A Consumer Library Interface to DWARF

UNIX® International Programming Languages Special Interest Group

1. INTRODUCTION

This document describes the proposed 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.

1.1 Purpose and Scope

The purpose of this document is to propose 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 will be addressed seperately.

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

1.2 Definitions

DWARF debugging information entries (DIE) 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 document "*DWARF Debugging Information Format*" from UI PLSIG for a more complete description of these entries.

This document adopts all the terms and definitions in "*DWARF Debugging Information Format*" version 2. It focuses on the implementation at Silicon Graphics Computer Systems. Although we believe the interface is general enough to be of interest to other vendors too, there are a few places where changes may need to be made.

1.3 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's 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.

1.4 Items Changed

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 book by Steve Maguire titled "Writing Solid Code" at your bookstore.

1.5 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.

1.6 Revision History

March 93	Work on dwarf2 SGI draft begins
June 94	The function returns are changed to return an error/success code only.

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 libdwarf error handler error handler function**

Figure 1. Scalar Types

2.3 Aggregate Types

The following aggregate types are defined by the SGI *libdwarf.h*: `Dwarf_Loc`, `Dwarf_Locdesc`, `Dwarf_Block`, `Dwarf_Frame_Op`. 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

The `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 implemenation 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.

2.4 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 free'd, using `dwarf_dealloc()` when it is no longer of use. 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 free'd, 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 free'd, 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 free'd, 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). The storage pointed to by these descriptors should be individually free'd, using `dwarf_dealloc()` with the allocation type `DW_DLA_GLOBAL`, when no longer needed.

```
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 free'd, using `dwarf_dealloc()` with the allocation type `DW_DLA_WEAK` 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. The storage pointed to by these descriptors should be individually free'd, using `dwarf_dealloc()` with the allocation type `DW_DLA_FUNC`, when no longer needed.

`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. The storage pointed to by this descriptor should be individually free'd, using `dwarf_dealloc()` with the allocation type `DW_DLA_TYPENAME` when no longer needed.

`typedef struct Dwarf_Var_s* Dwarf_Var;`

Instances of `Dwarf_Var` type are returned from a successful call to the SGI-specfic `dwarf_get_vars()` function, and are used as descriptors for queries about static variables. The storage pointed to by this descriptor should be individually free'd, using `dwarf_dealloc()` with the allocation type `DW_DLA_VAR` when no longer needed.

`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 free'd, 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 free'd, 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 free'd, 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. The storage pointed to by these descriptors should be individually free'd, using `dwarf_dealloc()` with the allocation type `DW_DLA_FDE` when no longer needed.

`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. The storage pointed to by this descriptor should be individually free'd, using `dwarf_dealloc()` with the allocation type `DW_DLA_CIE` when no longer needed.

`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 free'd, 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 occured 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 behaviour is undefined. In particular, supplying a `NULL` pointer to a `libdwarf` function (except where explicitely permitted), or pointers to invalid addresses or uninitialized data causes undefined behaviour; 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 thru the pointer. No other pointer value in the interface returns a value.

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

If `DW_DLV_NO_OK` is returned the `Dwarf_Error` pointer, if supplied, is not touched, but any other values to be returned through pointers are returned.

Pointers passed to allow values to be returned thru 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 thru *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

In some cases the pointers returned by a *libdwarf* call are pointers to data which is not free-able. 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 sepcifying 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;

if ((errv = dwarf_attrlist(somedie, &atlist,&atcnt, &error)) == 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()` is the only dynamic storage that cannot be free'd, 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.

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**
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	**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	**Dwarf_Func**
DW_DLA_TYPENAME	**Dwarf_Type**
DW_DLA_VAR	**Dwarf_Var**
DW_DLA_WEAK	**Dwarf_Weak**

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.

```
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 thru `dbg` a `Dwarf_Debug` descriptor that represents a handle for accessing debugging records associated with the open file descriptor `fd`. `DW_DLV_ERROR` is returned if the object does not contain debugging information or 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()`.

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.

```
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.

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

When it returns `DW_DLV_OK`, the function `dwarf_get_elf()` returns thru 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.

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

```
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.

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

```
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);
```

The function `dwarf_next_cu_header()` 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()` 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`, or `address_size` is `NULL`, the argument is ignored (meaning it is not an error to provide a `NULL` pointer).

```
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` tag.

```
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`.

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

The function `dwarf_child()` 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. 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.

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 unsucessful 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.

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

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

```
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. 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 die's i.e .debug_info. It returns `DW_DLV_ERROR` on error.

```
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.

```
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 free'd using the allocation type `DW_DLA_STRING` when no longer of interest (see `dwarf_dealloc()`).

```
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 thru `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 free'd 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;

if ((errv = dwarf_attrlist(somedie, &atlist,&atcnt, &error)) == 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);
}
```

```
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`.

```
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 ***`W_DLV_NO_ENTRY`*** if `attr` is not contained in `die`. It returns ***`W_DLV_ERROR`*** if an error occurred.

```
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.

```
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.

```
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.

```
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.

```
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. 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.

```
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.

```
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 Form Queries

Based on the attribute's 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.

```
nt 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.

```
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.

```
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.

```
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 offset represented by the descriptor `attr` if the form of the attribute belongs to the `REFERENCE` class. It is an error for the form to not belong to this class. It returns `DW_DLV_ERROR` on error.

```
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.

```
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.

```
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.

```
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.

```
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 free'd using the allocation type `DW_DLA_BLOCK`, when no longer of interest (see `dwarf_dealloc()`). It returns `DW_DLV_ERROR` on error.

```
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 be free'd using the allocation type `DW_DLA_STRING` when no longer of interest (see `dwarf_dealloc()`). It returns `DW_DLV_ERROR` on error.

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

The function `dwarf_loclist()` 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. 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;

if ((lres = dwarf_loclist(someattr, &llbuf,&lcnt &error)) == DW_DLV_OK) {
        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 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.

5.5.1 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.

```
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, each line number information structure pointed to by an entry in the block should be free'd using `dwarf_dealloc()` with the allocation type `DW_DLA_LINE` when no longer of interest. Also the block of descriptors itself should be free'd using `dwarf_dealloc()` with the allocation type `DW_DLA_LIST` when no longer of interest.

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

if ((sres = dwarf_srclines(somedie, &linebuf, &error)) == 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);
}
```

5.5.2 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`. 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 free'd 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;

if ((res = dwarf_srcfiles(somedie, &srcfiles,&cnt &error)) == 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);
}
```

5.5.3 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.

```
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`.

```
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* if `line` represents a line number entry that is marked as ending a text sequence) or *zero* ((if `line` represents a line number entry that is not marked as ending a text sequence). It returns `DW_DLV_ERROR` on error. It never returns `DW_DLV_NO_ENTRY`.

```
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`.

```
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`.

```
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 *-1* if the column number of the statement is not represented. On error it returns `DW_DLV_ERROR`. It never returns `DW_DLV_NO_ENTRY`.

```
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. The storage pointed to by a successful return of `dwarf_linesrc()` should be free'd using `dwarf_dealloc()` with the allocation type `DW_DLA_STRING` when no longer of interest. It never returns `DW_DLV_NO_ENTRY`.

```
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 Global Name Space Operations

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

5.6.1 Debugger Interface Operations

```
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. 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 this function, the `Dwarf_Global` descriptors should be individually free'd using `dwarf_dealloc()` with the allocation type `DW_DLA_GLOBAL`, 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;

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

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

```
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 free'd using `dwarf_dealloc()`, with the allocation type `DW_DLA_STRING` when no longer of interest. It never returns `DW_DLV_NO_ENTRY`.

```
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 DIE's, 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`.

```
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 DIE's, 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`.

```
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 compilation-unit header of 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 free'd using dwarf_dealloc(), with the allocation type DW_DLA_STRING when no longer of interest.

5.7 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.

5.7.1 Debugger Interface Operations

```
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. On a successful return from this function, the Dwarf_Weak descriptors should be individually free'd using dwarf_dealloc() with the allocation type DW_DLA_WEAK, 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;

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

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

```
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 free'd 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 DIE's, 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`.

```
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 DIE's, 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`.

```
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 weak name, and the compilation-unit header of the compilation-unit containing the weak name, 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 the return value should be free'd using `dwarf_dealloc()`, with the allocation type `DW_DLA_STRING` when no longer of interest.

5.8 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 `DIE`s 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.

5.8.1 Debugger Interface Operations

```
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. It returns `DW_DLV_NOCOUNT` on error. It returns `DW_DLV_NO_ENTRY` if the .debug_funcnames section does not exist. On a successful return from this function, the `Dwarf_Func` descriptors should be individually free'd using `dwarf_dealloc()` with the allocation type `DW_DLA_FUNC`, 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;

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

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

```
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 free'd using `dwarf_dealloc()`, with the allocation type `DW_DLA_STRING` when no longer of interest.

```
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 DIE's, 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`.

```
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 DIE's, 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`.

```
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 compilation-unit header of 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 the return value should be free'd using `dwarf_dealloc()`, with the allocation type `DW_DLA_STRING` when no longer of interest.

5.9 User Defined Type Names Operations

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

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.

5.9.1 Debugger Interface Operations

```
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. 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 this function, the `Dwarf_Type` descriptors should be individually free'd using `dwarf_dealloc()` with the allocation type `DW_DLA_TYPENAME`, 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;

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

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

```
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 free'd using `dwarf_dealloc()`, with the allocation type `DW_DLA_STRING` when no longer of interest.

```
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 DIE's, 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`.

```
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 DIE's, 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`.

```
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 compilation-unit header of 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 the return value should be free'd 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.

5.10.1 Debugger Interface Operations

```
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. It returns DW_DLV_ERROR on error. It returns DW_DLV_NO_ENTRY if the .debug_varnames section does not exist. On a successful return from this function, the Dwarf_Var descriptors should be individually free'd using dwarf_dealloc() with the allocation type DW_DLA_VAR, 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;

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

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

```
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 free'd using `dwarf_dealloc()`, with the allocation type `DW_DLA_STRING` when no longer of interest.

```
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 DIE's, 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`.

```
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 DIE's, 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`.

```
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 compilation-unit header of 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 the return value should be free'd using `dwarf_dealloc()`, with the allocation type `DW_DLA_STRING` when no longer of interest.

5.11 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 frame pointer), as well as a column for the return address. Each cell in the table contains one of the following:

1. A register + offset

2. A register

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*

Figure 3 is machine dependent and represents MIPS cpu register assignments.

NAME	value	PURPOSE
DW_FRAME_CFA_COL	**0**	**column used for CFA**
DW_FRAME_REG1	**1**	**integer regster 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

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*.

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**

Figure 5. Frame Information Special Values

```
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_EROR` 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, each of the structures pointed to by a descriptor should be individually free'd using `dwarf_dealloc()` with either the allocation type `DW_DLA_CIE`, or `DW_DLA_FDE` as appropriate when no longer of interest. Each of the blocks of descriptors should be free'd using `dwarf_dealloc()` with the allocation type `DW_DLA_LIST` when no longer of interest.

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

    if ((fres = dwarf_get_fde_list(dbg,&cie_data,&cie_count,
                    &fde_data,&fde_count,&error)) == 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);
    }
```

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.

```
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`.

```
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.

```
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.

```
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()`.

```
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. 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()`.

```
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. 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()`.

```
int dwarf_expand_frame_instructions(
        Dwarf_Debug dbg,
        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).

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

    if (expand_frame_instructions(dbg,instruction,len, &frameops,&cnt, &error)
             == DW_DLV_OK) {

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

5.12 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.

5.12.1 Location List Internal-level Interface

```
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)
```

`dwarf_dwarf_get_loclist_entry()` returns `DW_DLV_OK` if successful. `DW_DLV_ERROR` is returned on 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`. When `hipc` and `lopc` are zero, this is the end of a particular location list.

The `hipc_offset`, low pc `lopc_offset` are offsets from the beginning of the current procedure, not genuine pc values.

5.13 Abbreviations access

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

```
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_fde` 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.

```
int dwarf_get_abbrev_tag(
        Dwarf_abbrev abbrev,
        Dwarf_Half   return_tag,
        Dwarf_Error *error);P
```

If successfull, `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`.

```
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.

```
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.

5.14 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.

```
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.

5.15 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.

```
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. 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;

    if ((dwarf_get_aranges(dbg, &arang,&cnt, &error)) == 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);
    }
```

```
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.

```
Dwarf_Off dwarf_get_cu_die_offset(
        Dwarf_Arange arange,
        Dwarf_Off   *returned_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_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.

```
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.

5.16 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.

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

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

```
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()`.

The set of errors enumerated in Figure 3 below were defined in Dwarf 1. These errors are not used by the current implementation of Dwarf 2.

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 SGI `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 (contd.)

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 contect 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 should be 0 for MIPS processors**
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 (contd.)

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 necessarily complete; additional errors might be added when functionality to create debugging information entries are added to *libdwarf* and by the implementors of *libdwarf* to describe internal errors not addressed by the above list. Some of the above errors may be unused. Errors may not have the same meaning in different implementations.

```
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.

```
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.

```
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.

CONTENTS

LIST OF FIGURES

A Consumer Library Interface to DWARF

UNIX® International Programming Languages Special Interest Group

ABSTRACT

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

The document is oriented to reading DWARF version 2. There are certain sections which are SGI-specific (those are clearly identified in the document). We intend to propose this to the PLSIG committee as the basis for a standard libdwarf interface.

The proposals made in this document are subject to change.

$Revision: 1.5 $

$Date: 1994/06/20 18:53:21 $

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