SPECIFICATION FOR THE DWARF WRITING LIBRARY

Draft #6

October 7, 19119

Debugging Information

The include file "dw.h" should be included to access the DW library.

Data Types

The following types are defined in "dwcnf.h" and may be redefined if the entire library is to be recompiled. ("dw.h" automatically includes "dwcnf.h".);

Type	*Description*
dw_sym_handle	Has a client defined meaning; the DW library will pass these back to the client in **CLIRelocs** for **DW_W_STATIC**, and **DW_W_SEGMENT**.
dw_targ_addr	The contents of **dw_targ_addr** is unimportant to the DW library; it is only used for **sizeof(dw_targ_addr)**. A **dw_targ_addr** is the type that will be emitted for relocations to run-time addresses.
dw_targ_seg	This is the size of the quantity that **DW_W_SEGMENT** emits.

dw_addr_offset	The type used for offsets from some base address. For example, the **start_scope** parameter to typing routines, or the **addr** parameter to line number information. The code assumes this is an unsigned integer type.
dw_addr_delta	An integer type that can hold the largest possible difference between the **addr** parameter for two subsequent calls to **DWLineNum**.
dw_linenum	A line number. It must be an unsigned integer type.
dw_linenum_delta	dw_linenum_delta is a type that can hold the largest possible difference between two adjacent line numbers passed to **DWLineNum** or **DWReference**.
dw_column	A column number. It must be an unsigned integer type.
dw_column_delta	Type that can hold the largest possible difference between two adjacent column numbers passed to **DWReference**.
dw_size_t	Used for sizes of various things such as block constants (i.e. for DWAddConstant) and the **size** parameter to **CLIWrite**.
dw_uconst	An unsigned integer type that can hold the largest possible unsigned integer constant.
dw_sconst	A signed integer type that can hold the largest possible signed integer constant.
dw_sectnum	Enumerated type that can hold all defined Dwarf sections, passed to client functions
dw_out_offset	A integer type that can hold the largest possible section offset passed to **CLISeek** and returned by **CLITell**
dw_reloc_type	A integer type that can hold the all relocation type

Initialization and Finalization

In the following functions, unless specified otherwise all strings are assumed to be null-terminated.

The DW library does not assume that a pointer passed to it is valid beyond the function call used to pass it. For example, you can pass the address of an auto-buffer that contains a string.

All names passed to the DW library should be unmangled.

The **cli** parameter required for all DW functions except **DWInit** is assumed to be a valid value returned by a call to **DWInit**.

Currently DWENTRY is defined to be nothing. It was created in case there is ever a need to put the DW library into a DLL.

dw_client DWENTRY DWInit(dw_init_info *info);

Initialization for a compilation unit. Return an unique client id. This function will call client functions passed to it, so any client function initialization must be done before the call to DWInit.

```
typedef struct {
    void          (*reloc)( dw_ sectnum, uint, ... );
    void          (*write)( dw_ sectnum, const void *, dw_ size_ t );
    void          (*seek)( dw_ sectnum, dw_ out_ offset, int );
    dw_ out_ offset (*tell)( dw_ sectnum );
    void          *(*alloc)( size_ t );
    void          (*free)( void * );
} dw_ funcs;

typedef struct {
    dw_ lang       language;
    uint_ 8        compiler_ options;
    const char    *producer_ name;
    jmp_ buf       exception_ handler;
    dw_ funcs      funcs;
} dw_ init_ info;
```

Member	*Description*
language	Language used.

Constant	*Language*
DWLANG_C	ISO/ANSI C
DWLANG_CPP	C++
DWLANG_FORTRAN	FORTRAN77

compile_options Compilation option, which is a combination of bits:

Bit	*Description*
DW_CM_BROWSER	The library generates the debugging information for the class browser.
DW_CM_DEBUGGER	The library generates the debugging information for the debugger.
DW_CM_UPPER	For FORTRAN - The compiler converts all identifier names to upper case.
DW_CM_LOWER	For FORTRAN - The compiler converts all identifier names to lower case.

producer A string that identifies the compiler.

exception_handler	If the library ends up in a situation which it can't handle (can we say bug ;-)) this jmp_buf will be called with a non-zero value. This is a fatal exit, and the client should not call any of the DW functions. (FIXME: The library is currently not very good at cleaning up memory in these situations.);
funcs	These functions are described in a later section. The initialization routines may call any of them; so any initialization necessary for these routines must be done before DWInit is called.

The details of the above functions are discussed in Part 3.

void DWENTRY DWFini(dw_client cli);

Finalize the debugging information generator. This routine must be called last. It frees any structures that the DW library required, and flushes all the debugging information.

dw_handle DWENTRY DWBeginCompileUnit(dw_client cli, const char *source_filename, const char *directory, dw_loc_handle segment, const unsigned offset_size);

This function is called some time after **DWInit**. The only other DW functions that can be called in between are those dealing with location expressions.

Parameter	*Definition*
source_filename	Name of the source file.
directory	Compilation directory.
segment	A location expression who's result is the code segment portion of the low_pc and high_pc.
offset_size	The size in bytes of the offset portion of an address in this compile unit

The following CLIRelocs will be required:

DW_W_HIGH_PC
DW_W_LOW_PC
DW_W_SECTION_POS
DW_W_UNIT_SIZE

void DWENTRY DWEndCompileUnit(dw_client cli);

This function pairs up with **DWBeginCompileUnit**. After this, until the next **DWBeginCompileUnit**, the only valid calls are those made to location expression routines (or **DWFini**).

4 The DWARF writing library

Ordering Considerations

In general the DW routines are called in an order that matches the order of the declarations during the source program. The sole exception to this are the Macro information routines. Since it is possible to have a separate preprocessor pass, the library assumes that these routines can be called before any of the other routines. That is why the macro routines have a separate mechanism for specifying file and line number.

Macro Information

void DWENTRY DWMacStartFile(dw_client cli, dw_linenum line, const char *name);

Subsequent DWMac calls refer to the named file.

void DWENTRY DWMacEndFile(dw_client cli);

End the current included file.

dw_macro DWENTRY DWMacDef(dw_client cli, dw_linenum line, const char *name);

Defines a macro. **name** is the name of the macro. A **dw_macro** is returned and must be used in a subsequent call to **DWMacFini** (and possibly **DWMacParam**).

void DWENTRY DWMacParam(dw_client cli, dw_macro mac, const char *name);

Adds a parameter to the macro definition **mac**. **name** is the name of the parameter with no leading or trailing white-space. The order of parameters must be the same as they appear in the source program.

void DWENTRY DWMacFini(dw_client cli, dw_macro mac, const char *def);

Finishes the macro definition **mac**. **def** is the definition string.

void DWENTRY DWMacUnDef(dw_client cli, dw_linenum line, const char *name);

Undefines the macro named **name**.

void DWENTRY DWMacUse(dw_client cli, dw_linenum line, const char *name);

Indicate where the macro named **name** is used.

File and Line-Number Management

void DWENTRY DWSetFile(dw_client cli, const char *file);

Specifies the current file. The default is for the source_filename parameter from the dw_init_info to be the current file.

void DWLineNum(dw_client cli, uint info, dw_linenum line, uint col, dw_addr_offset addr);

Sets the current source line number and machine address. The line numbers information of all instructions, not just declarations, are stored by this routine. Note that all source line numbers are relative to the beginning of their corresponding source file. So the line number of the first line of an included file is one.

Parameter	*Definition*
info	The information about the line, which is established by the combination of the following bits:

Bit	*Description*
DW_LN_DEFAULT	There is no special information about the line.

	DW_LN_STMT	The line is the beginning of a statement.
	DW_LN_BLK	The line is the beginning of a block.
line	Source line number, numbered beginning with one on the first line of the file.	
col	Source column number, which begins at 1.	
addr	Address of instruction relative to the beginning of the compilation unit. If it is at all possible the client should call **DWLineNum** with increasing addrs. The line parameter does not have to be increasing. The size of the emitted line number information is smaller if increasing addrs are used. (There is also an implementation limitation that the maximum decrease of addr between two calls is 32768.);	

void DWENTRY DWDeclFile(dw_client cli, const char *name);

Subsequent declarations are from the file named **name**.

void DWENTRY DWDeclPos(dw_client cli, dw_linenum line, dw_column column);

The next declaration occurs at the indicated line and column in the source file set by the last call to **DWDeclFile**. Note that the position is only used for the immediate next declaration. If there are multiple declarations on the same line, then multiple calls should be made.

void DWENTRY DWReference(dw_client cli, dw_linenum line, dw_column column, dw_handle dependant);

Indicate that in the source code there is a reference to the dependant. This reference is attributed to the current scope of debugging information. (i.e., if it is done inside a structure, then the structure is considered to be the "referencer").

Location Expression Routines

Many functions require a **dw_loc_handle**. These are handles for expressions that the debugger will evaluate. A **dw_loc_handle** can be either a single expression, or a list of expressions created by **DWListFini**. The BROWSER is only interested in whether a location expression is present or not; so when creating BROWSER output the client may create an empty location expression and use that wherever appropriate.

The expressions are evaluated on a stack machine, with operations described later. In some cases the stack will be initially empty, in other cases (such as when calculating the address of a structure field) some base address will be on the stack.

A location expression is limited to roughly 64K. Since each op-code is a single byte, this shouldn't pose much of a limitation (famous last words). The destination of the branch instructions **DW_LOC_BRA** and **DW_LOC_SKIP** must be within 32K of the current instruction. (This is a limitation of the DWARF format, not a limitation of the DW library.);

dw_loc_id DWENTRY DWLocInit(dw_client cli);

First function called to create a location expression for a symbol. An unique **dw_loc_id** is returned to the front end.

dw_loc_label DWENTRY DWLocNewLabel(dw_client cli, dw_loc_id loc);

Create a label for the location expression being built in **loc**. This label can be used for forward or backward references by **DW_LOC_SKIP** and **DW_LOC_BRA**.

void DWENTRY DWLocSetLabel(dw_client cli, dw_loc_id loc, dw_loc_label label);

Give the label **label** the address of the next operation emitted into the location expression **loc**.

void DWENTRY DWLocReg(dw_client cli, dw_loc_id loc, uint reg);

This 'operation' informs the debugger that the value it seeks is in the register named by **reg**. FIXME: need to define the possible values of **reg**.

void DWENTRY DWLocStatic(dw_client cli, dw_loc_id loc, dw_sym_handle sym);

This operation pushes the address of **sym** on the stack.

void DWENTRY DWLocSegment(dw_client cli, dw_loc_id loc, dw_sym_handle sym);

This operation pushes the segment of the address of **sym** on the stack.

void DWENTRY DWLocConstU(dw_client cli, dw_loc_id loc, dw_uconst value);

Pushes an atom which is has an unsigned constant value **value**.

void DWENTRY DWLocConstS(dw_client cli, dw_loc_id loc, dw_sconst value);

Pushes an atom which is has a signed constant value **value**.

void DWENTRY DWLocOp0(dw_client cli, dw_loc_id loc, dw_loc_op op);

Performs one of the operations listed below.

Operation	*Description*
DW_LOC_ABS	It pops the top stack entry and pushes its absolute value.
DW_LOC_AND	It pops the top two stack values, performs the logical AND operation on the two, and pushes the result.
DW_LOC_DEREF	It pops the top stack entry and treats it as an address. The value retrieved from that address is pushed. The size of data retrieved from the dereferenced address is an addressing unit.

DW_LOC_DIV	It pops the top two stack values, divides the former second entry by the former top of the stack using signed division, and pushes the result.
DW_LOC_DROP	It pops the value at the top of the stack.
DW_LOC_DUP	It duplicates the value at the top of the stack.
DW_LOC_EQ	Pop two entries from stack, push 1 if they are equal; push 0 otherwise.
DW_LOC_GE, DW_LOC_GT, DW_LOC_LE, DW_LOC_LT	These operation pop the top two stack values, compare the former top of stack from the former second entry, and pushes 1 onto stack if the comparison is true, 0 if it is false. The comparisons are signed comparison.
DW_LOC_MINUS	It pops the top two stack values, subtracts the former top of the stack from the former second entry, and pushes the result.
DW_LOC_MOD	It pops the top two stack values, and pushes the result of the calculation: former second stack entry modulo the former top of the stack.
DW_LOC_MUL	It pops the top two stack values, multiplies them together, and pushes the result.
DW_LOC_NE	Pop two entries from stack, push 0 if they are equal; push 1 otherwise.
DW_LOC_NEG	It pops the top value and pushes its negation.
DW_LOC_NOP	A placeholder; has no side-effects.
DW_LOC_NOT	It pops the top value and pushes its logical complement.
DW_LOC_OR	It pops the top two stack entries, performs the logical OR operation on them, and pushes the result.
DW_LOC_OVER	It duplicates the entry currently second in the stack at the top of the stack.
DW_LOC_PLUS	It pops the top two stack entries, and pushes their sum.
DW_LOC_ROT	It rotates the first three stack entries. The entry at the top of the stack becomes the third entry, the second entry becomes the top, and the third entry becomes the second.
DW_LOC_SHL	It pops the top two stack entries, shifts the former second entry left by the number of bits specified by the former top of the stack, and pushes the result.
DW_LOC_SHR	It pops the top two stack entries, shifts the former second entry right (logically) by the number of bits specified by the former top of the stack, and pushes the result.
DW_LOC_SHRA	It pops the top two stack entries, shifts the former second entry right (arithmetically) by the number of bits specified by the former top of the stack, and pushes the result.
DW_LOC_SWAP	It swaps the top two stack entries.

DW_LOC_XDEREF	It provides an extended dereference mechanism. The entry at the top of the stack is treated as an address. The second stack entry is treated as an "address space identifier" for those architectures that support multiple address spaces. The top two stack elements are popped, a data item is retrieved through an implementation-defined address calculation and pushed as the new stack top. The size of data retrieved is an addressing unit.
DW_LOC_XOR	It pops the top two stack entries, performs the logical EXCLUSIVE-OR operation on them, and pushes the result.

void DWENTRY DWLocOp(dw_client cli, dw_loc_id loc, dw_loc_op op, ...);

Performs one of the following operations:

Operation	*Description*
DW_LOC_BRA	It is followed by a dw_loc_label operand. This operation pops the top stack entry, if the value is not zero, then jump to the label.
DW_LOC_BREG	Followed by two operands, the first is a register, and the second is an dw_sconst to add to the value in the register. The result is pushed onto the stack.
DW_LOC_FBREG	Takes one dw_sconst parameter which is added to the value calculated by the frame_base_loc parameter to the current subroutine, then pushed on the stack.
DW_LOC_PICK	It is followed by a uint operand which is an index. The stack entry with the specified index (0 through 255, inclusive; 0 means the top) is pushed on the stack.
DW_LOC_PLUS_UCONST	It is followed an dw_sconst operand. It pops the top stack entry, adds it to the operand and pushes the result.
DW_LOC_SKIP	It is followed by a dw_loc_label operand. Control is transferred immediately to this label.

dw_loc_handle DWENTRY DWLocFini(dw_client cli, dw_loc_id loc);

Ends the location expression for a symbol, and returns a handle that may be passed to other DW routines.

dw_list_id DWENTRY DWListInit(dw_client cli);

First function called to create a location list for a symbol.

void DWENTRY DWListEntry(dw_client cli, dw_list_id id, dw_sym_handle beg, dw_sym_handle end, dw_loc_handle loc);

Define an entry in the location list.

Parameter	*Description*
beg	A beginning address. This address is relative to the base address of the compilation unit referencing this location list. It marks the beginning of the range over which the location is valid.
end	A ending address. This address is relative to the base address of the compilation unit referencing this location list. It marks the first address past the end of the range over which the location is valid. Overlapping ranges are possible and are interpreted to mean that the value may be found in one of many places during the overlap. A CLIReloc for **DW_W_LABEL** will be made for each dw_sym_handle.
loc	A location expression describing the location of the object over the range specified by the beginning and end addresses.

dw_loc_handle DWENTRY DWListFini(dw_client cli, dw_list_id);

Finishes the creation of the location list.

void DWENTRY DWLocTrash(dw_client cli, dw_loc_handle loc);

Frees the memory associated with the location expression or list loc. A location expression/list can be created and used over and over again until it is freed by calling this function.

Typing Information

Unless otherwise noted, calls to these functions emit debugging information immediately. The DWARF format requires that debugging information appear in the same order as it does in the source code. So, for example, a structure's fields must be created in the same order that they appear in the source program.

Some of the following functions have common parameters. Here is the documentation for these common parameters:

Parameter	***Description***
char *name	A null-terminated type name. i.e., "struct foobar {}" has the name foobar. If this parm is NULL then no name is emitted.
dw_addr_offset start_scope	This is the offset from the low_pc value for the enclosing block that the declaration occurs at. This is most commonly 0.
uint flags	Some routines have additional flags available here; but unless otherwise noted, the following are always available:

Flag	***Description***
DW_DECLARATION	The object is a declaration, not a definition
DW_FLAG_PRIVATE	The object has the C++ private attribute.
DW_FLAG_PROTECTED	The object has the C++ protected attribute.
DW_FLAG_PUBLIC	The object has the C++ public attribute.

dw_handle DWENTRY DWFundamental(dw_client cli, char * name, unsigned fund_idx, unsigned size);

Get a handle for a fundamental type. fund_idx is one of the following:

DW_FT_ADDRESS
DW_FT_BOOLEAN
DW_FT_COMPLEX_FLOAT
DW_FT_FLOAT
DW_FT_SIGNED
DW_FT_SIGNED_CHAR
DW_FT_UNSIGNED
DW_FT_UNSIGNED_CHAR

For convenience, DW_FT_MIN, and DW_FT_MAX are defined. A valid fundamental type is in the range DW_FT_MIN <= ft < DW_FT_MAX. The DW library will always return the same handle when called with the same fundamental type (so the client does not need to save fundamental type handles).

Parameters	*Description*
name	The name of the type being defined.
size	The size in bytes of the type being defined.

dw_handle DWENTRY DWModifier(dw_client cli, dw_handle base_type, uint modifiers);

Specifies a modifier to a type. **base_type** is the base type to be modified with the modifier **modifier**. The available modifiers are:

Modifier Constant	*Description*
DW_MOD_CONSTANT	The object is a constant
DW_MOD_VOLATILE	The object is volatile.
DW_MOD_NEAR	The object is a near object.
DW_MOD_FAR	The object is a far object.
DW_MOD_HUGE	The object is a huge object.
DW_MOD_FAR16	The object is a far16 object.

dw_handle DWENTRY DWTypedef(dw_client cli, dw_handle base_type, const char *name, dw_addr_offset start_scope, uint flags);

This function gives a name to a type. The **name** must not be NULL. The flag value **DW_FLAG_DECLARATION** is not allowed.

dw_handle DWENTRY DWPointer(dw_client cli, dw_handle base_type, uint flags);

Declares a pointer type.

Parameter	*Description*
base_type	The pointed-at type.
flags	Only the following flags are available:

Flags	*Description*
DW_FLAG_REFERENCE	Declare a pointer that is dereferenced automatically.
DW_PTR_TYPE_NORMAL	A normal pointer (i.e. a model dependant pointer).
DW_PTR_TYPE_NEAR16	A near 16-bit pointer.
DW_PTR_TYPE_FAR16	A far 16-bit pointer.
DW_PTR_TYPE_HUGE	A huge 16-bit pointer.
DW_PTR_TYPE_NEAR32	A near 32-bit pointer.
DW_PTR_TYPE_FAR32	A far 32-bit pointer.

dw_handle DWENTRY DWString(dw_client cli, dw_loc_handle string_length, dw_size_t byte_size, const char *name, dw_addr_offset start_scope, uint flags);

Declares a type to be a block of characters.

Parameter	*Description*
string_length	If this parameter is non-NULL then it is a location expression that the debugger executes to get the address where the length of the string is stored in the program. In this case the **byte_size** parameter describes the number of bytes to be retrieved at the location calculated. If **byte_size** is 0, then the debugger will use **sizeof(long)**.
byte_size	If **string_length** is NULL then this parameter is the number of bytes in the string. Otherwise see **string_length**.

dw_handle DWENTRY DWMemberPointer(dw_client cli, dw_handle containing_struct, dw_loc_handle use_location, dw_handle base_type, const char *name, unsigned flags);

Declares a C++ pointer type to a data or function member of a class or structure.

Parameter	*Description*
containing_struct	A handle to the class or struct to whose members objects of this type may point.
use_location	This refers to the location expression which describes how to get to the member it points to from the beginning of the entire class. It expects the base address of the structure/class object to be on the stack before the debugger starts to execute the location description.
base_type	The type of the member to which this object may point to.

Array Types

dw_handle DWENTRY DWBeginArray(dw_client cli, dw_handle elt_type, uint stride_size, const char *name, dw_addr_offset scope, uint flags);

Begin the declaration of an array. This function call must be followed by calls to **DWArrayDimension** and **DWEndArray**.

Parameter	*Description*
elt_type	Handle for the type of the elements of this array.
stride_size	If this value is non-zero then it indicates the number of bits of each element of the array. (Useful if the number of bits used to store an element in the array is different from the number of bits used to store an individual element of type **elt_type**.);

void DWENTRY DWArrayDimension(dw_client cli, const dw_dim_info *info);

Add a dimension to the previously started array. This function must be called for each dimension in the order that the dimensions appear in the source program. **info** points to an instance of the following structure:

```
typedef struct {
    dw_handle   index_type;
    dw_uconst   lo_data;
    dw_uconst   hi_data;
} dw_dim_info;
```

Field	*Description*
hi_bound_fmt	This is similar to **lo_bound_fmt** but describes the high bound of this dimension.
index_type	This is the handle of the type of the indicies for this dimension.
lo_data	The low bound of the array.
hi_data	The upper bound of the array.

void DWENTRY DWEndArray(dw_client cli, dw_handle array_hdl, dw_handle elt_type, uint stride_size, const char *name, dw_addr_offset scope, uint flags);

This finishes the writing of the record to describe the array A sufficient number of calls to **DWArrayDimension** must have been made before **DWEndArray** is called.

Structure Types

dw_handle DWENTRY DWStruct(dw_client cli, uint kind);

Create a handle for a structure type that will be defined later. This handle can be used for other DW routines even before **DWBeginStruct** has been called.

Kind	*Description*
DW_ST_CLASS	A C++ class type.
DW_ST_STRUCT	A structure type.
DW_ST_UNION	A union type.

void DWENTRY DWBeginStruct(dw_client cli, dw_handle struct_hdl, dw_size_t size, const char *name, dw_addr_offset scope, uint flags);

Begin the declaration of the structure reserved by a call to **DWStruct**. This function begins a nesting of the debugging information. Subsequent calls, up to the corresponding **DWEndStruct** call, to the DW library become children of this structure. i.e., this function marks the beginning of the scope of the structure definition.

Parameter	*Description*
struct_hdl	A dw_handle returned by a call to **DWStruct**.
size	If this is non-zero it indicates the number of bytes required to hold an element of this structure including any padding bytes.

void DWENTRY DWAddFriend(dw_client cli, dw_handle friend);

Add **friend** as a friend to the current structure.

dw_handle DWENTRY DWAddInheritance(dw_client cli, dw_handle ancestor, dw_loc_handle loc, uint flags);

Indicate the the current structure inherits from another structure.

Parameter	*Description*
ancestor	The handle of the ancestor to be inherited.

loc	A location expression that describes the location of the beginning of the data members contributed to the entire class by the ancestor relative to the beginning of the address of the data members of the entire class.
flags	In addition to the common values of **flags**, the flag **DW_FLAG_VIRTUAL** may be supplied to indicate that the inheritance serves as a virtual base class. As well, the flag **DW_FLAG_DECLARATION** is not allowed here.

dw_handle DWENTRY DWAddField(dw_client cli, dw_handle field_hdl, dw_loc_handle loc, const char *name, uint flags);

Add a data member to a structure.

Parameter	*Description*
field_hdl	The dw_handle of the type of this field.
loc	A location expression which expects the base address of the structure to be pushed on the stack and calculates the base address of this field. If the structure is a union type, then this parameter may be NULL. If this is a static data member of a class then this parameter may be NULL if the actual definition of the parameter is outside the class.
flags	The additional flag **DW_FLAG_STATIC** may be used to indicate a static structure member.

dw_handle DWENTRY DWAddBitField(dw_client cli, dw_handle field_hdl, dw_loc_handle loc, dw_size_t byte_size, uint bit_offset, uint bit_size, const char *name, uint flags);

Add a bitfield member to a structure.

Parameter	*Description*
field_hdl	the dw_handle of the type of this field.
loc	A location expression which expects the base address of the structure most closely containing the bit field to be pushed and the stack, and which calculates the base address of this field.
byte_size	This field must be the non-zero byte size of the unit of storage containing the bit-field. This is required only if the storage required cannot be determined by the type of the bit-field (i.e., padding bytes). If the size can be determined by the type of the bit-field, then this value may be 0.
bit_offset	The number of bits to the left of the leftmost (most significant); bit of the bit field value.
bit_size	The number of bits occupied by this bit-field value.

void DWENTRY DWEndStruct(dw_client cli);

End the current structure. Client must ensure proper Begin/End matching.

Enumeration Types

dw_handle DWENTRY DWBeginEnumeration(dw_client cli, dw_size_t byte_size, const char *name, dw_addr_offset scope, uint flags);

Begin the definition of an enumerated type. **byte_size** is the number of bytes required to hold an instance of this enumeration. This call must be followed by calls to **DWAddConstant** and **DWEndEnumeration**. No other DW calls may be made before the call to **DWEndEnumeration**. The DWARF standard requires that the constants be defined in *reverse* order to which they appear in the source program.

void DWENTRY DWAddConstant(dw_client cli, dw_uconst value, const char *name);

Add the constant **value** (that is **byte_size** bytes large as determined by the parameter to **DWBeginEnumeration**); with the name **name** to the current enumeration.

void DWENTRY DWEndEnumeration(dw_client cli);

Finish the current enumeration.

Subroutine Type Declarations

These function calls deal with declarations of subroutines. That is, their prototypes, or for use in creating function pointers.

dw_handle DWENTRY DWBeginSubroutineType(dw_client cli, dw_handle return_type, const char *name, dw_addr_offset scope, uint flags);

Begin the nested declaration of the subroutine type. All calls to the DW library after this, until **DWEndSubroutineType** are in the scope of the declaration of the subroutine type. (i.e., if it's a prototyped C function, then declarations before **DWEndSubroutineType** are similar to declarations inside the prototype.) Parameters for this type are declared using the entries **DWAddParmToSubroutineType** and **DWAddEllipsisToSubroutineType**.

Parameter	*Description*
return_type	If the function is void, this parameter must be NULL. Otherwise it is a handle for the return type of the subroutine.
flags	In addition to the standard flags, **DW_FLAG_PROTOTYPED** indicates that the declaration of the subroutine type was prototyped in the source code. As well, the "address class" set of flags used in **DWPointer** are also allowed here (e.g. **DW_TYPE_FAR16** etc.)

void DWENTRY DWEndSubroutineType(dw_client cli);

The client must ensure that proper Begin/End matching is done.

Lexical Blocks

dw_handle DWENTRY DWBeginLexicalBlock(dw_client cli, dw_loc_handle segment, const char *name);

Begin a new lexical scope. **name** may be NULL indicating an un-named scope. Two CLIReloc calls will made, one for **DW_W_LOW_PC** and one for **DW_W_HIGH_PC** which indicate the first byte of the scope, and the first byte beyond the end of the scope. **segment** if non-null is an expression that evaluates to the segment this block is in.

void DWENTRY DWEndLexicalBlock(dw_client cli);

End a lexical scope. As usual, the client must ensure that Begin/End pairs match.

Common Blocks

dw_handle DWENTRY DWBeginCommonBlock(dw_client cli, dw_loc_handle loc, dw_loc_handle segment, const char *name, unsigned flag);

Begin the declarations for the common block named **name** and located at **loc**. **segment** if non-null indicates which segment the common block is in. The only flag that is valid for the **flag** parameter is **DW_FLAG_DECLARATION**.

void DWENTRY DWEndCommonBlock(dw_client cli);

End of declarations for the common block.

dw_handle DWENTRY DWIncludeCommonBlock(dw_client cli, dw_handle common_block);

Used in the subroutine scope that references the common block.

Subroutines

dw_handle DWENTRY DWBeginInlineSubroutine(dw_client cli, dw_handle out_of_line, dw_loc_handle ret_addr, dw_loc_handle segment);

Begin a definition of a particular instance of an inlined subroutine. **out_of_line** is a handle to the "out of line" instance of the subroutine (i.e., a handle from a **DWBeginSubroutine** call that had the DW_FLAG_OUT_OF_LINE flag). Each instance of the inlined subroutine must have it's own copies of entries describing parameters to that subroutine and it's local variables. **ret_addr** gives the location of the return address (if any). **segment** if non-null indicates which segment the expansion occurs in.

dw_handle DWENTRY DWBeginSubroutine(dw_client cli, dw_call_type call_type, dw_handle return_type, dw_loc_handle return_addr_loc, dw_loc_handle frame_base_loc, dw_loc_handle structure_loc, dw_handle member_hdl, dw_loc_handle segment, const char *name, dw_addr_offset start_scope, uint flags);

Begin a declaration/definition of a subroutine or entry point. This begins a nesting of the debugging information, and must be followed by calls to **DWFormalParameter** et al to declare the parameters, types, and variables for this subroutine. Unless **DW_FLAG_DECLARATION** is set, this will require a **DW_W_LOW_PC** and/or a **DW_W_HIGH_PC**.

Parameter	*Description*
call_type	Not currently used, but should be one of: DW_SB_NEAR_CALL DW_SB_FAR_CALL DW_SB_FAR16_CALL
return_type	Handle for the return type. Must be NULL for void-type subroutines.
return_addr_loc	If non-NULL then this is a location expression that calculates the address of memory that stores the return address.
frame_base_loc	If non-NULL then this is a location expression that describes the "frame base" for the subroutine or entry point. (If the frame base changes during the subroutine, it might be desirable for local variables to be calculated from the frame base, and then use a location list for the frame base.);
structure_loc	For member functions of structure types, this calculates the address of the slot for the function within the virtual function table for the enclosing class or structure.
member_hdl	If this is a definition of a member function occuring outside the body of the structure type, then this is the handle for the type definition of the structure.
segment	If non-null then this is a location expression that evaluates to the segment for this subprogram.

The following additional flags are available:

flag	description
DW_FLAG_PROTOTYPED	The function was declared with ANSI-C style prototyping, as opposed to K&R-C style parameter lists.
DW_FLAG_ARTIFICIAL	The function was created by the compiler (i.e. not explicitly declared in any of the user's source files)
DW_FLAG_VIRTUAL	This is a virtual subroutine.
DW_FLAG_PURE_VIRTUAL	This is a pure virtual subroutine.
DW_FLAG_MAIN	For Fortran PROGRAM-type subroutines.
DW_SUB_STATIC	A file static subroutine or function. Also used for a static member function, and for nested subroutine declarations.

DW_SUB_ENTRY A FORTRAN Entry point. DW requires only a DW_W_LOW_PC for this type of function.
DW_FLAG_WAS_INLINED The function was generated inline by the compiler.
DW_FLAG_DECLARED_INLINED The function was declared inline by the user.

void DWENTRY DWEndSubroutine(dw_client cli);

End the current nesting of **DWBeginSubroutine** or **DWBeginInlineSubroutine**.

dw_handle DWENTRY DWFormalParameter(dw_client cli, dw_handle parm_type, dw_loc_handle parm_loc, dw_loc_handle segment, const char *name, uint default_value_type, ...);

Declare a formal parameter to the current function.

parm_type The type of the parameter.
parm_loc A location description that yields the address of the parameter. May be NULL indicating unknown address.
segment A location expression that yields the segment of the parameter. May be NULL indicating the default segment.
default_value_type One of the following:

DW_DEFAULT_NONE There is no default value for this parameter.
DW_DEFAULT_FUNCTION The default value for this parameter is returned by a function with no args, that is specified by a CLIReloc for DW_W_DEFAULT_FUNCTION.
DW_DEFAULT_STRING The default value is a null-terminated string that is specified as an extra parameter to this **DWFormalParameter**.
DW_DEFAULT_BLOCK The default value is a constant block of data that is specified by extra "const void *" and "dw_size_t" parameters to **DWFormalParameter**.

... Extra parameters depend on the **default_value_type**.

dw_handle DWENTRY DWEllipsis(dw_client cli);

Indicate that the current subroutine has unspecified parameters. Used for "..." in C.

dw_handle DWENTRY DWLabel(dw_client cli, dw_loc_handle segment, const char *name, dw_addr_offset start_scope);

Declare a label inside a subroutine. **start_scope** will usually be 0, but is here for future compatibility. A CLIReloc for **DW_W_LABEL** will be made. **segment** if non-null indicates which segment the label belongs to.

dw_handle DWENTRY DWVariable(dw_client cli, dw_handle type, dw_loc_handle loc, dw_handle member_of, dw_loc_handle segment, const char *name, dw_addr_offset start_scope, uint flags);

Declare a variable.

type	The type of this variable.
loc	A location expression yielding the address of this variable.
member_of	If this is the definition of a static data member then this is the handle to the structure type. Otherwise this is NULL.
segment	If this is non-null then it evaluates to the segment the variable is in.
flags	If **DW_FLAG_GLOBAL** is set then this is a global variable. Otherwise it is a local variable. File static variables in C and C++ are considered local variables. If **DW_FLAG_ARTIFICIAL** is set then this is a variable that has been created by the compiler.

dw_handle DWENTRY DWConstant(dw_client cli, dw_handle type, const void *value, dw_size_t len, dw_handle member_of, const char *name, dw_addr_offset start_scope, uint flags);

Declare a named constant.

type	The type of this constant.
value	Pointer to the value for this constant.
len	The length of this constant. If len is 0, then value is considered to be a null-terminated string.
member_of	If this is the definition of a constant member of a structure type, then this is the handle to the structure type. Otherwise it is NULL.

void DWENTRY DWAddress(dw_client cli, uint_32 len);

DWARF builds a table of all the addresses attributed to a compilation unit. The client calls this function to add addresses to this table. **len** is the length of this address range. The base of the address range is filled in by a CLIReloc for **DW_W_ARANGE_ADDR**.

void DWENTRY DWPubname(dw_client cli, dw_handle hdl, const char *name);

These are used to speed up the debugger. This should be called for any name that has global scope. **hdl** is the handle for the debugging entry that declares/defines the **name**.

Required Client Routines

The debugging information has several sections indicated by the following enumerated type:

Constant	*Description*
DW_DEBUG_INFO	This section is called `.debug_ info`, which stores all the debugging information entries.
DW_DEBUG_PUBNAMES	This section is called `.debug_ pubnames`, which stores a table consisting of object name information that is used in lookup by Name.
DW_DEBUG_ARANGES	This section is called `.debug_ aranges`, which stores a table consisting of object address information that is used in lookup by Address.
DW_DEBUG_LINE	This section is called `.debug_ line`, which stores the line number information generated for the compilation units.
DW_DEBUG_LOC	This section is called `.debug_ loc`, which stores the location lists information.
DW_DEBUG_ABBREV	This section is called `.debug_ abbrev`, which stores abbreviation declarations.
DW_DEBUG_MACINFO	This section is called `.debug_ macinfo`, which stores macro information.
DW_DEBUG_REF	This section is called `.WATCOM_ references`, which contains information about the symbols of every instructions in the source files.

DW_DEBUG_MAX	Defined for convenience; it is the number of sections.

Performance Considerations

The DW library does it's best to try and group CLIWrite operations together into one larger CLIWrite, and to try and avoid using CLISeek. But the library does not go out of it's way to provide this massaging of output. The client should attempt to buffer the data itself. CLISeek is most often called on the DW_DEBUG_INFO, and the DW_DEBUG_LOC sections. The other sections may have one CLISeek performed at the DWFini stage, and the seek will be to the zero offset. The client might wish to optimize performance for only the DW_DEBUG_INFO and the DW_DEBUG_LOC sections.

void CLISeek(uint section, long offset, uint mode);

Repositions the pointer in **section** so that subsequent output occurs at the new pointer.

Mode	*Description*
DW_SEEK_SET	The position is set to the absolute location **offset**.
DW_SEEK_CUR	**offset** is added to the current position.
DW_SEEK_END	The position is set to **offset** bytes from the current end of **section**

long CLITell(uint section);

Return the offset of the next byte to be written to the section.

void CLIReloc(uint section, uint reloc_type, ...);

Even when writing BROWSER information, relocations such as DW_W_LOC_PC may be asked for. This is because the DWARF format requires the presence of certain fields to indicate something specific about a record. For example, if a subroutine record doesn't have a low pc then it is assumed to be a declaration of the subroutine rather than a definition.

section	The section to write a relocation entry to.
reloc_type	The type of the relocation, as follows:

DW_W_LOW_PC	Emit a dw_targ_addr. Used by various entry points to get the low pc address of an object.
DW_W_HIGH_PC	Emit a dw_targ_addr. Used by various entry points to get the high pc address of an object.

DW_W_STATIC Emit a dw_targ_addr. This relocation has an extra parameter of type dw_sym_handle. This parameter is the target of the relocation; the offset of the symbol should be generated. This is used any time a location expression involving a **DWLocStatic** is generated.

DW_W_SEGMENT Emit a dw_segment. This relocation has an extra parameter of type dw_sym_handle. It indicates that the segment portion of the address of the symbol should be generated. This is used any time a location expression involving a **DWLocSegment** operation is generated.

DW_W_LABEL Emit a dw_targ_addr. Used by **DWLabel**.

DW_W_SECTION_POS Emit a uint_32. This relocation has an extra parameter of type uint called **targ_sect**. **targ_sect** parameter is the number of a section for which the current offset is the target of the relocation. The relocation is emitted into **section**.

DW_W_DEFAULT_FUNCTION Emit a dw_targ_addr. Used by **DWFormalParameter**.

DW_W_ARANGE_ADDR Emit a dw_targ_addr. Used by **DWAddress**.

DW_W_UNIT_SIZE Emit an uint_32 that is the number of bytes of code in the current compilation unit.

DW_W_MAX Defined for convenience. This enumerated type starts at 0 and goes to DW_W_MAX.

void CLIWrite(uint section, const void *block, size_t len);

Writes out the debugging information.

Parameter	*Description*
section	The section to which the debugging information is written.
block	Points to the debugging information block.
len	Length of the debugging information block.

void *CLIAlloc(size_t size);

Allocates a memory block of size **size** for the library and returns its address. This function cannot return NULL.

void CLIFree(void *blk);

Free the block pointed by **blk**.

Examples

This section needs a major rewrite.

The example below shows what functions should be called in order to store the debugging information for this C program.

N.B. In this example, for all the CLIWrite() calls, only the section id is accurate. Also for all DWLineNum() calls, the advances in machine instruction address are inaccurate.

```
test.c:

1 #include <stdlib.h>

2 int    a;

3 typedef near char NCHAR;

4 void main();
5 {
6    NCHAR    b;

7    b := 5;
8 }

Functions called by the client and the DWARF library:

Client:

        cli_ id = DWInit( DW_ LANG_ C89, DW_ CM_ DEBUGGER, "test.c",
                          "c:\mydir", 0x123, 1, CLILoc, CLIType,
                          CLIName, CLIWrite, CLIAlloc, CLIFree );
```

DWARF Library:

```
        /* Initialize the .debug_ line section */
        CLIWrite( DW_ DEBUG_ LINE, 0, &info, 20, block );

        /* Initialize the .debug_ abbrevs section */
        CLIWrite( DW_ DEBUG_ ABBREVS, 0, &info, 50, block );

        /* Initialize the .debug_ pubnames section */
        CLIWrite( DW_ DEBUG_ PUBNAMES, 0, &info, 50, block );

        /* Initialize the .debug_ aranges section */
        CLIWrite( DW_ DEBUG_ ARANGES, 0, &info, 50, block );

        /* Write all strings to the string table */
        CLIWrite( DW_ DEBUG_ STR, 0, &info, 17, block );
```

Client:

```
#include <stdlib.h>

        DWLineNum( cli_ id, DW_ LN_ STMT|DW_ LN_ BLK, 1, 1, 0 );
        DWIncl( id, "stdlib.h" );
        ...Function calls for "stdlib.h"...
        DWInclFini( cli_ id );
```

DWARF Library:

```
        CLIWrite( DW_ DEBUG_ LINE, 0, &info, 28, block );
        CLIWrite( DW_ DEBUG_ INFO, 30, &info, 12, block );
```

Client:

```
int   a;

        DWLineNum( cli_ id, DW_ LN_ STMT, 1, 1, 4 );
        a_ dw_ handle = DWModSym( cli_ id, a_ cg_ handle, DW_ SM_ VAR,
                        DW_ SM_ GLO|DW_ SM_ FILE, DW_ SM_ NULL );
```

DWARF Library:

```
        name = CLIName( a_ cg_ handle );
                /* It returns the string "a". */
        type = CLIType( a_ cg_ handle );
                /* It returns DW_ FT_ INTEGER. */
        loc = CLILoc( a_ cg_ handle );
        CLIWrite( DW_ DEBUG_ LINE, 0, &info, 28, block );
        CLIWrite( DW_ DEBUG_ INFO, 0, &info, 24, block );
        CLIWrite( DW_ DEBUG_ PUBNAMES, 0, &info, 12, block );
```

Inside CLILoc():

```
        loc_ id = DWLocInt();
        DWLocAtom( cli_ id, a_ cg_ handle, DW_ LOC_ STATIC );
        /* The actual address will be filled in by the client when
           the debugging information is written to the object
file.*/
        a_ loc_ hd = DWLocFini( loc_ id );
        return a_ loc_ hd;
```

Client:

```
typedef near char NCHAR;

        DWLineNum( cli_ id, DW_ LN_ STMT, 1, 1, 14 );
        mod_ handle = DWMod( cli_ id, DW_ FT_ CHAR, DW_ MOD_ NEAR );
        nchar_ handle = DWModSym( cli_ id, nchar_ cg_ handle,
                DW_ SM_ TYPEDEF, DW_ SM_ NULL, DW_ SM_ NULL );
```

DWARF Library:

```
        name = CLIName( nchar_ cg_ handle );
                /* It returns the string "NCHAR". */
        type = CLIType( nchar_ cg_ handle );
                /* It returns mod_ handle. */
        CLIWrite( DW_ DEBUG_ LINE, 0, &info, 20, block );
        CLIWrite( DW_ DEBUG_ INFO, 0, &info, 24, block );
```

Client:

```
void main();

        DWLineNum( cli_ id, DW_ LN_ DEFAULT, 1, 1, 23 );
        pro_ handle = DWBegProc( cli_ id, DW_ SB_ NEAR_ CALL,
DW_ FT_ VOID,
                        ret_ loc_ hd, DW_ LOC_ NULL,
                        DW_ SB_ GLOBAL_ SUB|DW_ SB_ FUNC_ PROTOTYPE );
```

In order to get ret_loc_ad:

```
        loc_ id = DWLocInit();
        DWLocAtom( cli_ id, some_ cg_ handle, DW_ LOC_ STATIC );
                /* Assume that the return address of main() is
stored
                   in a symbol with some_ cg_ handle as its handle.
                   The actual address will be filled in by the
                   client when the debugging information is written
                   to the object file.
*/
        ret_ loc_ ad = DWLocFini( cli_ id );
```

DWARF Library:

```
        CLIWrite( DW_ DEBUG_ LINE, 0, &info, 20, block );
```

Client:

```
{
        DWLineNum( cli_ id, DW_ LN_ BLK, 1, 1, 0 );
```

DWARF Library:

```
        CLIWrite( DW_ DEBUG_ LINE, 0, &info, 24, block );
```

Client:

```
NCHAR b;

        DWLineNum( cli_ id, DW_ LN_ STMT, 1, 1, 10 );
        b_ handle = DWModSym( cli_ id, b_ cg_ handle, DW_ SM_ VAR,
                        DW_ SM_ NULL, DW_ SM_ LOC|DW_ SM_ ROUT );
```

DWARF Library:

```
        loc = CLILoc( b_ cg_ handle );
        name = CLIName( b_ cg_ handle );
                /* It returns the string "b". */
        type = CLIType( b_ cg_ handle );
                /* It returns nchar_ handle. */
        CLIWrite( DW_ DEBUG_ LINE, 0, &info, 20, block );
```

Inside CLILoc():

```
        loc_ id = DWLocInt();
        DWLocAtom( cli_ id, b_ cg_ handle, DW_ LOC_ STACK );
        /* The offset from stack frame base will be filled in by
           the client when the debugging information is written
           to the object file.   */
        b_ loc_ hd = DWLocFini( loc_ id );
        return b_ loc_ hd;
```

Client:

```
b := 5;
        DWLineNum( cli_ id, DW_ LN_ STMT, 1, 4, 14 );
```

DWARF Library:

```
CLIWrite( DW_DEBUG_LINE, 0, &info, 24, block );
```

Client:

```
}
        DWLineNum( cli_id, DW_LN_DEFAULT, 1, 1, 4 );
        DWEndProc( cli_id, pro_handle );
        main_handle = DWModSym( cli_id, main_cg_handle, DW_SM_SUB,
                        DW_SM_NULL, DW_SM_NULL );
```

DWARF Library:

```
name = CLIName( main_cg_handle );
        /* It returns the string "main" */
type = CLIType( main_cg_handle );
        /* It returns pro_handle */
CLIWrite( DW_DEBUG_LINE, 0, &info, 24, block );
CLIWrite( DW_DEBUG_INFO, -50, &info, 86, block );
CLIWrite( DW_DEBUG_REF, 0, &info, 12, block );
CLIWrite( DW_DEBUG_PUBNAMES, 0, &info, 12, block );
        /* For the global object "main" */
```

Client:

```
DWFini( cli_id );
```

DWARF Library:

```
CLIWrite( DW_DEBUG_LINE, 0, &info, 24, block );
CLIWrite( DW_DEBUG_INFO, -120, &info, 54, block );
```

Revision History

Draft	*Description*
Draft 5	Changed the arguments to a number of the function calls for use with draft 5 of dwarf.
Draft 6	Changed the arguments to a number of the function calls for use with draft 6 of dwarf.