_Printf_Callback_Info_s .....	47
6.5.3	dwarf_printf_callback_function_type .....	47
6.5.4	Example of printf callback use in a C++ application using libdwarf .....	48
6.6	Debugging Information Entry Delivery Operations .....	48
6.6.1	dwarf_get_die_section_name() .....	48
6.6.2	dwarf_get_die_section_name_b() .....	49
6.6.3	dwarf_next_cu_header_d() .....	49
6.6.4	dwarf_next_cu_header_c() .....	51
6.6.5	dwarf_next_cu_header_b() .....	52

6.6.6	dwarf_next_cu_header()	52
6.6.7	dwarf_siblingof_b()	52
6.6.8	dwarf_siblingof()	54
6.6.9	dwarf_child()	54
6.6.10	dwarf_offdie_b()	55
6.6.11	dwarf_offdie()	56
6.6.12	dwarf_validate_die_sibling()	56
6.7	Debugging Information Entry Query Operations	57
6.7.1	dwarf_get_die_infotypes_flag()	57
6.7.2	dwarf_tag()	58
6.7.3	dwarf_dieoffset()	58
6.7.4	dwarf_addr_form_is_indexed()	58
6.7.5	dwarf_debug_addr_index_to_addr()	58
6.7.6	dwarf_die_CU_offset()	59
6.7.7	dwarf_die_offsets()	59
6.7.8	dwarf_ptr_CU_offset()	59
6.7.9	dwarf_CU_dieoffset_given_die()	60
6.7.10	dwarf_die_CU_offset_range()	61
6.7.11	dwarf_diename()	61
6.7.12	dwarf_die_text()	62
6.7.13	dwarf_die_abbrev_code()	62
6.7.14	dwarf_die_abbrev_children_flag()	63
6.7.15	dwarf_die_abbrev_global_offset()	63
6.7.16	dwarf_get_version_of_die()	63
6.7.17	dwarf_attrlist()	64
6.7.18	dwarf_hasattr()	65
6.7.19	dwarf_attr()	65
6.7.20	dwarf_lowpc()	66
6.7.21	dwarf_highpc_b()	66
6.7.22	dwarf_highpc()	67
6.7.23	dwarf_dietype_offset()	67
6.7.24	dwarf_offset_list()	67
6.7.25	dwarf_bytesize()	68
6.7.26	dwarf_bitsize()	69
6.7.27	dwarf_bitoffset()	69
6.7.28	dwarf_srclang()	69
6.7.29	dwarf_arrayorder()	70
6.8	Attribute Queries	70
6.8.1	dwarf_hasform()	70
6.8.2	dwarf_whatform()	71



6.8.3	dwarf_whatform_direct()	71
6.8.4	dwarf_whatattr()	71
6.8.5	dwarf_formref()	72
6.8.6	dwarf_global_formref()	72
6.8.7	dwarf_convert_to_global_offset()	73
6.8.8	dwarf_formaddr()	73
6.8.9	dwarf_get_debug_str_index()	74
6.8.10	dwarf_formflag()	74
6.8.11	dwarf_formudata()	75
6.8.12	dwarf_formsdata()	75
6.8.13	dwarf_formblock()	76
6.8.14	dwarf_formstring()	76
6.8.15	dwarf_formsig8()	76
6.8.16	dwarf_formexprloc()	77
6.8.17	dwarf_get_form_class()	77
6.8.18	dwarf_discr_list()	78
6.8.19	dwarf_discr_entry_u()	81
6.8.20	dwarf_discr_entry_s()	81
6.9	Location List Operations, Raw .debug_loclists	82
6.9.1	dwarf_load_loclists()	84
6.9.2	dwarf_get_loclist_context_basics()	85
6.9.3	dwarf_get_loclist_offset_index_value()	86
6.9.4	dwarf_get_loclist_lle()	86
6.10	Location List operations .debug_loc & .debug_loclists	87
6.10.1	dwarf_get_loclist_c()	87
6.10.2	dwarf_get_locdesc_entry_d()	91
6.10.3	dwarf_get_loclist_head_kind()	92
6.10.4	dwarf_get_location_op_value_d()	92
6.10.5	dwarf_loclist_from_expr_c()	93
6.10.6	dwarf_loc_head_c_dealloc()	96
6.10.7	dwarf_loclist_n()	96
6.10.8	dwarf_loclist()	98
6.10.9	dwarf_loclist_from_expr()	99
6.10.10	dwarf_loclist_from_expr_b()	100
6.10.11	dwarf_loclist_from_expr_a()	101
6.11	Line Number Operations	101
6.11.1	Get A Set of Lines (including skeleton line tables)	102
6.11.2	dwarf_srclines_b()	102
6.11.3	dwarf_get_line_section_name_from_die()	102
6.11.4	dwarf_srclines_from_linecontext()	103

6.11.5	dwarf_srclines_two_levelfrom_linecontext()	103
6.11.6	dwarf_srclines_dealloc_b()	104
6.12	Line Context Details (DWARF5 style)	108
6.12.1	dwarf_srclines_table_offset()	108
6.12.2	dwarf_srclines_version()	108
6.12.3	dwarf_srclines_comp_dir()	108
6.12.4	dwarf_srclines_files_indexes()	109
6.12.5	dwarf_srclines_files_count()	109
6.12.6	dwarf_srclines_files_data_b()	110
6.12.7	dwarf_srclines_files_data()	110
6.12.8	dwarf_srclines_include_dir_count()	111
6.12.9	dwarf_srclines_include_dir_data()	111
6.12.10	dwarf_srclines_subprog_count()	111
6.12.11	dwarf_srclines_subprog_data()	112
6.13	Get A Set of Lines (DWARF2,3,4 style)	112
6.13.1	dwarf_srclines()	112
6.14	Get the set of Source File Names	113
6.14.1	dwarf_srcfiles()	113
6.15	Get Information About a Single Line Table Line	114
6.15.1	dwarf_linebeginstatement()	114
6.15.2	dwarf_lineendsequence()	115
6.15.3	dwarf_lineno()	115
6.15.4	dwarf_line_srcfileno()	115
6.15.5	dwarf_lineaddr()	116
6.15.6	dwarf_lineoff()	116
6.15.7	dwarf_lineoff_b()	117
6.15.8	dwarf_linesrc()	117
6.15.9	dwarf_lineblock()	117
6.15.10	dwarf_is_addr_set()	118
6.15.11	dwarf_prologue_end_etc()	118
6.16	Accelerated Access By Name operations	119
6.16.1	Fine Tuning Accelerated Access	119
6.16.1.1	dwarf_return_empty_pubnames	119
6.16.1.2	dwarf_get_globals_header	120
6.16.2	Accelerated Access Pubnames	120
6.16.2.1	dwarf_get_globals()	120
6.16.2.2	dwarf_globname()	122
6.16.2.3	dwarf_global_die_offset()	122
6.16.2.4	dwarf_global_cu_offset()	122
6.16.2.5	dwarf_get_cu_die_offset_given_cu_header_offset() ...	123

6.16.2.6	dwarf_get_cu_die_offset_given_cu_header_offset() ...	123
6.16.2.7	dwarf_global_name_offsets()	124
6.16.3	Accelerated Access Pubtypes	124
6.16.3.1	dwarf_get_pubtypes()	124
6.16.3.2	dwarf_pubtypename()	125
6.16.3.3	dwarf_pubtype_type_die_offset()	126
6.16.3.4	dwarf_pubtype_cu_offset()	126
6.16.3.5	dwarf_pubtype_name_offsets()	126
6.16.4	Accelerated Access Weaknames	127
6.16.4.1	dwarf_get_weaknames()	127
6.16.4.2	dwarf_weakname()	129
6.16.4.3	dwarf_weak_cu_offset()	130
6.16.4.4	dwarf_weak_name_offsets()	130
6.16.5	Accelerated Access Funcnames	130
6.16.5.1	dwarf_get_funcs()	131
6.16.5.2	dwarf_funcname()	132
6.16.5.3	dwarf_func_die_offset()	132
6.16.5.4	dwarf_func_cu_offset()	133
6.16.5.5	dwarf_func_name_offsets()	133
6.16.6	Accelerated Access Typenames	133
6.16.6.1	dwarf_get_types()	134
6.16.6.2	dwarf_typename()	135
6.16.6.3	dwarf_type_die_offset()	136
6.16.6.4	dwarf_type_cu_offset()	136
6.16.6.5	dwarf_type_name_offsets()	136
6.16.7	Accelerated Access varnames	137
6.16.7.1	dwarf_get_vars()	137
6.16.7.2	dwarf_varname()	139
6.16.7.3	dwarf_var_die_offset()	139
6.16.7.4	dwarf_var_cu_offset()	139
6.16.7.5	dwarf_var_name_offsets()	139
6.17	Names Fast Access (DWARF5) .debug_names	140
6.17.1	dwarf_debugnames_header()	140
6.17.2	dwarf_debugnames_sizes()	141
6.17.3	dwarf_debugnames_cu_entry()	141
6.17.4	dwarf_debugnames_local_tu_entry()	142
6.17.5	dwarf_debugnames_foreign_tu_entry()	142
6.17.6	dwarf_debugnames_bucket()	142
6.17.7	dwarf_debugnames_name()	143
6.17.8	dwarf_debugnames_abbrev_by_index()" 143	.....

6.17.9	dwarf_debugnames_abbrev_by_code()	143
6.17.10	dwarf_debugnames_form_by_index()	144
6.17.11	dwarf_debugnames_entrypool()	144
6.17.12	dwarf_debugnames_entrypool_values()	144
6.18	Macro Information Operations (DWARF4, DWARF5)	145
6.18.1	Getting access	145
6.18.1.1	dwarf_get_macro_context()	145
6.18.1.2	dwarf_get_macro_context_by_offset()	146
6.18.1.3	dwarf_dealloc_macro_context()	146
6.18.2	Getting Macro Unit Header Data	151
6.18.2.1	dwarf_macro_context_head()	151
6.18.2.2	dwarf_macro_operands_table()	152
6.18.3	Getting Individual Macro Operations Data	153
6.18.3.1	dwarf_get_macro_op()	153
6.18.3.2	dwarf_get_macro_defundef()	154
6.18.3.3	dwarf_get_macro_startend_file()	155
6.18.3.4	dwarf_get_macro_import()	155
6.19	Macro Information Operations (DWARF2, DWARF3, DWARF4)	156
6.19.1	General Macro Operations	156
6.19.1.1	dwarf_find_macro_value_start()	156
6.19.2	Debugger Interface Macro Operations	156
6.19.3	Low Level Macro Information Operations	156
6.19.3.1	dwarf_get_macro_details()	156
6.20	Low Level Frame Operations	159
6.20.1	dwarf_get_frame_section_name()	162
6.20.2	dwarf_get_frame_section_name_eh_gnu()	162
6.20.3	dwarf_get_fde_list()	163
6.20.4	dwarf_get_fde_list_eh()	165
6.20.5	dwarf_get_cie_of_fde()	167
6.20.6	dwarf_get_fde_for_die()	168
6.20.7	dwarf_get_fde_range()	168
6.20.8	dwarf_get_cie_info()	169
6.20.9	dwarf_get_cie_index()	170
6.20.10	dwarf_get_fde_instr_bytes()	170
6.20.11	dwarf_get_fde_info_for_reg()	171
6.20.12	dwarf_get_fde_info_for_all_regs()	172
6.20.13	dwarf_fde_section_offset()	172
6.20.14	dwarf_cie_section_offset()	173
6.20.15	dwarf_set_frame_rule_table_size()	173

6.20.16	dwarf_set_frame_rule_initial_value()	173
6.20.17	dwarf_set_frame_cfa_value()	174
6.20.18	dwarf_set_frame_same_value()	174
6.20.19	dwarf_set_frame_undefined_value()	175
6.20.20	dwarf_set_default_address_size()	175
6.20.21	dwarf_get_fde_info_for_reg3()	175
6.20.22	dwarf_get_fde_info_for_reg3_b()	177
6.20.23	dwarf_get_fde_info_for_cfa_reg3()	178
6.20.24	dwarf_get_fde_info_for_cfa_reg3_b()	178
6.20.25	dwarf_get_fde_info_for_all_regs3()	179
6.20.26	dwarf_get_fde_n()	180
6.20.27	dwarf_get_fde_at_pc()	180
6.20.28	dwarf_expand_frame_instructions()	181
6.20.29	dwarf_get_fde_exception_info()	182
6.21	Location Expression Evaluation	182
6.21.1	Location List Internal-level Interface	183
6.21.1.1	dwarf_get_loclist_entry()	183
6.22	Abbreviations access	184
6.22.1	dwarf_get_abbrev()	184
6.22.2	dwarf_get_abbrev_tag()	185
6.22.3	dwarf_get_abbrev_code()	185
6.22.4	dwarf_get_abbrev_children_flag()	185
6.22.5	dwarf_get_abbrev_entry_b()	186
6.22.6	dwarf_get_abbrev_entry()	187
6.23	String Section Operations	187
6.23.1	dwarf_get_string_section_name()	187
6.23.2	dwarf_get_str()	188
6.24	String Offsets Section Operations	188
6.24.1	dwarf_open_str_offsets_table_access()	191
6.24.2	dwarf_close_str_offsets_table_access()	192
6.24.3	dwarf_next_str_offsets_table()	192
6.24.4	dwarf_str_offsets_value_by_index()	193
6.24.5	dwarf_str_offsets_statistics()	193
6.25	Address Range Operations	194
6.25.1	dwarf_get_aranges_section_name()	194
6.25.2	dwarf_get_aranges()	195
6.25.3	dwarf_get_arange()	196
6.25.4	dwarf_get_cu_die_offset()	196
6.25.5	dwarf_get_arange_cu_header_offset()	196
6.25.6	dwarf_get_arange_info_b()	197

6.25.7	dwarf_get_arange_info()	197
6.26	General Low Level Operations	197
6.26.1	dwarf_get_offset_size()	198
6.26.2	dwarf_get_address_size()	198
6.26.3	dwarf_get_die_address_size()	198
6.27	Ranges Operations DWARF5 (.debug_rnglists)	199
6.27.1	Getting rnglists data for a DIE	199
6.27.1.1	dwarf_rnglists_get_rle_head()	201
6.27.1.2	dwarf_get_rnglist_head_basics()	202
6.27.1.3	dwarf_get_rnglists_entry_fields()	202
6.27.1.4	dwarf_dealloc_rnglists_head()	203
6.27.2	Getting raw .debug_rnglists entries	203
6.27.2.1	dwarf_load_rnglists()	205
6.27.2.2	dwarf_get_rnglist_context_basics()	206
6.27.2.3	dwarf_get_rnglist_offset_index_value()	207
6.27.2.4	dwarf_get_rnglist_rle()	207
6.28	Ranges Operations DWARF3,4 (.debug_ranges)	208
6.28.1	dwarf_get_ranges_section_name()	209
6.28.2	dwarf_get_ranges()	209
6.28.3	dwarf_get_ranges_a()	209
6.28.4	dwarf_ranges_dealloc()	211
6.29	Gdb Index operations	211
6.29.1	dwarf_gdbindex_header()	212
6.29.2	dwarf_gdbindex_culist_array()	215
6.29.3	dwarf_gdbindex_culist_entry()	215
6.29.4	dwarf_gdbindex_types_culist_array()	216
6.29.5	dwarf_gdbindex_types_culist_entry()	216
6.29.6	dwarf_gdbindex_addressarea()	217
6.29.7	dwarf_gdbindex_addressarea_entry()	217
6.29.8	dwarf_gdbindex_symboltable_array()	218
6.29.9	dwarf_gdbindex_symboltable_entry()	221
6.29.10	dwarf_gdbindex_cuvector_length()	221
6.29.11	dwarf_gdbindex_cuvector_inner_attributes()	222
6.29.12	dwarf_gdbindex_cuvector_instance_expand_value()	222
6.29.13	dwarf_gdbindex_string_by_offset()	223
6.30	GNU linking (.gnu_debuglink, .note.gnu.build-id) operations	224
6.30.1	dwarf_gnu_debuglink()	224
6.30.2	dwarf_add_debuglink_global_path()	227
6.31	Debug Fission (.debug_tu_index, .debug_cu_index) operations	228
6.31.1	Dwarf_Debug_Fission_Per_CU	228

6.31.2	dwarf_die_from_hash_signature()	229
6.31.3	dwarf_get_debugfission_for_die()	230
6.31.4	dwarf_get_debugfission_for_key()	231
6.31.5	dwarf_get_xu_index_header()	231
6.31.6	dwarf_get_xu_index_section_type()	233
6.31.7	dwarf_get_xu_header_free()	233
6.31.8	dwarf_get_xu_hash_entry()	233
6.31.9	dwarf_get_xu_section_names()	235
6.31.10	dwarf_get_xu_section_offset()	236
6.32	TAG ATTR etc names as strings	238
6.32.1	dwarf_get_ACCESS_name()	240
6.32.2	dwarf_get_AT_name()	240
6.32.3	dwarf_get_ATE_name()	240
6.32.4	dwarf_get_ADDR_name()	240
6.32.5	dwarf_get_ATCF_name()	240
6.32.6	dwarf_get_CHILDREN_name()	240
6.32.7	dwarf_get_children_name()	240
6.32.8	dwarf_get_CC_name()	240
6.32.9	dwarf_get_CFA_name()	240
6.32.10	dwarf_get_DS_name()	240
6.32.11	dwarf_get_DSC_name()	240
6.32.12	dwarf_get_EH_name()	240
6.32.13	dwarf_get_END_name()	241
6.32.14	dwarf_get_FORM_name()	241
6.32.15	dwarf_get_FRAME_name()	241
6.32.16	dwarf_get_ID_name()	241
6.32.17	dwarf_get_INL_name()	241
6.32.18	dwarf_get_LANG_name()	241
6.32.19	dwarf_get_LLE_name()	241
6.32.20	dwarf_get_LNE_name()	241
6.32.21	dwarf_get_LNS_name()	241
6.32.22	dwarf_get_MACINFO_name()	241
6.32.23	dwarf_get_MACRO_name()	241
6.32.24	dwarf_get_OP_name()	241
6.32.25	dwarf_get_ORD_name()	241
6.32.26	dwarf_get_TAG_name()	241
6.32.27	dwarf_get_VIRTUALITY_name()	242
6.32.28	dwarf_get_VIS_name()	242
6.33	Section Operations	242
6.33.1	dwarf_get_section_count()	242

6.33.2	dwarf_get_section_info_by_name()	242
6.33.3	dwarf_get_section_info_by_index()	243
6.34	Utility Operations	243
6.34.1	dwarf_errno()	243
6.34.2	dwarf_errmsg()	243
6.34.3	dwarf_errmsg_by_number()	244
6.34.4	dwarf_get_endian_copy_function()	244
6.34.5	dwarf_get_harmless_error_list()	244
6.34.6	dwarf_insert_harmless_error()	245
6.34.7	dwarf_set_harmless_error_list_size()	246
6.34.8	dwarf_dealloc()	249
6.34.9	dwarf_encode_leb128()	249
6.34.10	dwarf_encode_signed_leb128()	249
6.35	Finding Memory Leaks	250
6.35.1	Compiling libdwarf -DDEBUG=1	250
6.35.2	Making use of the output of -DDEBUG=1	250



## LIST OF FIGURES

Figure 1.	Scalar Types .....	9
Figure 2.	Error Indications .....	23
Figure 3.	Example_dwarf_dealloc() .....	25
Figure 4.	Example_dwarf_dealloc_die() .....	25
Figure 5.	Example_dwarf_dealloc_error() .....	26
Figure 6.	Example_dwarf_dealloc_attribute() .....	26
Figure 7.	Example1 dwarf_attrlist() .....	27
Figure 8.	Allocation/Deallocation Identifiers .....	28
Figure 9.	Example2 dwarf_set_died_dbg() .....	35
Figure 10.	Example3 dwarf_set_tied_dbg() obsolete .....	35
Figure 11.	Example4 dwarf_siblingof() .....	53
Figure 12.	Example5 dwarf_child() .....	54
Figure 13.	Example6 dwarf_offdie_b() .....	55
Figure 14.	Example7 dwarf_CU_dieoffset_given_die() .....	60
Figure 15.	Example8 dwarf_attrlist() free .....	64
Figure 16.	Exampleoffset_list dwarf_offset_list() free .....	68
Figure 17.	Example Raw Loclist .....	82
Figure 18.	Examplea dwarf_loclist() .....	99
Figure 19.	Exampleb dwarf_loclist_from_expr() .....	100
Figure 20.	Examplec dwarf_srclines_b() .....	104
Figure 21.	Exampled dwarf_srclines() .....	112
Figure 22.	Exampled dwarf_srcfiles() .....	114
Figure 23.	Exampled dwarf_get_globals() .....	121
Figure 24.	Exampled dwarf_get_pubtypes() .....	125
Figure 25.	Exampleh dwarf_get_weakes() .....	128

Figure 26.	Examplei dwarf_get_weak() obsolete .....	128
Figure 27.	Examplej dwarf_get_funcs() .....	131
Figure 28.	Examplek dwarf_get_funcs() obsolete .....	132
Figure 29.	Examplel dwarf_get_types() .....	134
Figure 30.	Examplel dwarf_get_types() obsolete .....	135
Figure 31.	Examplen dwarf_get_vars() .....	137
Figure 32.	Exampleo dwarf_get_vars() obsolete .....	138
Figure 33.	Examplep5 dwarf_dealloc_macro_context() .....	147
Figure 34.	Examplep2 dwarf_get_macro_details() .....	157
Figure 35.	Frame Information Rule Assignments MIPS .....	161
Figure 36.	Frame Information Special Values any architecture .....	161
Figure 37.	Exampleq dwarf_get_fde_list() .....	163
Figure 38.	Exampleqb dwarf_get_fde_list() obsolete .....	164
Figure 39.	Example r dwarf_get_fde_list_eh() .....	166
Figure 40.	Examples dwarf_expand_frame_instructions() .....	181
Figure 41.	Examples dwarf_get_loclist_entry() .....	183
Figure 42.	examplestringoffsets dwarf_open_str_offsets_table_access() etc .....	189
Figure 43.	Exampleu dwarf_get_aranges() .....	195
Figure 44.	Example .debug_rnglist for attribute .....	200
Figure 45.	Examplev dwarf_get_ranges_a() .....	203
Figure 46.	Examplev dwarf_get_ranges_a() .....	210
Figure 47.	Examplew dwarf_get_gdbindex_header() .....	213
Figure 48.	Examplew gdbindex dwarf_gdbindex_addressarea() .....	218

Figure 49.	Example dwarf_gdbindex_symboltable_array() .....	220
Figure 50.	Example debuglink () .....	226
Figure 51.	Example dwarf_get_xu_index_header() .....	232
Figure 52.	Example dwarf_get_xu_hash_entry() .....	235
Figure 53.	Example dwarf_get_xu_section_names() .....	238
Figure 54.	Example dwarf_get_TAG_name() .....	239
Figure 55.	Dwarf Error Codes .....	247
Figure 56.	Dwarf 2 and later Error Codes .....	248

# **A Consumer Library Interface to DWARF**

*David Anderson*

## *ABSTRACT*

This document describes an interface to a library of functions to access DWARF debugging information entries, DWARF line number information, and other DWARF2/3/4/5 information).

There are a few sections which are SGI-specific (those are clearly identified in the document).

Rev 2.97, 13 June 2020