n."
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 2007-2011 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 for a more complete description of these entries.

This document adopts all the terms and definitions in "*DWARF Debugging Information Format*" versions 2,3, and 4. 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 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

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

October 2011   DWARF4 support for reading .debug_types 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.

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.

2.3.1 Location Record

The Dwarf_Loc type identifies a single atom of a location description or a location expression.

```
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

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

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_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 for DWARF3. A separate interface usable for DWARF3 and for DWARF2 is described below. This interface is deprecated. Use the interface for DWARF3 and DWARF2. 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` figure in the `dwarf` document. If not used with the Op it is 0.

`fp_offset` is the address, delta, offset, or second register as defined in the `Call Frame Instruction Encodings` figure in the `dwarf` document. 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 larger of Dwarf_Signed and Dwarf_Addr for this to work properly. If not used with the op it is 0.

`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. 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() 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 (and DWARF2)

This interface is adequate for DWARF3 and for DWARF2 (and DWARF4). It is new in libdwarf in April 2006. See also the section "Low Level Frame Operations" below.

The DWARF2 `Dwarf_Frame_Op3` 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;

        /* Value may be signed, depends on op.
           Any applicable data_alignment_factor has
           not been applied, this is the  raw offset. */
        Dwarf_Unsigned  fp_offset_or_block_len;
        Dwarf_Small     *fp_expr_block;

        Dwarf_Off       fp_instr_offset;
} Dwarf_Frame_Op3;
```

`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` figure in the `dwarf` document. If not used with the Op it is 0.

`fp_offset_or_block_len` is the address, delta, offset, or second register as defined in the `Call Frame Instruction Encodings` figure in the `dwarf` document. Or (depending on the op, it may be the length of the dwarf-expression block pointed to by `fp_expr_block`. 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 larger of Dwarf_Signed and Dwarf_Addr for this to work properly. If not used with the op it is 0.

`fp_expr_block` (if applicable to the op) points to a dwarf-expression block which is `fp_offset_or_block_len` bytes long.

`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.

2.3.7 Frame Regtable: DWARF 3

This interface is adequate for DWARF3 and for DWARF2. 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() 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.macinfo section.

```
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_macinfo 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_MACINFO_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_MACINFO_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_MACINFO_define` this is the macro name and value. For a `DW_MACINFO_undef`, or this is the macro name. For a `DW_MACINFO_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.

```
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()`, or `dwarf_elf_init()`, 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.

```
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_weaks()` 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.

`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.

`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.

3. 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.

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()`). 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()`). 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 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 terminate execution by 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()` at initialization, call `errhand()` passing it the error descriptor and the value of the `errarg` argument provided to `dwarf_init()`. If the error handling function returns, return a value indicating an error condition.

3. 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.

3.1 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
DW_DLV_ERROR	1	Error
DW_DLV_OK	0	Successful call
DW_DLV_NO_ENTRY	-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.

4. Memory Management

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

4.1 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.

4.2 Storage Deallocation

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

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 the 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);
```

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:

```
Dwarf_Unsigned atcnt;
Dwarf_Attribute *atlist;
int errv;

errv = dwarf_attrlist(somedie, &atlist,&atcnt, &error);
if (errv == DW_DLV_OK) {

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

The `Dwarf_Debug` returned from `dwarf_init()` or `dwarf_elf_init()` 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()` or `dwarf_elf_init()` 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()` or `dwarf_elf_init()`.

The codes that identify the storage pointed to in calls to `dwarf_dealloc()` are described in figure 3.

IDENTIFIER	USED TO FREE
DW_DLA_STRING	char*
DW_DLA_LOC	Dwarf_Loc
DW_DLA_LOCDESC	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_Pubtype

Figure 3. Allocation/Deallocation Identifiers

5. 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.

5.1 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.

5.1.1 dwarf_init()

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

When it returns `DW_DLV_OK`, the function `dwarf_init()` 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 `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()`.

With SGI IRIX, by default it is allowed that the app `close()` `fd` immediately after calling `dwarf_init()`, but that is not a portable approach (that it works is an accidental side effect of the fact that SGI IRIX uses `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.

5.1.2 dwarf_elf_init()

```
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()` except that an open `Elf *` pointer is passed instead of a file descriptor. In systems supporting `ELF` object files this may be more space or time-efficient than using `dwarf_init()`. 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.

5.1.3 dwarf_get_elf()

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

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.

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()`.

This function is not meaningful for a system that does not use the Elf format for objects.

5.1.4 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()`.

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

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

5.1.6 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.

5.1.7 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.

5.2 Section size operations

These operations are informative but not normally needed.

5.2.1 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.

5.2.2 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.

5.3 Debugging Information Entry Delivery Operations

These functions are concerned with accessing debugging information entries.

5.3.1 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 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. 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 DWARF4, or 4 for DWARF4.

`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 is_info is zero, 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 if `is_info` is zero, and 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.

5.3.2 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()`

5.3.3 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);
```

5.3.4 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.

```
    Dwarf_Die return_sib = 0;
    Dwarf_Error error = 0;
    int res;
    Dwarf_Bool is_info = 1;
    /* 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(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;
    }
```

5.3.5 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.

5.3.6 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`.

```
   Dwarf_Die return_kid = 0;
   Dwarf_Error error = 0;
   int res;

   res = dwarf_child(dbg,in_die,&return_kid, &error);
   if (res == DW_DLV_OK) {
      /* Use return_kid here. */
      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;
   }
```

5.3.7 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.

```
   Dwarf_Error error = 0;
   Dwarf_Die return_die = 0;
   int res;

   res = dwarf_offdie_b(dbg,die_offset,&return_die, &error);
   if (res == DW_DLV_OK) {
      /* Use return_die here. */
      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;
   }
```

5.3.8 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.

5.3.9 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.

5.4 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.

5.4.1 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.

5.4.2 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.

5.4.3 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.

5.4.4 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).

5.4.5 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.

5.4.6 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.

5.4.7 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 that, for some reason, you have decided needs CU context printed (assuming print_die_data does some reasonable printing).

```
    int res;
    Dwarf_Off cudieoff = 0;
    Dwarf_Die cudie = 0;

    print_die_data(dbg,in_die);
    res = dwarf_CU_dieoffset_given_die(in_die,&cudieoff,&error);
    if(res != DW_DLV_OK) {
       printf("FAIL: dwarf_CU_dieoffset_given_die did not work0);
       exit(1);
    }
    res = dwarf_offdie_b(dbg,cudieoff,&cudie,&error);
    if(res != DW_DLV_OK) {
       printf("FAIL: dwarf_offdie did not work0);
       exit(1);
    }
    print_die_data(dbg,cudie);
    dwarf_dealloc(dbg,cudie, DW_DLA_DIE);
```

5.4.8 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.

5.4.9 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 of `die`. It returns `DW_DLV_NO_ENTRY` if `die` does not have a name attribute. It returns `DW_DLV_ERROR` if an error occurred. The storage pointed to by a successful return of `dwarf_diename()` should be freed using the allocation type `DW_DLA_STRING` when no longer of interest (see `dwarf_dealloc()`).

5.4.10 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.

5.4.11 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.

5.4.12 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.

5.4.13 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:

```
    Dwarf_Unsigned atcnt;
    Dwarf_Attribute *atlist;
    int errv;

    errv = dwarf_attrlist(somedie, &atlist,&atcnt, &error);
    if (errv == DW_DLV_OK) {

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

5.4.14 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`.

5.4.15 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.

5.4.16 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.

5.4.17 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).

So this function is useless for a `DW_AT_high_pc` which is encoded as a constant.

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 of class constant (which means this function is not very useful for some DWARF as of 2013).

5.4.18 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.

5.4.19 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.

5.4.20 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.

5.4.21 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.

5.4.22 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.

5.5 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.

5.5.1 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 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.

5.5.2 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.

5.5.3 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`.

5.5.4 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.

5.5.5 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.

5.5.6 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.

5.5.7 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.

5.5.8 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.

5.5.9 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.

5.5.10 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`.

5.5.11 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.

5.5.12 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.

5.5.13 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.

5.5.14 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.

5.5.15 dwarf_formsig8()

```
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_experloc`. 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.

5.5.16 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()`.

5.5.17 dwarf_loclist_n()

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

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.

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_LOCDESC`. This should be followed by deallocation of the `llbuf` using the allocation type `DW_DLA_LIST`.

```
    Dwarf_Signed lcnt;
    Dwarf_Locdesc **llbuf;
    int lres;

    lres = dwarf_loclist_n(someattr, &llbuf,&lcnt &error);
    if (lres == DW_DLV_OK) {
            for (i = 0; i < lcnt; ++i) {
                /* use llbuf[i] */

                dwarf_dealloc(dbg, llbuf[i]->ld_s, DW_DLA_LOC_BLOCK);
                dwarf_dealloc(dbg,llbuf[i], DW_DLA_LOCDESC);
            }
            dwarf_dealloc(dbg, llbuf, DW_DLA_LIST);
    }
```

5.5.18 dwarf_loclist()

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

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_LOCDESC.

```
    Dwarf_Signed lcnt;
    Dwarf_Locdesc *llbuf;
    int lres;

    lres = dwarf_loclist(someattr, &llbuf,&lcnt,&error);
    if (lres == DW_DLV_OK) {
            /* lcnt 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_LOCDESC);
/*          Earlier version.
 *            for (i = 0; i < lcnt; ++i) {
 *                  /* use llbuf[i] */
 *
 *                  /* Deallocate Dwarf_Loc block of llbuf[i] */
 *                  dwarf_dealloc(dbg, llbuf[i].ld_s, DW_DLA_LOC_BLOCK);
 *            }
 *            dwarf_dealloc(dbg, llbuf, DW_DLA_LOCDESC);
 */

    }
```

5.5.19 dwarf_loclist_from_expr()

```
int dwarf_loclist_from_expr(
        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`.

```
    Dwarf_Signed lcnt;
    Dwarf_Locdesc *llbuf;
    int lres;
    /* Example with an empty buffer here. */
    Dwarf_Ptr data = "";
    Dwarf_Unsigned len = 0;

    lres = dwarf_loclist_from_expr(data,len, &llbuf,&lcnt, &error);
    if (lres == DW_DLV_OK) {
            /* lcnt is always 1 */

            /* Use llbuf  here.*/

            dwarf_dealloc(dbg, llbuf->ld_s, DW_DLA_LOC_BLOCK);
            dwarf_dealloc(dbg, llbuf, DW_DLA_LOCDESC);

    }
```

5.5.20 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)
```

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.

5.5.21 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)
```

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.

5.6 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.

5.6.1 Get A Set of Lines

The function returns information about every source line for a particular compilation-unit. The compilation-unit is specified by the corresponding die.

5.6.1.1 dwarf_srclines()

```
int dwarf_srclines(
        Dwarf_Die die,
        Dwarf_Line **linebuf,
        Dwarf_Signed *linecount,
        Dwarf_Error *error)
```

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. 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.

```
    Dwarf_Signed cnt;
    Dwarf_Line *linebuf;
    int sres;

    sres = dwarf_srclines(somedie, &linebuf,&cnt, &error);
    if (sres == DW_DLV_OK) {
            for (i = 0; i < cnt; ++i) {
                    /* use linebuf[i] */
            }
            dwarf_srclines_dealloc(dbg, linebuf, cnt);
    }
```

The following dealloc code (the only documented method before July 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.

```
    Dwarf_Signed cnt;
    Dwarf_Line *linebuf;
    int sres;

    sres = dwarf_srclines(somedie, &linebuf,&cnt, &error);
    if (sres == DW_DLV_OK) {
            for (i = 0; i < cnt; ++i) {
                    /* use linebuf[i] */
                    dwarf_dealloc(dbg, linebuf[i], DW_DLA_LINE);
            }
            dwarf_dealloc(dbg, linebuf, DW_DLA_LIST);
    }
```

5.6.2 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 are returned.

```
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 this function, 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).

```
    Dwarf_Signed cnt;
    char **srcfiles;
    int res;

    res = dwarf_srcfiles(somedie, &srcfiles,&cnt &error);
    if (res == DW_DLV_OK) {

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

5.6.3 Get information about a Single Table Line

The following functions can be used on the Dwarf_Line descriptors returned by dwarf_srclines()
to obtain information about the source lines.

5.6.3.1 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.

5.6.3.2 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.

5.6.3.3 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`.

5.6.3.4 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`.

5.6.3.5 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`.

5.6.3.6 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 specication (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`.

5.6.3.7 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.

5.6.3.8 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.

5.6.3.9 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.

5.6.3.10 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.

5.7 Global Name Space Operations

These operations operate on the .debug_pubnames section of the debugging information.

5.8 Global Name Space Operations

These operations operate on the .debug_pubnames section of the debugging information.

5.8.1 Debugger Interface Operations

5.8.1.1 dwarf_get_globals()

```
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..

```
    Dwarf_Signed cnt;
    Dwarf_Global *globs;
    int res;

    res = dwarf_get_globals(dbg, &globs,&cnt, &error);
    if (res == DW_DLV_OK) {

            for (i = 0; i < cnt; ++i) {
                    /* use globs[i] */
            }
            dwarf_globals_dealloc(dbg, globs, cnt);
    }
```

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) {

            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);
    }
```

5.8.1.2 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.

5.8.1.3 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.

5.8.1.4 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.

5.8.1.5 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.

5.8.1.6 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 thought 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 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.

5.8.1.7 dwarf_global_name_offsets()

```
int dwarf_global_name_offsets(
        Dwarf_Global global,
        char      **return_name,
        Dwarf_Off *die_offset,
        Dwarf_Off *cu_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 offsets of the DIE representing the pubname, and 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.

5.9 DWARF3 Type Names Operations

Section ".debug_pubtypes" is new in DWARF3.

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 `DIE`s that represent the definitions of those types, and the offsets of the compilation-units that contain the definitions of those types.

5.9.1 Debugger Interface Operations

5.9.1.1 dwarf_get_pubtypes()

```
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_Pubtype` 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.

```
    Dwarf_Signed cnt;
    Dwarf_Pubtype *types;
    int res;

    res = dwarf_get_pubtypes(dbg, &types,&cnt, &error);
    if (res == DW_DLV_OK) {

            for (i = 0; i < cnt; ++i) {
                    /* use types[i] */
            }
            dwarf_types_dealloc(dbg, types, cnt);
    }
```

5.9.1.2 dwarf_pubtypename()

```
int dwarf_pubtypename(
        Dwarf_Pubtype   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_Pubtype` 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.

5.9.1.3 dwarf_pubtype_die_offset()

```
int dwarf_pubtype_die_offset(
        Dwarf_Pubtype type,
        Dwarf_Off  *return_offset,
        Dwarf_Error *error)
```

The function `dwarf_pubtype_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_Pubtype` descriptor, `type`. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

5.9.1.4 dwarf_pubtype_cu_offset()

```
int dwarf_pubtype_cu_offset(
        Dwarf_Pubtype 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_Pubtype` descriptor, `type`. It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

5.9.1.5 dwarf_pubtype_name_offsets()

```
int dwarf_pubtype_name_offsets(
        Dwarf_Pubtype   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_Pubtype` 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.

5.10 User Defined Static Variable Names Operations

This section is SGI specific and is not part of standard DWARF version 2.

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.

5.11 Weak Name Space Operations

These operations operate on the .debug_weaknames section of the debugging information.

These operations are SGI specific, not part of standard DWARF.

5.11.1 Debugger Interface Operations

5.11.1.1 dwarf_get_weaks()

```
int dwarf_get_weaks(
        Dwarf_Debug dbg,
        Dwarf_Weak **weaks,
        Dwarf_Signed *weak_count,
        Dwarf_Error *error)
```

The function dwarf_get_weaks() 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 *weaks, 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_weaks_dealloc() when the data is no longer of interest. dwarf_weaks_dealloc()is new as of July 15, 2005.

```
    Dwarf_Signed cnt;
    Dwarf_Weak *weaks;
    int res;

    res = dwarf_get_weaks(dbg, &weaks, &cnt, &error);
    if (res == DW_DLV_OK) {

            for (i = 0; i < cnt; ++i) {
                    /* use weaks[i] */
            }
            dwarf_weaks_dealloc(dbg, weaks, cnt);
    }
```

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_weaks()` 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.

```
    Dwarf_Signed cnt;
    Dwarf_Weak *weaks;
    int res;

    res = dwarf_get_weaks(dbg, &weaks, &cnt, &error);
    if (res == DW_DLV_OK) {

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

5.11.1.2 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`.

5.11.1.3 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`.

5.11.1.4 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.

5.12 Static Function Names Operations

This section is SGI specific and is not part of standard DWARF version 2.

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.

5.12.1 Debugger Interface Operations

5.12.1.1 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.

```
    Dwarf_Signed cnt;
    Dwarf_Func *funcs;
    int fres;

    fres = dwarf_get_funcs(dbg, &funcs, &cnt, &error);
    if (fres == DW_DLV_OK) {

            for (i = 0; i < cnt; ++i) {
                    /* use funcs[i] */
            }
            dwarf_funcs_dealloc(dbg, funcs, cnt);
    }
```

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.

```
    Dwarf_Signed cnt;
    Dwarf_Func *funcs;
    int fres;

    fres = dwarf_get_funcs(dbg, &funcs, &error);
    if (fres == DW_DLV_OK) {

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

5.12.1.2 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.

5.12.1.3 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`.

5.12.1.4 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`.

5.12.1.5 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.

5.13 User Defined Type Names Operations

Section "debug_typenames" is SGI specific and is not part of standard DWARF version 2. (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_typenames section contains the names of file-scope user-defined types, the offsets of the `DIE`s that represent the definitions of those types, and the offsets of the compilation-units that contain the definitions of those types.

5.13.1 Debugger Interface Operations

5.13.1.1 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_typenames. 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_typenames 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_typenames 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().

```
    Dwarf_Signed cnt;
    Dwarf_Type *types;
    int res;

    res = dwarf_get_types(dbg, &types,&cnt, &error);
    if (res == DW_DLV_OK) {

            for (i = 0; i < cnt; ++i) {
                    /* use types[i] */
            }
            dwarf_types_dealloc(dbg, types, cnt);
    }
```

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_TYPENAME_CONTEXT, (or DW_DLA_TYPENAME, 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_Type *types;
    int res;

    res = dwarf_get_types(dbg, &types,&cnt, &error);
    if (res == DW_DLV_OK) {

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

5.13.1.2 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.

5.13.1.3 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.

5.13.1.4 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.

5.13.1.5 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.

5.14 User Defined Static Variable Names Operations

This section is SGI specific and is not part of standard DWARF version 2.

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 `DIE`s that represent the definitions of those variables, and the offsets of the compilation-units that contain the definitions of those variables.

5.14.1 Debugger Interface Operations

5.14.1.1 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()`.

```
    Dwarf_Signed cnt;
    Dwarf_Var *vars;
    int res;

    res = dwarf_get_vars(dbg, &vars,&cnt &error);
    if (res == DW_DLV_OK) {

            for (i = 0; i < cnt; ++i) {
                    /* use vars[i] */
            }
            dwarf_vars_dealloc(dbg, vars, cnt);
    }
```

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.

```
    Dwarf_Signed cnt;
    Dwarf_Var *vars;
    int res;

    res = dwarf_get_vars(dbg, &vars,&cnt &error);
    if (res == DW_DLV_OK) {

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

5.14.1.2 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.

5.14.1.3 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.

5.14.1.4 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.

5.14.1.5 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 file-scope static variable, and the DIE 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.

5.15 Macro Information Operations

5.15.1 General Macro Operations

5.15.1.1 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.

5.15.2 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.

5.15.3 Low Level Macro Information Operations

5.15.3.1 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.

```
    Dwarf_Unsigned max = 0;
    Dwarf_Off cur_off = 0;
    Dwarf_Signed count = 0;
    Dwarf_Macro_Details *maclist;
    int errv;

    /* Loop through all the compilation units macro info.
       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 usually work.
    */
    while((errv = dwarf_get_macro_details(dbg, cur_off,max,
         &count,&maclist,&error))== DW_DLV_OK) {
      for (i = 0; i < count; ++i) {
        /* use maclist[i] */
      }
      cur_off = maclist[count-1].dmd_offset + 1;
      dwarf_dealloc(dbg, maclist, DW_DLA_STRING);
    }
```

5.16 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 3 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 4. 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 5. 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').

5.16.0.1 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.

```
    Dwarf_Signed cnt;
    Dwarf_Cie *cie_data;
    Dwarf_Signed cie_count;
    Dwarf_Fde *fde_data;
    Dwarf_Signed fde_count;
    int fres;

    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.

```
    Dwarf_Signed cnt;
    Dwarf_Cie *cie_data;
    Dwarf_Signed cie_count;
    Dwarf_Fde *fde_data;
    Dwarf_Signed fde_count;
    int fres;

    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);
    }
```

5.16.0.2 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.

```
    Dwarf_Signed cnt;
    Dwarf_Cie *cie_data;
    Dwarf_Signed cie_count;
    Dwarf_Fde *fde_data;
    Dwarf_Signed fde_count;
    int fres;

    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);
    }
```

5.16.0.3 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 and does not wish to dwarf_dealloc() all the individual FDEs immediately, one must also avoid dwarf_dealloc-ing the CIEs for those FDEs not immediately dealloc'd. 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.

5.16.0.4 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, is 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.

5.16.0.5 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.

5.16.0.6 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.

5.16.0.7 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.

5.16.0.8 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.

5.16.0.9 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.

5.16.0.10 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.

5.16.0.11 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).

5.16.0.12 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_inital_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).

5.16.0.13 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).

5.16.0.14 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).

5.16.0.15 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).

5.16.0.16 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).

5.16.0.17 dwarf_get_fde_info_for_reg3()

This interface is suitable for DWARF3 and DWARF2. It returns the values for a particular real register (Not for the CFA 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);
```

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().

5.16.0.18 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).

5.16.0.19 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.

5.16.0.20 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()`.

5.16.0.21 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().

5.16.0.22 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.

```
    Dwarf_Signed cnt;
    Dwarf_Frame_Op *frameops;
    Dwarf_Ptr instruction;
    Dwarf_Unsigned len;
    int res;

    res = expand_frame_instructions(dbg,instruction,len, &frameops,&cnt, &error);
    if (res == DW_DLV_OK) {
        for (i = 0; i < cnt; ++i) {
            /* use frameops[i] */
        }
        dwarf_dealloc(dbg, frameops, DW_DLA_FRAME_BLOCK);
    }
```

5.16.0.23 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.

5.17 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.

5.17.1 Location List Internal-level Interface

5.17.1.1 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)
```

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 usually work correctly (meaning with most objects) 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.

```
    /* Looping through the dwarf_loc section finding loclists:
       an example.  */
    int res;
    Dwarf_Unsigned next_entry;
    Dwarf_unsigned offset=0;
    Dwarf_Addr hipc_off;
    Dwarf_Addr lopc_off;
    Dwarf_Ptr data;
    Dwarf_Unsigned entry_len;
    Dwarf_Unsigned next_entry;
    Dwarf_Error err;

        for(;;) {
            res = dwarf_get_loclist_entry(dbg,newoffset,&hipc_off,
                &lowpc_off, &data, &entry_len,&next_entry,&err);
            if (res == DW_DLV_OK) {
                /* A valid entry. */
                newoffset = next_entry;
                continue;
            } else if (res ==DW_DLV_NO_ENTRY) {
                /* Done! */
                break;
            } else {
                /* Error! */
                break;
            }


        }
    }
```

5.18 Abbreviations access

These are Internal-level Interface functions. Debuggers can ignore this.

5.18.1 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.

5.18.2 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`.

5.18.3 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`.

5.18.4 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.

5.18.5 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)
```

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.

5.19 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.

5.19.1 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.

5.20 Address Range Operations

These functions provide information about address ranges. Address ranges map ranges of pc values to the corresponding compilation-unit die that covers the address range.

5.20.1 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.

```
Dwarf_Signed cnt;
Dwarf_Arange *arang;
int res;

res = dwarf_get_aranges(dbg, &arang,&cnt, &error);
if (res == DW_DLV_OK) {

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

5.20.2 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.

5.20.3 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.

5.20.4 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.

5.20.5 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)
```

The function `dwarf_get_arange_info()` returns `DW_DLV_OK` and stores 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.

5.21 General Low Level Operations

This function is low-level and intended for use only by programs such as dwarf-dumpers.

5.21.1 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.

5.21.2 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.

5.22 Ranges Operations (.debug_ranges)

5.23 Ranges Operations (.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. 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`.

5.23.1 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).

5.23.2 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.

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.

```
    Dwarf_Signed cnt;
    Dwarf_Ranges *ranges;
    Dwarf_Unsigned bytes;
    int res;
    res = dwarf_get_ranges_a(dbg,off,dieptr, &ranges,&cnt,&bytes,&error);
    if (res == DW_DLV_OK) {
            Dwarf_Signed i;
            for( i = 0; i < cnt; ++i ) {
                Dwarf_Ranges *cur = ranges+i;
                /* Use cur. */
            }
            dwarf_ranges_dealloc(dbg,ranges,cnt);
    }
```

5.23.3 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.

5.24 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:

```
const char * out;
int res;
/* 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. */
}
```

5.25 dwarf_get_ACCESS_name()

Returns an accessibility code name through the s_out pointer.

5.26 dwarf_get_AT_name()

Returns an attribute code name through the s_out pointer.

5.27 dwarf_get_ATE_name()

Returns a base type encoding name through the s_out pointer.

5.28 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.

5.29 dwarf_get_ATCF_name()

Returns a SUN code flag encoding name through the s_out pointer. This code flag is entirely a DWARF extension.

5.30 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).

5.31 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).

5.32 dwarf_get_CC_name()

Returns a calling convention case code name through the s_out pointer.

5.33 dwarf_get_CFA_name()

Returns a call frame information instruction name through the s_out pointer.

5.34 dwarf_get_DS_name()

Returns a decimal sign code name through the s_out pointer.

5.35 dwarf_get_DSC_name()

Returns a discriminant descriptor code name through the s_out pointer.

5.36 dwarf_get_EH_name()

Returns a GNU exception header code name through the s_out pointer.

5.37 dwarf_get_END_name()

Returns an endian code name through the s_out pointer.

5.38 dwarf_get_FORM_name()

Returns an form code name through the s_out pointer.

5.39 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.

5.40 dwarf_get_ID_name()

Returns an identifier case code name through the s_out pointer.

5.41 dwarf_get_INL_name()

Returns an inline code name through the s_out pointer.

5.42 dwarf_get_LANG_name()

Returns a language code name through the s_out pointer.

5.43 dwarf_get_LNE_name()

Returns a line table extended opcode code name through the s_out pointer.

5.44 dwarf_get_LNS_name()

Returns a line table standard opcode code name through the s_out pointer.

5.45 dwarf_get_MACINFO_name()

Returns a macro information macinfo code name through the s_out pointer.

5.46 dwarf_get_OP_name()

Returns a DWARF expression operation code name through the s_out pointer.

5.47 dwarf_get_ORD_name()

Returns an array ordering code name through the `s_out` pointer.

5.48 dwarf_get_TAG_name()

Returns a TAG name through the `s_out` pointer.

5.49 dwarf_get_VIRTUALITY_name()

Returns a virtuality code name through the `s_out` pointer.

5.50 dwarf_get_VIS_name()

Returns a visibility code name through the `s_out` pointer.

5.51 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.

5.51.1 dwarf_get_section_count()

```
int dwarf_get_section_count(
    Dwarf_Debug dbg)
```

Returns a count of the number of object sections found.

5.51.2 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.

5.51.3 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.

5.52 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.

5.52.1 dwarf_errno()

```
Dwarf_Unsigned dwarf_errno(
        Dwarf_Error error)
```

The function `dwarf_errno()` returns the error number corresponding to the error specified by `error`.

5.52.2 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 returned by `dwarf_errmsg()` should not be deallocated using `dwarf_dealloc()`.

5.52.3 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 MININUM(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.

5.52.4 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.

5.52.5 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 3 below were defined in Dwarf 1. These errors are not used by the `libdwarf` implementation for Dwarf 2 or later.

SYMBOLIC NAME	DESCRIPTION
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 6. List of Dwarf Error Codes

The set of errors returned by Libdwarf functions is listed below. Some of the errors are SGI specific.

SYMBOLIC NAME	DESCRIPTION
DW_DLE_DBG_ALLOC	Could not allocate Dwarf_Debug struct
DW_DLE_FSTAT_ERROR	Error in fstat()-ing object
DW_DLE_FSTAT_MODE_ERROR	Error in mode of object file
DW_DLE_INIT_ACCESS_WRONG	Incorrect access to dwarf_init()
DW_DLE_ELF_BEGIN_ERROR	Error in elf_begin() on object
DW_DLE_ELF_GETEHDR_ERROR	Error in elf_getehdr() on object
DW_DLE_ELF_GETSHDR_ERROR	Error in elf_getshdr() on object
DW_DLE_ELF_STRPTR_ERROR	Error in elf_strptr() on object
DW_DLE_DEBUG_INFO_DUPLICATE	Multiple .debug_info sections
DW_DLE_DEBUG_INFO_NULL	No data in .debug_info section
DW_DLE_DEBUG_ABBREV_DUPLICATE	Multiple .debug_abbrev sections
DW_DLE_DEBUG_ABBREV_NULL	No data in .debug_abbrev section
DW_DLE_DEBUG_ARANGES_DUPLICATE	Multiple .debug_arange sections
DW_DLE_DEBUG_ARANGES_NULL	No data in .debug_arange section
DW_DLE_DEBUG_LINE_DUPLICATE	Multiple .debug_line sections
DW_DLE_DEBUG_LINE_NULL	No data in .debug_line section
DW_DLE_DEBUG_LOC_DUPLICATE	Multiple .debug_loc sections
DW_DLE_DEBUG_LOC_NULL	No data in .debug_loc section
DW_DLE_DEBUG_MACINFO_DUPLICATE	Multiple .debug_macinfo sections
DW_DLE_DEBUG_MACINFO_NULL	No data in .debug_macinfo section
DW_DLE_DEBUG_PUBNAMES_DUPLICATE	Multiple .debug_pubnames sections
DW_DLE_DEBUG_PUBNAMES_NULL	No data in .debug_pubnames section
DW_DLE_DEBUG_STR_DUPLICATE	Multiple .debug_str sections
DW_DLE_DEBUG_STR_NULL	No data in .debug_str section
DW_DLE_CU_LENGTH_ERROR	Length of compilation-unit bad
DW_DLE_VERSION_STAMP_ERROR	Incorrect Version Stamp
DW_DLE_ABBREV_OFFSET_ERROR	Offset in .debug_abbrev bad
DW_DLE_ADDRESS_SIZE_ERROR	Size of addresses in target bad
DW_DLE_DEBUG_INFO_PTR_NULL	Pointer into .debug_info in DIE null
DW_DLE_DIE_NULL	Null Dwarf_Die
DW_DLE_STRING_OFFSET_BAD	Offset in .debug_str bad
DW_DLE_DEBUG_LINE_LENGTH_BAD	Length of .debug_line segment bad
DW_DLE_LINE_PROLOG_LENGTH_BAD	Length of .debug_line prolog bad
DW_DLE_LINE_NUM_OPERANDS_BAD	Number of operands to line instr bad
DW_DLE_LINE_SET_ADDR_ERROR	Error in DW_LNE_set_address instruction
DW_DLE_LINE_EXT_OPCODE_BAD	Error in DW_EXTENDED_OPCODE instruction
DW_DLE_DWARF_LINE_NULL	Null Dwarf_line argument
DW_DLE_INCL_DIR_NUM_BAD	Error in included directory for given line
DW_DLE_LINE_FILE_NUM_BAD	File number in .debug_line bad
DW_DLE_ALLOC_FAIL	Failed to allocate required structs
DW_DLE_DBG_NULL	Null Dwarf_Debug argument
DW_DLE_DEBUG_FRAME_LENGTH_BAD	Error in length of frame
DW_DLE_FRAME_VERSION_BAD	Bad version stamp for frame
DW_DLE_CIE_RET_ADDR_REG_ERROR	Bad register specified for return address
DW_DLE_FDE_NULL	Null Dwarf_Fde argument
DW_DLE_FDE_DBG_NULL	No Dwarf_Debug associated with FDE
DW_DLE_CIE_NULL	Null Dwarf_Cie argument
DW_DLE_CIE_DBG_NULL	No Dwarf_Debug associated with CIE
DW_DLE_FRAME_TABLE_COL_BAD	Bad column in frame table specified

Figure 7. List of Dwarf 2 Error Codes (continued)

SYMBOLIC NAME	DESCRIPTION
DW_DLE_PC_NOT_IN_FDE_RANGE	PC requested not in address range of FDE
DW_DLE_CIE_INSTR_EXEC_ERROR	Error in executing instructions in CIE
DW_DLE_FRAME_INSTR_EXEC_ERROR	Error in executing instructions in FDE
DW_DLE_FDE_PTR_NULL	Null Pointer to Dwarf_Fde specified
DW_DLE_RET_OP_LIST_NULL	No location to store pointer to Dwarf_Frame_Op
DW_DLE_LINE_CONTEXT_NULL	Dwarf_Line has no context
DW_DLE_DBG_NO_CU_CONTEXT	dbg has no CU context for dwarf_siblingof()
DW_DLE_DIE_NO_CU_CONTEXT	Dwarf_Die has no CU context
DW_DLE_FIRST_DIE_NOT_CU	First DIE in CU not DW_TAG_compilation_unit
DW_DLE_NEXT_DIE_PTR_NULL	Error in moving to next DIE in .debug_info
DW_DLE_DEBUG_FRAME_DUPLICATE	Multiple .debug_frame sections
DW_DLE_DEBUG_FRAME_NULL	No data in .debug_frame section
DW_DLE_ABBREV_DECODE_ERROR	Error in decoding abbreviation
DW_DLE_DWARF_ABBREV_NULL	Null Dwarf_Abbrev specified
DW_DLE_ATTR_NULL	Null Dwarf_Attribute specified
DW_DLE_DIE_BAD	DIE bad
DW_DLE_DIE_ABBREV_BAD	No abbreviation found for code in DIE
DW_DLE_ATTR_FORM_BAD	Inappropriate attribute form for attribute
DW_DLE_ATTR_NO_CU_CONTEXT	No CU context for Dwarf_Attribute struct
DW_DLE_ATTR_FORM_SIZE_BAD	Size of block in attribute value bad
DW_DLE_ATTR_DBG_NULL	No Dwarf_Debug for Dwarf_Attribute struct
DW_DLE_BAD_REF_FORM	Inappropriate form for reference attribute
DW_DLE_ATTR_FORM_OFFSET_BAD	Offset reference attribute outside current CU
DW_DLE_LINE_OFFSET_BAD	Offset of lines for current CU outside .debug_line
DW_DLE_DEBUG_STR_OFFSET_BAD	Offset into .debug_str past its end
DW_DLE_STRING_PTR_NULL	Pointer to pointer into .debug_str NULL
DW_DLE_PUBNAMES_VERSION_ERROR	Version stamp of pubnames incorrect
DW_DLE_PUBNAMES_LENGTH_BAD	Read pubnames past end of .debug_pubnames
DW_DLE_GLOBAL_NULL	Null Dwarf_Global specified
DW_DLE_GLOBAL_CONTEXT_NULL	No context for Dwarf_Global given
DW_DLE_DIR_INDEX_BAD	Error in directory index read
DW_DLE_LOC_EXPR_BAD	Bad operator read for location expression
DW_DLE_DIE_LOC_EXPR_BAD	Expected block value for attribute not found
DW_DLE_OFFSET_BAD	Offset for next compilation-unit in .debug_info bad
DW_DLE_MAKE_CU_CONTEXT_FAIL	Could not make CU context
DW_DLE_ARANGE_OFFSET_BAD	Offset into .debug_info in .debug_aranges bad
DW_DLE_SEGMENT_SIZE_BAD	Segment size will be 0 for MIPS processors and should always be < 8.
DW_DLE_ARANGE_LENGTH_BAD	Length of arange section in .debug_arange bad
DW_DLE_ARANGE_DECODE_ERROR	Aranges do not end at end of .debug_aranges
DW_DLE_ARANGES_NULL	NULL pointer to Dwarf_Arange specified
DW_DLE_ARANGE_NULL	NULL Dwarf_Arange specified
DW_DLE_NO_FILE_NAME	No file name for Dwarf_Line struct
DW_DLE_NO_COMP_DIR	No Compilation directory for compilation-unit
DW_DLE_CU_ADDRESS_SIZE_BAD	CU header address size not match Elf class
DW_DLE_ELF_GETIDENT_ERROR	Error in elf_getident() on object
DW_DLE_NO_AT_MIPS_FDE	DIE does not have DW_AT_MIPS_fde attribute
DW_DLE_NO_CIE_FOR_FDE	No CIE specified for FDE
DW_DLE_DIE_ABBREV_LIST_NULL	No abbreviation for the code in DIE found
DW_DLE_DEBUG_FUNCNAMES_DUPLICATE	Multiple .debug_funcnames sections
DW_DLE_DEBUG_FUNCNAMES_NULL	No data in .debug_funcnames section

Figure 8. List of Dwarf 2 Error Codes (continued)

SYMBOLIC NAME	DESCRIPTION
DW_DLE_DEBUG_FUNCNAMES_VERSION_ERROR	Version stamp in .debug_funcnames bad
DW_DLE_DEBUG_FUNCNAMES_LENGTH_BAD	Length error in reading .debug_funcnames
DW_DLE_FUNC_NULL	NULL Dwarf_Func specified
DW_DLE_FUNC_CONTEXT_NULL	No context for Dwarf_Func struct
DW_DLE_DEBUG_TYPENAMES_DUPLICATE	Multiple .debug_typenames sections
DW_DLE_DEBUG_TYPENAMES_NULL	No data in .debug_typenames section
DW_DLE_DEBUG_TYPENAMES_VERSION_ERROR	Version stamp in .debug_typenames bad
DW_DLE_DEBUG_TYPENAMES_LENGTH_BAD	Length error in reading .debug_typenames
DW_DLE_TYPE_NULL	NULL Dwarf_Type specified
DW_DLE_TYPE_CONTEXT_NULL	No context for Dwarf_Type given
DW_DLE_DEBUG_VARNAMES_DUPLICATE	Multiple .debug_varnames sections
DW_DLE_DEBUG_VARNAMES_NULL	No data in .debug_varnames section
DW_DLE_DEBUG_VARNAMES_VERSION_ERROR	Version stamp in .debug_varnames bad
DW_DLE_DEBUG_VARNAMES_LENGTH_BAD	Length error in reading .debug_varnames
DW_DLE_VAR_NULL	NULL Dwarf_Var specified
DW_DLE_VAR_CONTEXT_NULL	No context for Dwarf_Var given
DW_DLE_DEBUG_WEAKNAMES_DUPLICATE	Multiple .debug_weaknames section
DW_DLE_DEBUG_WEAKNAMES_NULL	No data in .debug_varnames section
DW_DLE_DEBUG_WEAKNAMES_VERSION_ERROR	Version stamp in .debug_varnames bad
DW_DLE_DEBUG_WEAKNAMES_LENGTH_BAD	Length error in reading .debug_weaknames
DW_DLE_WEAK_NULL	NULL Dwarf_Weak specified
DW_DLE_WEAK_CONTEXT_NULL	No context for Dwarf_Weak given

Figure 9. List of Dwarf 2 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.

5.52.6 dwarf_seterrhand()

```
Dwarf_Handler dwarf_seterrhand(
        Dwarf_Debug dbg,
        Dwarf_Handler errhand)
```

The function dwarf_seterrhand() replaces the error handler (see dwarf_init()) with errhand. The old error handler is returned. This function is currently unimplemented.

5.52.7 dwarf_seterrarg()

```
Dwarf_Ptr dwarf_seterrarg(
        Dwarf_Debug dbg,
        Dwarf_Ptr errarg)
```

The function dwarf_seterrarg() replaces the pointer to the error handler communication area (see dwarf_init()) with errarg. A pointer to the old area is returned. This function is currently unimplemented.

5.52.8 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.

5.52.9 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`

5.52.10 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.

CONTENTS

LIST OF FIGURES

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 and DWARF line number information (and other DWARF2/3 information). It does not make recommendations as to how the functions described in this document should be implemented nor does it suggest possible optimizations.

The document is oriented to reading DWARF version 2 and version 3. There are certain sections which are SGI-specific (those are clearly identified in the document).

rev 2.11, June 07, 2013

---

0.   UNIX is a registered trademark of UNIX System Laboratories, Inc. in the United States and other countries.