MIPS Extensions to DWARF Version 2.0

Silicon Graphics Computer Systems

1. INTRODUCTION

This document describes MIPS extensions to the DWARF debugging information format. The extensions documented here are subject to change at any time.

2. 64 BIT DWARF

The DWARF2 spec has no provision for 64 bit offsets. SGI-IRIX/MIPS Elf64 objects contain DWARF 2 with all offsets (and addresses) as 64bit values. This non-standard extension was adopted in 1992. Nothing in the dwarf itself identifies the dwarf as 64bit. This extension 64bit-offset dwarf cannot be mixed with 32bit-offset dwarf in a single object or executable, and SGI-IRIX/MIPS compilers and tools do not mix the sizes.

In 2001 DWARF3 adopted a very different 64bit-offset format which can be mixed usefully with 32bit-offset DWARF2 or DWARF3. It is not very likely SGI-IRIX/MIPS compilers will switch to the now-standard DWARF3 64bit-offset scheme, but such a switch is theoretically possible and would be a good idea.

SGI-IRIX/MIPS Elf32 objects contain DWARF2 with all offsets (and addresses) 32 bits.

3. How much symbol information is emitted

The following standard DWARF V2 sections may be emitted:

1. .debug_abbrev contains abbreviations supporting the .debug_info section.
2. .debug_info contains Debug Information Entries (DIEs).
3. .debug_frame contains stack frame descriptions.
4. .debug_line contains line number information.
5. .debug_aranges contains address range descriptions.
6. .debug_pubnames contains names of global functions and data.

   The following are MIPS extensions. Theses were created to allow debuggers to know names without having to look at the .debug_info section.

7. .debug_weaknames is a MIPS extension containing .debug_pubnames-like entries describing weak symbols.
8. .debug_funcnames is a MIPS extension containing .debug_pubnames-like entries describing file-static functions (C static functions). The gcc extension of nested subprograms (like Pascal) adds non-global non-static functions. These should be treated like static functions and gcc should add such to this section so that IRIX libexc(3C) will work correctly. Similarly, Ada functions which are non-global should be here too so that libexc(3C) can work. Putting it another way, every function (other than inline code) belongs either in .debug_pubnames or in .debug_funcnames or else libexc(3C) cannot find the function name.
9. .debug_varnames is a MIPS extension containing .debug_pubnames-like entries describing file-static data symbols (C static variables).
10. .debug_typenames is a MIPS extension containing .debug_pubnames-like entries describing file-level types.

The following are not currently emitted.

11. .debug_macinfo Macro information is not currently emitted.
12. .debug_loc Location lists are not currently emitted.
13. .debug_str The string section is not currently emitted.

3.1 Overview of information emitted

We emit debug information in 3 flavors. We mention C here. The situation is essentially identical for f77, f90, and C++.

1. "default C" We emit line information and DIEs for each subprogram. But no local symbols and no type information. Frame information is output. The DW_AT_producer string has the optimization level: for example "-O2". We put so much in the DW_AT_producer that the string is a significant user of space in .debug_info -- this is perhaps a poor use of space. When optimizing the IRIX CC/cc option -DEBUG:optimize_space eliminates such wasted space. Debuggers only currently use the lack of -g of DW_AT_producer as a hint as to how a 'step' command should be interpreted, and the rest of the string is not used for anything (unless a human looks at it for some reason), so if space-on-disk is an issue, it is quite appropriate to use -DEBUG:optimize_space and save disk space. Every function definition (not inline instances though) is mentioned in either .debug_pubnames or .debug_funcnames. This is crucial to allow libexc(3C) stack-traceback to work and show function names (for all languages).
2. "C with full symbols" All possible info is emitted. DW_AT_producer string has all options that might be of interest, which includes -D's, -U's, and the -g option. These options look like they came from the command line. We put so much in the DW_AT_producer that the string is a significant user of space in .debug_info. this is perhaps a poor use of space. Debuggers only currently use the -g of DW_AT_producer as a hint as to how a 'step' command should be interpreted, and the rest of the string is not used for anything (unless a human looks at it for some reason). Every function definition (not inline instances though) is mentioned in either .debug_pubnames or .debug_funcnames. This is crucial to allow libexc(3C) stack-traceback to work and show function names (for all languages).
3. "Assembler (-g, non -g are the same)" Frame information is output. No type information is emitted, but DIEs are prepared for globals.

3.2 Detecting 'full symbols' (-g)

The debugger depends on the existence of the DW_AT_producer string to determine if the compilation unit has full symbols or not. It looks for -g or -g[123] and accepts these as full symbols but an absent -g or a present -g0 is taken to mean that only basic symbols are defined and there are no local symbols and no type information.

In various contexts the debugger will think the program is stripped or 'was not compiled with -g' unless the -g is in the DW_AT_producer string.

3.3 DWARF and strip(1)

The DWARF section ".debug_frame" is marked SHF_MIPS_NOSTRIP and is not stripped by the strip(1) program. This is because the section is needed for doing stack back traces (essential for C++ and Ada exception handling).

All .debug_* sections are marked with elf type SHT_MIPS_DWARF. Applications needing to access the various DWARF sections must use the section name to discriminate between them.

3.4 Evaluating location expressions

When the debugger evaluates location expressions, it does so in 2 stages. In stage one it simply looks for the trivial location expressions and treats those as special cases.

If the location expression is not trivial, it enters stage two. In this case it uses a stack to evaluate the expression.

If the application is a 32-bit application, it does the operations on 32-bit values (address size values). Even though registers can be 64 bits in a 32-bit program all evaluations are done in 32-bit quantities, so an attempt to calculate a 32-bit quantity by taking the difference of 2 64-bit register values will not work. The notion is that the stack machine is, by the dwarf definition, working in address size units.

These values are then expanded to 64-bit values (addresses or offsets). This extension does not involve sign-extension.

If the application is a 64-bit application, then the stack values are all 64 bits and all operations are done on 64 bits.

3.4.1 The fbreg location op

Compilers shipped with IRIX 6.0 and 6.1 do not emit the fbreg location expression and never emit the DW_AT_frame_base attribute that it depends on. However, this changes with release 6.2 and these are now emitted routinely.

4. Frame Information

4.1 Initial Instructions

The DWARF V2 spec provides for "initial instructions" in each CIE (page 61, section 6.4.1). However, it does not say whether there are default values for each column (register).

Rather than force every CIE to have a long list of bytes to initialize all 32 integer registers, we define that the default values of all registers (as returned by libdwarf in the frame interface) are 'same value'. This is a good choice for many non-register-windows implementations.

4.2 Augmentation string in debug_frame

The augmentation string we use in shipped compilers (up thru irix6.2) is the empty string. IRIX6.2 and later has an augmentation string the empty string ("") or "z" or "mti v1" where the "v1" is a version number (version 1).

We do not believe that "mti v1" was emitted as the augmentation string in any shipped compiler.

4.2.1 CIE processing based on augmentation string:

If the augmentation string begins with 'z', then it is followed immediately by a unsigned_leb_128 number giving the code alignment factor. Next is a signed_leb_128 number giving the data alignment factor. Next is a unsigned byte giving the number of the return address register. Next is an unsigned_leb_128 number giving the length of the 'augmentation' fields (the length of augmentation bytes, not including the unsigned_leb_128 length itself). As of release 6.2, the length of the CIE augmentation fields is 0. What this means is that it is possible to add new augmentations, z1, z2, etc and yet an old consumer to understand the entire CIE as it can bypass the augmentation it does not understand because the length of the augmentation fields is present. Presuming of course that all augmentation fields are simply additional information, not some 'changing of the meaning of an existing field'. Currently there is no CIE data in the augmentation for things beginning with 'z'.

If the augmentation string is "mti v1" or "" then it is followed immediately by a unsigned_leb_128 number giving the code alignment factor. Next is a signed_leb_128 number giving the data alignment factor. Next is a unsigned byte giving the number of the return address register.

If the augmentation string is something else, then the code alignment factor is assumed to be 4 and the data alignment factor is assumed to be -1 and the return address register is assumed to be 31. Arbitrarily. The library (libdwarf) assumes it does not understand the rest of the CIE.

4.2.2 FDE processing based on augmentation

If the CIE augmentation string for an fde begins with 'z' then the next FDE field after the address_range field is an unsigned_leb_128 number giving the length of the 'augmentation' fields, and those fields follow immediately.

4.2.2.1 FDE augmentation fields

If the CIE augmentation string is "mti v1" or "" then the FDE is exactly as described in the Dwarf Document section 6.4.1.

Else, if the CIE augmentation string begins with "z" then the next field after the FDE augmentation length field is a Dwarf_Sword size offset into exception tables. If the CIE augmentation string does not begin with "z" (and is neither "mti v1" nor "") the FDE augmentation fields are skipped (not understood). Note that libdwarf actually (as of MIPSpro7.3 and earlier) only tests that the initial character of the augmentation string is 'z', and ignores the rest of the string, if any. So in reality the test is for a _prefix_ of 'z'.

If the CIE augmentation string neither starts with 'z' nor is "" nor is "mti v1" then libdwarf (incorrectly) assumes that the table defining instructions start next. Processing (in libdwarf) will be incorrect.

4.3 Stack Pointer recovery from debug_frame

There is no identifiable means in DWARF2 to say that the stack register is recovered by any particular operation. A 'register rule' works if the caller's stack pointer was copied to another register. An 'offset(N)' rule works if the caller's stack pointer was stored on the stack. However if the stack pointer is some register value plus/minus some offset, there is no means to say this in an FDE. For MIPS/IRIX, the recovered stack pointer of the next frame up the stack (towards main()) is simply the CFA value of the current frame, and the CFA value is precisely a register (value of a register) or a register plus offset (value of a register plus offset). This is a software convention.

5. egcs dwarf extensions (egcs-1.1.2 extensions)

This and following egcs sections describe the extensions currently shown in egcs dwarf2. Note that egcs has chosen to adopt tag and attribute naming as if their choices were standard dwarf, not as if they were extensions. However, they are properly numbered as extensions.

5.1 DW_TAG_format_label 0x4101

For FORTRAN 77, Fortran 90. Details of use not defined in egcs source, so unclear if used.

5.2 DW_TAG_function_template 0x4102

For C++. Details of use not defined in egcs source, so unclear if used.

5.3 DW_TAG_class_template 0x4103

For C++. Details of use not defined in egcs source, so unclear if used.

5.4 DW_AT_sf_names 0x2101

Apparently only output in DWARF1, not DWARF2.

5.5 DW_AT_src_info 0x2102

Apparently only output in DWARF1, not DWARF2.

5.6 DW_AT_mac_info 0x2103

Apparently only output in DWARF1, not DWARF2.

5.7 DW_AT_src_coords 0x2104

Apparently only output in DWARF1, not DWARF2.

5.8 DW_AT_body_begin 0x2105

Apparently only output in DWARF1, not DWARF2.

5.9 DW_AT_body_end 0x2106

Apparently only output in DWARF1, not DWARF2.

6. egcs .eh_frame (non-sgi) (egcs-1.1.2 extensions)

egcs-1.1.2 (and earlier egcs) emits by default a section named .eh_frame for ia32 (and possibly other platforms) which is nearly identical to .debug_frame in format and content. This section is used for helping handle C++ exceptions.

Because after linking there are sometimes zero-ed out bytes at the end of the eh_frame section, the reader code in dwarf_frame.c considers a zero cie/fde length as an indication that it is the end of the section.

6.1 CIE_id 0

The section is an ALLOCATED section in an executable, and is therefore mapped into memory at run time. The CIE_pointer (aka CIE_id, section 6.4.1 of the DWARF2 document) is the field that distinguishes a CIE from an FDE. The designers of the egcs .eh_frame section decided to make the CIE_id be 0 as the CIE_pointer definition is

the number of bytes from the CIE-pointer in the FDE back to the applicable CIE.

In a dwarf .debug_frame section, the CIE_pointer is the offset in .debug_frame of the CIE for this fde, and since an offset can be zero of some CIE, the CIE_id cannot be 0, but must be all 1 bits . Note that the dwarf2.0 spec does specify the value of CIE_id as 0xffffffff (see section 7.23 of v2.0.0), though earlier versions of this extensions document incorrectly said it was not specified in the dwarf document.

6.2 augmentation eh

The augmentation string in each CIE is "eh" which, with its following NUL character, aligns the following word to a 32bit boundary. Following the augmentation string is a 32bit word with the address of the __EXCEPTION_TABLE__, part of the exception handling data for egcs.

6.3 DW_CFA_GNU_window_save 0x2d

This is effectively a flag for architectures with register windows, and tells the unwinder code that it must look to a previous frame for the correct register window set. As of this writing, egcs gcc/frame.c indicates this is for SPARC register windows.

6.4 DW_CFA_GNU_args_size 0x2e

DW_CFA_GNU_args_size has a single uleb128 argument which is the size, in bytes, of the function's stack at that point in the function.

6.5 __EXCEPTION_TABLE__

A series of 3 32bit word entries by default: 0 word: low pc address 1 word: high pc address 2 word: pointer to exception handler code The end of the table is signaled by 2 words of -1 (not 3 words!).

7. Interpretations of the DWARF V2 spec

7.1 template TAG spellings

The DWARF V2 spec spells two attributes in two ways. DW_TAG_template_type_param (listed in Figure 1, page 7) is spelled DW_TAG_template_type_parameter in the body of the document (section 3.3.7, page 28). We have adopted the spelling DW_TAG_template_type_param.

DW_TAG_template_value_param (listed in Figure 1, page 7) is spelled DW_TAG_template_value_parameter in the body of the document (section 3.3.7, page 28). We have adopted the spelling DW_TAG_template_value_parameter.

We recognize that the choices adopted are neither consistently the longer nor the shorter name. This inconsistency was an accident.

7.2 DW_FORM_ref_addrconfusing

Section 7.5.4, Attribute Encodings, describes DW_FORM_ref_addr. The description says the reference is the size of an address on the target architecture. This is surely a mistake, because on a 16bit-pointer-architecture it would mean that the reference could not exceed 16 bits, which makes only a limited amount of sense as the reference is from one part of the dwarf to another, and could (in theory) be *on the disk* and not limited to what fits in memory. Since MIPS is 32 bit pointers (at the smallest) the restriction is not a problem for MIPS/SGI. The 32bit pointer ABIs are limited to 32 bit section sizes anyway (as a result of implementation details). And the 64bit pointer ABIs currently have the same limit as a result of how the compilers and tools are built (this has not proven to be a limit in practice, so far).

This has been clarified in the DWARF3 spec and the IRIX use of DW_FORM_ref_addr being an offset is correct.

7.3 .debug_macinfo in a debugger

It seems quite difficult, in general, to tie specific text(code) addresses to points in the stream of macro information for a particular compilation unit. So it’s been difficult to see how to design a consumer interface to libdwarf for macro information.

The best (simple to implement, easy for a debugger user to understand) candidate seems to be that the debugger asks for macros of a given name in a compilation unit, and the debugger responds with *all* the macros of that name.

7.3.1 only a single choice exists

If there is exactly one, that is usable in expressions, if the debugger is able to evaluate such.

7.3.2 multiple macros with same name.

If there are multiple macros with the same name in a compilation unit, the debugger (and the debugger user and the application programmer) have a problem: confusion is quite possible. If the macros are simple the debugger user can simply substitute by hand in an expression. If the macros are complicated hand substitution will be impractical, and the debugger will have to identify the choices and let the debugger user choose an interpretation.

7.4 Section 6.1.2 Lookup by address problem

Each entry is a beginning-address followed by a length. And the distinguished entry 0,0 is used to denote the end of a range of entries.

This means that one must be careful not to emit a zero length, as in a .o (object file) the beginning address of a normal entry might be 0 (it is a section offset after all), and the resulting 0,0 would be taken as end-of-range, not as a valid entry. A dwarf dumper would have trouble with such data in an object file.

In an a.out or shared object (dynamic shared object, DSO) no text will be at address zero so in such this problem does not arise.

7.5 Section 5.10 Subrange Type Entries problem

It is specified that DW_AT_upper_bound (and lower bound) must be signed entries if there is no object type info to specify the bound type (Sec 5.10, end of section). One cannot tell (with some dwarf constant types) what the signedness is from the form itself (like DW_FORM_data1), so it is necessary to determine the object and type according to the rules in 5.10 and then if all that fails, the type is signed. It’s a bit complicated and earlier versions of mips_extensions incorrectly said signedness was not defined.

7.6 Section 5.5.6 Class Template Instantiations problem

Lots of room for implementor to canonicalize template declarations. Ie various folks won't agree. This is not serious since a given compiler will be consistent with itself and debuggers will have to cope!

7.7 Section 2.4.3.4 # 11. operator spelling

DW_OP_add should be DW_OP_plus (page 14) (this mistake just one place on the page).

7.8 No clear specification of C++ static funcs

There is no clear way to tell if a C++ member function is a static member or a non-static member function. (dwarf2read.c in gdb 4.18, for example, has this observation)

7.9 Misspelling of DW_AT_const_value

Twice in appendix 1, DW_AT_const_value is misspelled as DW_AT_constant_value.

7.10 Mistake in Atribute Encodings

Section 7.5.4, "Attribute Encodings" has a brief discussion of "constant" which says there are 6 forms of constants. It is incorrect in that it fails to mention (or count) the block forms, which are clearly allowed by section 4.1 "Data Object Entries" (see entry number 9 in the numbered list, on constants).

7.11 DW_OP_bregx

The description of DW_OP_bregx in 2.4.3.2 (Register Based Addressing) is slightly misleading, in that it lists the offset first. As section 7.7.1 (Location Expression) makes clear, in the encoding the register number comes first.

8. MIPS attributes

8.1 DW_AT_MIPS_fde

This extension to Dwarf appears only on subprogram TAGs and has as its value the offset, in the .debug_frame section, of the fde which describes the frame of this function. It is an optimization of sorts to have this present.

8.2 DW_CFA_MIPS_advance_loc8 0x1d

This obvious extension to dwarf line tables enables encoding of 8 byte advance_loc values (for cases when such must be relocatable, and thus must be full length). Applicable only to 64-bit objects.

8.3 DW_TAG_MIPS_loop 0x4081

For future use. Not currently emitted. Places to be emitted and attributes that this might own not finalized.

8.4 DW_AT_MIPS_loop_begin 0x2002

For future use. Not currently emitted. Attribute form and content not finalized.

8.5 DW_AT_MIPS_tail_loop_begin 0x2003

For future use. Not currently emitted. Attribute form and content not finalized.

8.6 DW_AT_MIPS_epilog_begin 0x2004

For future use. Not currently emitted. Attribute form and content not finalized.

8.7 DW_AT_MIPS_loop_unroll_factor 0x2005

For future use. Not currently emitted. Attribute form and content not finalized.

8.8 DW_AT_MIPS_software_pipeline_depth 0x2006

For future use. Not currently emitted. Attribute form and content not finalized.

8.9 DW_AT_MIPS_linkage_name 0x2007

The rules for mangling C++ names are not part of the C++ standard and are different for different versions of C++. With this extension, the compiler emits both the DW_AT_name for things with mangled names (recall that DW_AT_name is NOT the mangled form) and also emits DW_AT_MIPS_linkage_name whose value is the mangled name.

This makes looking for the mangled name in other linker information straightforward. It also is passed (by the debugger) to the libmangle routines to generate names to present to the debugger user.

8.10 DW_AT_MIPS_stride 0x2008

F90 allows assumed shape arguments and pointers to describe non-contiguous memory. A (runtime) descriptor contains address, bounds and stride information - rank and element size is known during compilation. The extent in each dimension is given by the bounds in a DW_TAG_subrange_type, but the stride cannot be represented in conventional dwarf. DW_AT_MIPS_stride was added as an attribute of a DW_TAG_subrange_type to describe the location of the stride. Used in the MIPSpro 7.2 (7.2.1 etc) compilers.

If the stride is constant (ie: can be inferred from the type in the usual manner) DW_AT_MIPS_stride is absent.

If DW_AT_MIPS_stride is present, the attribute contains a reference to a DIE which describes the location holding the stride, and the DW_AT_stride_size field of DW_TAG_array_type is ignored if present. The value of the stride is the number of 4 byte words between elements along that axis.

This applies to
a) Intrinsic types whose size is greater
 or equal to 4bytes ie: real*4,integer*8
 complex etc, but not character types.

b) Derived types (ie: structs) of any size,
 unless all components are of type character.

8.11 DW_AT_MIPS_abstract_name 0x2009

This attribute only appears in a DA_TAG_inlined_subroutine DIE. The value of this attribute is a string. When IPA inlines a routine and the abstract origin is in another compilation unit, there is a problem with putting in a reference, since the ordering and timing of the creation of references is unpredicatable with reference to the DIE and compilation unit the reference refers to.

Since there may be NO ordering of the compilation units that allows a correct reference to be done without some kind of patching, and since even getting the information from one place to another is a problem, the compiler simply passes the problem on to the debugger.

The debugger must match the DW_AT_MIPS_abstract_name in the concrete inlined instance DIE with the DW_AT_MIPS_abstract_name in the abstract inlined subroutine DIE.

A dwarf-consumer-centric view of this and other inline issues could be expressed as follows:
If DW_TAG_subprogram
 If has DW_AT_inline is abstract instance root
 If has DW_AT_abstract_origin, is out-of-line instance

of function (need abstract origin for some data)
(abstract root in same CU (conceptually anywhere
a ref can reach, but reaching outside of CU is
a problem for ipa: see DW_AT_MIPS_abstract_name))
If has DW_AT_MIPS_abstract_name is abstract instance
root(must have DW_AT_inline) and this name is used to
match with the abstract root

If DW_TAG_inline_subroutine
Is concrete inlined subprogram instance.
If has DW_AT_abstract_origin, it is a CU-local inline.
If it has DW_AT_MIPS_abstract_name it is an
inline whose abstract root is in another file (CU).

8.12 DW_AT_MIPS_clone_origin 0x200a

This attribute appears only in a cloned subroutine. The procedure is cloned from the same compilation unit. The value of this attribute is a reference to the original routine in this compilation unit.

The 'original' routine means the routine which has all the original code. The cloned routines will always have been 'specialized' by IPA. A routine with DW_AT_MIPS_clone_origin will also have the DW_CC_nocall value of the DW_AT_calling_convention attribute.

8.13 DW_AT_MIPS_has_inlines 0x200b

This attribute may appear in a DW_TAG_subprogram DIE. If present and it has the value True, then the subprogram has inlined functions somewhere in the body.

By default, at startup, the debugger may not look for inlined functions in scopes inside the outer function.

This is a hint to the debugger to look for the inlined functions so the debugger can set breakpoints on these in case the user requests 'stop in foo' and foo is inlined.

8.14 DW_AT_MIPS_stride_byte 0x200c

Created for f90 pointer and assumed shape arrays. Used in the MIPSpro 7.2 (7.2.1 etc) compilers. A variant of DW_AT_MIPS_stride. This stride is interpreted as a byte count. Used for integer*1 and character arrays and arrays of derived type whose components are all character.

8.15 DW_AT_MIPS_stride_elem 0x200d

Created for f90 pointer and assumed shape arrays. Used in the MIPSpro 7.2 (7.2.1 etc) compilers. A variant of DW_AT_MIPS_stride. This stride is interpreted as a byte-pair (2 byte) count. Used for integer*2 arrays.

8.16 DW_AT_MIPS_ptr_dopetype 0x200e

See following.

8.17 DW_AT_MIPS_allocatable_dopetype 0x200f

See following.

8.18 DW_AT_MIPS_assumed_shape_dopetype 0x2010

DW_AT_MIPS_assumed_shape_dopetype, DW_AT_MIPS_allocatable_dopetype, and DW_AT_MIPS_ptr_dopetype have an attribute value which is a reference to a Fortran 90 Dope Vector. These attributes are introduced in MIPSpro7.3. They only apply to f90 arrays (where they are needed to describe arrays never properly described before in debug information). C, C++, f77, and most f90 arrays continue to be described in standard dwarf.

The distinction between these three attributes is the f90 syntax distinction: keywords ’pointer’ and ’allocatable’ with the absence of these keywords on an assumed shape array being the third case.

A "Dope Vector" is a struct (C struct) which describes a dynamically-allocatable array. In objects with full debugging the C struct will be in the dwarf information (of the f90 object, represented like C). A debugger will use the link to find the main struct DopeVector and will use that information to decode the dope vector. At the outer allocatable/assumed-shape/pointer the DW_AT_location points at the dope vector (so debugger calculations use that as a base).

8.19 Overview of debugger use of dope vectors

Fundamentally, we build two distinct representations of the arrays and pointers. One, in dwarf, represents the statically-representable information (the types and variable/type-names, without type size information). The other, using dope vectors in memory, represents the run-time data of sizes. A debugger must process the two representations in parallel (and merge them) to deal with user expressions in a debugger.

8.20 Example f90 code for use in explanation

[Note We want dwarf output with *exactly* this little (arbitrary) example. Not yet available. end Note] Consider the following code.

```
type array_ptr
  real   :: myvar
  real, dimension (:), pointer :: ap
end type array_ptr

type (array_ptr), allocatable, dimension (:) :: arrays

allocate (arrays(20))
do i = 1,20
  allocate (arrays(i)%ap(i))
end do
```

arrays is an allocatable array (1 dimension) whose size is not known at compile time (it has a Dope Vector). At run time, the allocate statement creats 20 array_ptr dope vectors and marks the base arrays dopevector as allocated. The myvar variable is just there to add complexity to the example :-)
In the loop, arrays(1)%ap(1)
 is allocated as a single element array of reals.
In the loop, arrays(2)%ap(2)
 is allocated as an array of two reals.
In the loop, arrays(20)%ap(20)
 is allocated as an array of twenty reals.

8.21 the problem with standard dwarf and this example

In dwarf, there is no way to find the array bounds of arrays(3)%ap, for example, (which are 1:3 in f90 syntax) since any location expression in an ap array lower bound attribute cannot involve the 3 (the 3 is known at debug time and does not appear in the running binary, so no way for the location expression to get to it). And of course the 3 must actually index across the array of dope vectors in ’arrays’ in our implementation, but that is less of a problem than the problem with the ’3’.

Plus dwarf has no way to find the ’allocated’ flag in the dope vector (so the debugger can know when the allocate is done for a particular arrays(j)%ap).

Consequently, the calculation of array bounds and indices for these dynamically created f90 arrays is now pushed of into the debugger, which must know the field names and usages of the dope vector C structure and use the field offsets etc to find data arrays. C, C++, f77, and most f90 arrays continue to be described in standard dwarf. At the outer allocatable/assumed-shape/pointer the DW_AT_location points at the dope vector (so debugger calculations use that as a base).

It would have been nice to design a dwarf extension to handle the above problems, but the methods considered to date were not any more consistent with standard dwarf than this dope vector centric approach: essentially just as much work in the debugger appeared necessary either way. A better (more dwarf-ish) design would be welcome information.

8.22 A simplified sketch of the dwarf information

[Note: Needs to be written. end Note]

8.23 A simplified sketch of the dope vector information

[Note: This one is simplified. Details left out that should be here. Amplify. end Note] This is an overly simplified version of a dope vector, presented as an initial hint. Full details presented later.

```
struct simplified{
  void *base; // pointer to the data this describes
  long  el_len;
  int   assoc:1
  int   ptr_alloc:1
  int   num_dims:3;
  struct dims_s {
    long lb;
    long ext;
    long str_m;
  } dims[7];
};
```

Only 'num_dims' elements of dims[] are actually used.

8.24 The dwarf information

Here is dwarf information from the compiler for the example above, as printed by dwarfdump(1)
[Note:
The following may not be the test.
Having field names with '.' in the name is
not such a good idea, as it conflicts with the
use of '.' in dbx extended naming.
Something else, like _$, would be much easier
to work with in dbx (customers won't care about this,
for the most part,
but folks working on dbx will, and in those
rare circumstances when a customer cares,
the '.' will be a real problem in dbx.).
Note that to print something about .base., in dbx one
would have to do

```
	whatis `.base.`
```

where that is the grave accent, or back-quote I am using.
With extended naming one do

```
	whatis `.dope.`.`.base.`
```

which is hard to type and hard to read.
end Note]

```
<2>< 388>      DW_TAG_array_type
```

```
            DW_AT_name              .base.
            DW_AT_type              <815>
            DW_AT_declaration       yes(1)
<3>< 401>   DW_TAG_subrange_type
            DW_AT_lower_bound       0
            DW_AT_upper_bound       0
<2>< 405>   DW_TAG_pointer_type
            DW_AT_type              <388>
            DW_AT_byte_size         4
            DW_AT_address_class     0
<2>< 412>   DW_TAG_structure_type
            DW_AT_name              .flds.
            DW_AT_byte_size         28
<3>< 421>   DW_TAG_member
            DW_AT_name              el_len
            DW_AT_type              <815>
            DW_AT_data_member_location  DW_OP_consts 0
<3>< 436>   DW_TAG_member
            DW_AT_name              assoc
            DW_AT_type              <841>
            DW_AT_byte_size         0
            DW_AT_bit_offset        0
            DW_AT_bit_size          1
            DW_AT_data_member_location  DW_OP_consts 4
<3>< 453>   DW_TAG_member
            DW_AT_name              ptr_alloc
            DW_AT_type              <841>
            DW_AT_byte_size         0
            DW_AT_bit_offset        1
            DW_AT_bit_size          1
            DW_AT_data_member_location  DW_OP_consts 4
<3>< 474>   DW_TAG_member
            DW_AT_name              p_or_a
            DW_AT_type              <841>
            DW_AT_byte_size         0
            DW_AT_bit_offset        2
            DW_AT_bit_size          2
            DW_AT_data_member_location  DW_OP_consts 4
<3>< 492>   DW_TAG_member
            DW_AT_name              a_contig
            DW_AT_type              <841>
            DW_AT_byte_size         0
            DW_AT_bit_offset        4
            DW_AT_bit_size          1
            DW_AT_data_member_location  DW_OP_consts 4
<3>< 532>   DW_TAG_member
            DW_AT_name              num_dims
            DW_AT_type              <841>
            DW_AT_byte_size         0
            DW_AT_bit_offset        29
            DW_AT_bit_size          3
            DW_AT_data_member_location  DW_OP_consts 8
<3>< 572>   DW_TAG_member
            DW_AT_name              type_code
```

```
            DW_AT_type                  <841>
            DW_AT_byte_size              0
            DW_AT_bit_offset             0
            DW_AT_bit_size              32
            DW_AT_data_member_location  DW_OP_consts 16
<3>< 593>   DW_TAG_member
            DW_AT_name                  orig_base
            DW_AT_type                  <841>
            DW_AT_data_member_location  DW_OP_consts 20
<3>< 611>   DW_TAG_member
            DW_AT_name                  orig_size
            DW_AT_type                  <815>
            DW_AT_data_member_location  DW_OP_consts 24
<2>< 630>   DW_TAG_structure_type
            DW_AT_name                  .dope_bnd.
            DW_AT_byte_size              12
<3>< 643>   DW_TAG_member
            DW_AT_name                  lb
            DW_AT_type                  <815>
            DW_AT_data_member_location  DW_OP_consts 0
<3>< 654>   DW_TAG_member
            DW_AT_name                  ext
            DW_AT_type                  <815>
            DW_AT_data_member_location  DW_OP_consts 4
<3>< 666>   DW_TAG_member
            DW_AT_name                  str_m
            DW_AT_type                  <815>
            DW_AT_data_member_location  DW_OP_consts 8
<2>< 681>   DW_TAG_array_type
            DW_AT_name                  .dims.
            DW_AT_type                  <630>
            DW_AT_declaration            yes(1)
<3>< 694>   DW_TAG_subrange_type
            DW_AT_lower_bound            0
            DW_AT_upper_bound            0
<2>< 698>   DW_TAG_structure_type
            DW_AT_name                  .dope.
            DW_AT_byte_size              44
<3>< 707>   DW_TAG_member
            DW_AT_name                  base
            DW_AT_type                  <405>
            DW_AT_data_member_location  DW_OP_consts 0
<3>< 720>   DW_TAG_member
            DW_AT_name                  .flds
            DW_AT_type                  <412>
            DW_AT_data_member_location  DW_OP_consts 4
<3>< 734>   DW_TAG_member
            DW_AT_name                  .dims.
            DW_AT_type                  <681>
            DW_AT_data_member_location  DW_OP_consts 32
<2>< 750>   DW_TAG_variable
            DW_AT_type                  <815>
            DW_AT_location              DW_OP_fbreg -32
            DW_AT_artificial            yes(1)
```

```
<2>< 759>      DW_TAG_variable
               DW_AT_type                  <815>
               DW_AT_location              DW_OP_fbreg -28
               DW_AT_artificial            yes(1)
<2>< 768>      DW_TAG_variable
               DW_AT_type                  <815>
               DW_AT_location              DW_OP_fbreg -24
               DW_AT_artificial            yes(1)
<2>< 777>      DW_TAG_array_type
               DW_AT_type                  <815>
               DW_AT_declaration           yes(1)
<3>< 783>      DW_TAG_subrange_type
               DW_AT_lower_bound           <750>
               DW_AT_count                 <759>
               DW_AT_MIPS_stride           <768>
<2>< 797>      DW_TAG_variable
               DW_AT_decl_file             1
               DW_AT_decl_line             1
               DW_AT_name                  ARRAY
               DW_AT_type                  <698>
               DW_AT_location              DW_OP_fbreg -64 DW_OP_deref
<1>< 815>      DW_TAG_base_type
               DW_AT_name                  INTEGER_4
               DW_AT_encoding              DW_ATE_signed
               DW_AT_byte_size             4
<1>< 828>      DW_TAG_base_type
               DW_AT_name                  INTEGER_8
               DW_AT_encoding              DW_ATE_signed
               DW_AT_byte_size             8
<1>< 841>      DW_TAG_base_type
               DW_AT_name                  INTEGER*4
               DW_AT_encoding              DW_ATE_unsigned
               DW_AT_byte_size             4
<1>< 854>      DW_TAG_base_type
               DW_AT_name                  INTEGER*8
               DW_AT_encoding              DW_ATE_unsigned
               DW_AT_byte_size             8
```

8.25 The dope vector structure details

A dope vector is the following C struct, "dopevec.h". Not all the fields are of use to a debugger. It may be that not all fields will show up in the f90 dwarf (since not all are of interest to debuggers).
[Note:
Need details on the use of each field.
And need to know which are really 32 bits and which
are 32 or 64.
end Note]
The following
struct
is a representation of all the dope vector fields.
It suppresses irrelevant detail and may not
exactly match the layout in memory (a debugger must
examine the dwarf to find the fields, not
compile this structure into the debugger!).

```
struct .dope. {
 void *base;   // pointer to data
 struct .flds. {
  long el_len; // length of element in bytes?
  unsigned int assoc:1;     //means?
  unsigned int ptr_alloc:1;     //means?
  unsigned int p_or_a:2;    //means?
  unsigned int a_contig:1;  // means?
  unsigned int num_dims: 3; // 0 thru 7
  unsigned int type_code:32; //values?
  unsigned int orig_base; //void *? means?
  long         orig_size; // means?
 } .flds;

 struct .dope_bnd. {
   long lb   ; // lower bound
   long ext  ;  // means?
   long str_m; // means?
 } .dims[7];
}
```

8.26 DW_AT_MIPS_assumed_size 0x2011

This flag was invented to deal with f90 arrays. For example:

```
pointer (rptr, axx(1))
pointer (iptr, ita(*))
rptr = malloc (100*8)
iptr = malloc (100*4)
```

This flag attribute has the value ’yes’ (true, on) if and only if the size is unbounded, as iptr is. Both may show an explicit upper bound of 1 in the dwarf, but this flag notifies the debugger that there is explicitly no user-provided size.

So if a user asks for a printout of the rptr allocated array, the default will be of a single entry (as there is a user slice bound in the source). In contrast, there is no explicit upper bound on the iptr (ita) array so the default slice will use the current bound (a value calculated from the malloc size, see the dope vector).

Given explicit requests, more of rptr(axx) can me shown than the default.

9. Line information and Source Position

DWARF does not define the meaning of the term ’source statement’. Nor does it define any way to find the first user-written executable code in a function.

It does define that a source statement has a file name, a line number, and a column position (see Sec 6.2, Line Number Information of the Dwarf Version 2 document). We will call those 3 source coordinates a ’source position’ in this document. We’ll try not to accidentally call the source position a ’line number’ since that is ambiguous as to what it means.

9.1 Definition of Statement

A function prolog is a statement.

A C, C++, Pascal, or Fortran statement is a statement.

Each initialized local variable in C,C++ is a statement in that its initialization generates a source position. This means that

x =3, y=4; is two statements.

For C, C++: The 3 parts a,b,c in for(a;b;c) {d;} are individual statements. The condition portion of a while() and do {} while() is a statement. (of course d; can be any number of statements)

For Fortran, the controlling expression of a DO loop is a statement. Is a 'continue' statement in Fortran a DWARF statement?

Each function return, whether user coded or generated by the compiler, is a statement. This is so one can step over (in a debugger) the final user-coded statement (exclusive of the return statement if any) in a function wile not leaving the function scope.

9.2 Finding The First User Code in a Function

Consider:

```
int func(int a)
{               /* source position 1 */
    float b = a; /* source position 2 */
    int x;
    x = b + 2;   /* source position 3 */
}               /* source position 4 */
```

The DIE for a function gives the address range of the function, including function prolog(s) and epilog(s)

Since there is no scope block for the outer user scope of a function (and thus no beginning address range for the outer user scope: the DWARF committee explicitly rejected the idea of having a user scope block) it is necessary to use the source position information to find the first user-executable statement.

This means that the user code for a function must be presumed to begin at the code location of the second source position in the function address range.

If a function has exactly one source position, the function presumably consists solely of a return.

If a function has exactly two source positions, the function may consist of a function prolog and a return or a single user statement and a return (there may be no prolog code needed in a leaf function). In this case, there is no way to be sure which is the first source position of user code, so the rule is to presume that the first address is user code.

If a function consists of 3 or more source positions, one should assume that the first source position is function prolog and the second is the first user executable code.

9.3 Using debug_frame Information to find first user statement

In addition to the line information, the debug_frame information can be useful in determining the first user source line.

Given that a function has more than 1 source position, Find the code location of the second source position, then examine the debug_frame information to determine if the Canonical Frame Address (cfa) is updated before the second source position code location. If the cfa is updated, then one can be pretty sure that the code for the first source position is function prolog code.

Similarly, if the cfa is restored in the code for a source position, the source position is likely to represent a function exit block.

9.4 Debugger Use Of Source Position

Command line debuggers, such as dbx and gdb, will ordinarily want to consider multiple statements on one line to be a single statement: doing otherwise is distressing to users since it causes a ’step’ command to appear to have no effect.

An exception for command line debuggers is in determining the first user statement: as detailed above, there one wants to consider the full source position and will want to consider the function return a separate statement. It is difficult to make the function return a separate statement ’step’ reliably however if a function is coded all on one line or if the last line of user code before the return is on the same line as the return.

A graphical debugger has none of these problems if it simply highlights the portion of the line being executed. In that case, stepping will appear natural even stepping within a line.

10. Known Bugs

Up through at least MIPSpro7.2.1 the compiler has been emitting form DW_FORM_DATA1,2, or 4 for DW_AT_const_value in DW_TAG_enumerator. And dwarfdump and debuggers have read this with dwarf_formudata() or form_sdata() and gotten some values incorrect. For example, a value of 128 was printed by debuggers as a negative value. Since dwarfdump and the compilers were not written to use the value the same way, their output differed. For negative enumerator values the compiler has been emitting 32bit values in a DW_FORM_DATA4. The compiler should probably be emitting a DW_FORM_sdata for enumerator values. And consumers of enumerator values should then call form_sdata(). However, right now, debuggers should call form_udata() and only if it fails, call form_sdata(). Anything else will break backward compatibility with the objects produced earlier.

CONTENTS

1. INTRODUCTION 1

2. 64 BIT DWARF 1

3. How much symbol information is emitted 1
   3.1 Overview of information emitted 2
   3.2 Detecting 'full symbols' (-g) 2
   3.3 DWARF and strip(1) 2
   3.4 Evaluating location expressions 2

4. Frame Information 3
   4.1 Initial Instructions 3
   4.2 Augmentation string in debug_frame 3
   4.3 Stack Pointer recovery from debug_frame 4

5. egcs dwarf extensions (egcs-1.1.2 extensions) 4
   5.1 DW_TAG_format_label 0x4101 4
   5.2 DW_TAG_function_template 0x4102 4
   5.3 DW_TAG_class_template 0x4103 4
   5.4 DW_AT_sf_names 0x2101 4
   5.5 DW_AT_src_info 0x2102 4
   5.6 DW_AT_mac_info 0x2103 4
   5.7 DW_AT_src_coords 0x2104 5
   5.8 DW_AT_body_begin 0x2105 5
   5.9 DW_AT_body_end 0x2106 5

6. egcs .eh_frame (non-sgi) (egcs-1.1.2 extensions) 5
   6.1 CIE_id 0 5
   6.2 augmentation eh 5
   6.3 DW_CFA_GNU_window_save 0x2d 5
   6.4 DW_CFA_GNU_args_size 0x2e 5
   6.5 __EXCEPTION_TABLE__ 5

7. Interpretations of the DWARF V2 spec 5
   7.1 template TAG spellings 5
   7.2 DW_FORM_ref_addr 6
   7.3 .debug_macinfo in a debugger 6
   7.4 Section 6.1.2 Lookup by address problem 6
   7.5 Section 5.10 Subrange Type Entries problem 6
   7.6 Section 5.5.6 Class Template Instantiations problem 7
   7.7 Section 2.4.3.4 # 11. operator spelling 7
   7.8 No clear specification of C++ static funcs 7
   7.9 Misspelling of DW_AT_const_value 7
   7.10 Mistake in Atribute Encodings 7
   7.11 DW_OP_bregx 7

8. MIPS attributes 7
   8.1 DW_AT_MIPS_fde 7
   8.2 DW_CFA_MIPS_advance_loc8 0x1d 7
   8.3 DW_TAG_MIPS_loop 0x4081 7
   8.4 DW_AT_MIPS_loop_begin 0x2002 7
   8.5 DW_AT_MIPS_tail_loop_begin 0x2003 7
   8.6 DW_AT_MIPS_epilog_begin 0x2004 7
   8.7 DW_AT_MIPS_loop_unroll_factor 0x2005 8
   8.8 DW_AT_MIPS_software_pipeline_depth 0x2006 8

8.9 DW_AT_MIPS_linkage_name 0x2007 8
8.10 DW_AT_MIPS_stride 0x2008 8
8.11 DW_AT_MIPS_abstract_name 0x2009 8
8.12 DW_AT_MIPS_clone_origin 0x200a 9
8.13 DW_AT_MIPS_has_inlines 0x200b 9
8.14 DW_AT_MIPS_stride_byte 0x200c 9
8.15 DW_AT_MIPS_stride_elem 0x200d 9
8.16 DW_AT_MIPS_ptr_dopetype 0x200e 9
8.17 DW_AT_MIPS_allocatable_dopetype 0x200f 9
8.18 DW_AT_MIPS_assumed_shape_dopetype 0x2010 9
8.19 Overview of debugger use of dope vectors 10
8.20 Example f90 code for use in explanation 10
8.21 the problem with standard dwarf and this example 10
8.22 A simplified sketch of the dwarf information 11
8.23 A simplified sketch of the dope vector information 11
8.24 The dwarf information 11
8.25 The dope vector structure details 14
8.26 DW_AT_MIPS_assumed_size 0x2011 15

9. Line information and Source Position 15
9.1 Definition of Statement 15
9.2 Finding The First User Code in a Function 16
9.3 Using debug_frame Information to find first user statement 16
9.4 Debugger Use Of Source Position 17

10. Known Bugs 17

MIPS Extensions to DWARF Version 2.0

Silicon Graphics Computer Systems

ABSTRACT

This document describes the MIPS/Silicon Graphics extensions to the "DWARF Information Format" (version 2.0.0 dated July 27, 1993). DWARF3 draft 8 (or draft 9) is out as of 2005, and is mentioned below where applicable. MIPS/IRIX compilers emit DWARF2 (with extensions).

Rather than alter the base documents to describe the extensions we provide this separate document.

The extensions documented here are subject to change.

It also describes known bugs resulting in incorrect dwarf usage.

rev 1.18, 31 March 2005