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 Silcon 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

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 and 3. 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

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

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.

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.

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.

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 but not for DWARF3. A separate interface usable for DWARF3 and for DWARF2 is described 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.

```
/* 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. It is new in libdwarf in April 2006. 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 `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_type`s 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()` 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.

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.

5.2 Debugging Information Entry Delivery Operations

These functions are concerned with accessing debugging information entries.

5.2.1 Debugging Information Entry Debugger Delivery Operations

5.2.2 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);
```

The function `dwarf_next_cu_header_b()` 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 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`, or `extension_size`, 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.

5.2.3 dwarf_next_cu_header()

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

```
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);
```

5.2.4 dwarf_siblingof()

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

The function `dwarf_siblingof()` 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()` returns `DW_DLV_OK` and sets `*return_sib` to the `Dwarf_Die` descriptor of the sibling of `die`.

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;
   /* in_die might be NULL or a vaid Dwarf_Die */
   res = dwarf_siblingof(dbg,in_die,&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;
   }
```

5.2.5 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;
    }
```

5.2.6 dwarf_offdie()

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

The function `dwarf_offdie()` returns `DW_DLV_ERROR` and sets the `error` die on error. When it succeeds, `dwarf_offdie()` 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.

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

    res = dwarf_offdie(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 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.

5.3.1 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.

5.3.2 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.

5.3.3 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).

5.3.4 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()` 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(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);
```

5.3.5 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.

5.3.6 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()`).

5.3.7 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.

5.3.8 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);
}
```

5.3.9 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`.

5.3.10 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.

5.3.11 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 this attribute. It returns `DW_DLV_NO_ENTRY` if `die` does not have this attribute. It returns `DW_DLV_ERROR` if an error occurred.

5.3.12 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 this attribute. It returns `DW_DLV_NO_ENTRY` if `die` does not have this attribute. It returns `DW_DLV_ERROR` if an error occurred.

5.3.13 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.

5.3.14 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.

5.3.15 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.

5.3.16 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.

5.3.17 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.

5.4 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.

5.4.1 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.

5.4.2 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.

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

5.4.4 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.

5.4.5 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 this class. It returns `DW_DLV_ERROR` on error. See also `dwarf_global_formref` below.

5.4.6 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.

5.4.7 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.

5.4.8 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` `1` (i.e. true) (if the attribute has a non-zero value) or `0` (i.e. false) (if the attribute has a zero value). It returns `DW_DLV_ERROR` on error or if the `attr` does not have form flag.

5.4.9 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.

5.4.10 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.

5.4.11 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.

5.4.12 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.

5.4.13 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.

5.4.14 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.

5.4.15 dwarf_get_form_class()

---

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