A Producer 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 creation of DWARF debugging information records, DWARF line number information, DWARF address range and pubnames information, weak names informatio, and DWARF frame description information.

1.1 Purpose and Scope

The purpose of this document is to propose a library of functions to create DWARF debugging information. Reading (consuming) of such records is discussed in a separate document.

The functions in this document have been implemented at Silicon Graphics and are being used by the code generator to provide debugging information.

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

Error handling, error codes, and certain *Libdwarf* codes are discussed in the "*Proposed Interface to DWARF Consumer Library*", which should be read (or at least skimmed) before reading this document.

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. and the "*Proposed Interface to DWARF Consumer Library*".

In addition, this document refers to ELF, the ATT/USL System V Release 4 object format. This is because the library was first developed for that object format. Hopefully the functions defined here can easily be applied to other object formats.

1.3 Overview

The remaining sections of this document describe a proposed producer (compiler or assembler) interface to *Libdwarf*, first by describing the purpose of additional types defined by the interface, followed by descriptions of the available operations. This document assumes you are thoroughly familiar with the information contained in the *DWARF Debugging Information Format* document, and "*Proposed Interface to DWARF Consumer Library*".

The interface necessarily knows a little bit about the object format (which is assumed to be ELF). We make an attempt to make this knowledge as limited as possible. For example, *Libdwarf* does not do the writing of object data to the disk. The producer program does that.

1.4 Revision History

March 93	Work on dwarf2 sgi producer draft begins

2. Type Definitions

2.1 General Description

The *libdwarf.h* header file contains typedefs and preprocessor definitions of types and symbolic names used to reference objects of *Libdwarf*. The types defined by typedefs contained in *libdwarf.h* all use the convention of adding `Dwarf_` as a prefix to indicate that they refer to objects used by Libdwarf. The prefix `Dwarf_P_` is used for object referenced by the `Libdwarf` Producer when there are similar but distinct objects used by the Consumer.

2.2 Aggregate Types

3. Memory Management

Several of the functions that comprise the *Libdwarf* interface return values that have been dynamically allocated by the library. Space is always allocated for an object represented by a `Dwarf_P_Debug` descriptor. The `Libdwarf` typically deals with one object at a time. The dynamically allocated spaces can not be reclaimed except by `dwarf_producer_finish()`. This function reclaims all the space, and invalidates all descriptors returned from `Libdwarf` functions that add information to be object specified. After `dwarf_producer_finish()` is called, the `Dwarf_P_Debug` descriptor specified is also invalid.

3.1 Read-only Properties

All pointers returned by or as a result of a *Libdwarf* call should be assumed to point to read-only memory. Except as defined by this document, the results are undefined for *Libdwarf* clients that attempt to write to a region pointed to by a return value from a *Libdwarf* call.

3.2 Storage Deallocation

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

4. Functional Interface

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

The functions may be categorized into groups: *initialization and termination operations*, *debugging information entry creation*, *Elf section callback function*, *attribute creation*, *expression creation*, *line number creation*, *fast-access (aranges) creation*, *fast-access (pubnames) creation*, *fast-access (weak names) creation*, *macro information creation*, *low level (.debug_frame) creation*, and *location list (.debug_loc) creation*.

The following sections describe these functions.

4.1 Initialization and Termination Operations

These functions setup `Libdwarf` to accumulate debugging information for an object, usually a compilation-unit, provided by the producer. The actual addition of information is done by functions in the other sections of this document. Once all the information has been added, functions from this section are used to transform the information to appropriate byte streams, and help to write out the byte streams to disk.

Typically then, a producer would create a `Dwarf_P_Debug` descriptor to gather debugging information for a particular compilation-unit using `dwarf_producer_init()`. They would use this `Dwarf_P_Debug` descriptor to accumulate debugging information for this object using functions from other sections of this document. Once all the information had been added, they would call `dwarf_transform_to_disk_form()` to convert the accumulated information into byte streams in accordance with the `DWARF` standard. They would then repeatedly call `dwarf_get_section_bytes()` for each of the `.debug_*` created. This would give the producer information about the data bytes to be written to disk. At this point, the producer would release all resource used by `Libdwarf` for this object by calling `dwarf_producer_finish()`.

```
Dwarf_P_Debug dwarf_producer_init(
        Dwarf_Unsigned flags,
        Dwarf_Callback_Func func,
        Dwarf_Handler errhand,
        Dwarf_Ptr errarg,
        Dwarf_Error *error)
```

The function `dwarf_producer_init()` returns a new `Dwarf_P_Debug` descriptor that can be used to add `Dwarf` information to the object. On error it returns `DW_DLV_BADADDR`. `flags` determine whether the target object is 64-bit or 32-bit. `func` is a pointer to a function called-back from *Libdwarf* whenever *Libdwarf* needs to create a new object section (as it will for each .debug_* section and related relocation section). `errhand` is a pointer to a function that will be used for handling errors detected by `Libdwarf`. `errarg` is the default error argument used by the function pointed to by `errhand`.

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

The function `dwarf_transform_to_disk_form()` does the actual conversion of the `Dwarf` information provided so far, to the form that is normally written out as `Elf` sections. In other words, once all DWARF information has been passed to *Libdwarf*, call `dwarf_transform_to_disk_form()` to transform all the accumulated data into byte streams. This includes turning relocation information into byte streams. This function does not write anything to disk. If successful, it returns a count of the number of `Elf` sections ready to be retrieved (and, normally, written to disk). In case of error, it returns `DW_DLV_NOCOUNT`.

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

The function `dwarf_get_section_bytes()` must be called repetitively, with the index `dwarf_section` starting at 0 and continuing for the number of sections returned by `dwarf_transform_to_disk_form()`. It returns `NULL` to indicate that there are no more sections of `Dwarf` information. For each non-NULL return, the returned-pointer points to `*length` bytes of data that are normally added to the output object in `Elf` section `*elf_section` by the producer application.

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

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

4.2 Debugging Information Entry Creation

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

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

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

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

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

The function `dwarf_new_die()` creates a new `DIE` with its parent, child, left sibling, and right sibling `DIE`s specified by `parent`, `child`, `left_sibling`, and `right_sibling`, respectively. There is no requirement that all of these `DIE`s be specified, i.e. any of these descriptors may be `NULL`. If none is specified, this will be an isolated `DIE`. A `DIE` is transformed to disk form by dwarf_transform_to_disk_form() only if there is a path from the DIE specified by dwarf_add_die_to_debug to it. This function returns DW_DLV_BADADDR on error.

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

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

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

The function dwarf_die_link() links an existing DIE described by the given die to other existing DIEs. The given die can be linked to a parent DIE, a child DIE, a left sibling DIE, or a right sibling DIE by specifying non-NULL parent, child, left_sibling, and right_sibling Dwarf_P_Die descriptors. It returns the given Dwarf_P_Die descriptor, die, on success, and DW_DLV_BADADDR on error.

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

4.3 Attribute Creation

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

Some of the attributes have values that are relocatable. They need a symbol with respect to which the linker will perform relocation. This symbol is specified by means of an index into the Elf symbol table for the object.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.4 Expression Creation

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

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

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

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

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

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

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

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

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

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

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

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

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

4.5 Line Number Operations

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

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

The function `dwarf_add_line_entry()` adds an entry to the section containing information about source lines. It specifies in `code_offset`, the offset from the address set using `dwarfdwarf_lne_set_address()`, of the address of the first instruction in a contiguous block. The source file that gave rise to the instruction is specified by `file_index`, the source line number is specified by `lineno`, and the source column number is specified by `column_number`. `file_index` is the index of the source file in a list of source files which is built up using the function `dwarf_add_file_decl()`.

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

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

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

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

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

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

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

The function `dwarf_lne_end_sequence()` indicates the end of a contiguous block of instructions. To add information about another block of instructions, a call to `dwarf_lne_set_address()` will have to be made to set the address of the start of the target address of the block, followed by calls to `dwarf_add_line_entry()` for each of the instructions in the block.

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

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

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

It returns the index of the string just added, in the list of include directories for the object. This index is then used to refer to this string. It returns `DW_DLV_NOCOUNT` on error.

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

The function `dwarf_add_file_decl()` adds the name of a source file that contributed to the present object. The name of the file is specified by `name`. In case the name is not a fully-qualified pathname, it is prefixed with the name of the directory specified by `dir_idx`. `dir_idx` is the index of the directory to be prefixed in the list builtup using `dwarf_add_directory_decl()`.

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

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

4.6 Fast Access (aranges) Operations

These functions operate on the .debug_aranges section.

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

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

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

4.7 Fast Access (pubnames) Operations

These functions operate on the .debug_pubnames section.

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

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

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

4.8 Fast Access (weak names) Operations

These functions operate on the .debug_weaknames section.

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

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

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

4.9 Static Function Names Operations

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

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

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

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

4.10 File-scope User-defined Type Names Operations

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

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

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

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

4.11 File-scope Static Variable Names Operations

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

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

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

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

4.12 Low Level (.debug_frame) operations

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

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

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

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

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

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

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

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

The function `dwarf_add_frame_fde()` adds the `FDE` specified by `fde` to the list of `FDE`s for the object represented by the given `dbg`. `die` specifies the `DIE` that represents the function whose frame information is specified by the given `fde`. `cie` specifies the index of the `CIE` that should be used to setup the initial conditions for the given frame. `virt_addr` represents the relocatable address at which the code for the given function begins, and `sym_idx` gives the index of the relocatable symbol to be used to relocate this address (`virt_addr` that is). `code_len` specifies the size in bytes of the machine instructions for the given function.

It returns an index to the given `fde`.

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

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

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

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

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

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

CONTENTS

A Producer Library Interface to DWARF

UNIX® International Programming Languages Special Interest Group

ABSTRACT

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

The document is oriented to creating DWARF version 2. It will be proposed to the PLSIG DWARF committee as soon as that makes any sense.

No proposals like this have ever been submitted to the PLSIG committee....

The proposals made in this document are subject to change.

$Revision: 1.5 $ $Date: 1994/05/18 16:56:22 $

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