The “stabs” debug format

Julia Menapace, Jim Kingdon, David MacKenzie
Cygnus Support

Cygnus Support
Revision: 2.130
TEXinfo 2.257

1 Overview of Stabs

Stabs refers to a format for information that describes a program to a debugger. This format was apparently invented by Peter Kessler at the University of California at Berkeley, for the `pdx` Pascal debugger; the format has spread widely since then.

This document is one of the few published sources of documentation on stabs. It is believed to be comprehensive for stabs used by C. The lists of symbol descriptors (see Appendix B [Symbol Descriptors], page 51) and type descriptors (see Appendix C [Type Descriptors], page 53) are believed to be completely comprehensive. Stabs for COBOL-specific features and for variant records (used by Pascal and Modula-2) are poorly documented here.

Other sources of information on stabs are *Dbx and Dbxtool Interfaces*, 2nd edition, by Sun, 1988, and *AIX Version 3.2 Files Reference*, Fourth Edition, September 1992, "dbx Stabstring Grammar" in the a.out section, page 2-31. This document is believed to incorporate the information from those two sources except where it explicitly directs you to them for more information.

1.1 Overview of Debugging Information Flow

The GNU C compiler compiles C source in a '`.c`' file into assembly language in a '`.s`' file, which the assembler translates into a '`.o`' file, which the linker combines with other '`.o`' files and libraries to produce an executable file.

With the '`-g`' option, GCC puts in the '`.s`' file additional debugging information, which is slightly transformed by the assembler and linker, and carried through into the final executable. This debugging information describes features of the source file like line numbers, the types and scopes of variables, and function names, parameters, and scopes.

For some object file formats, the debugging information is encapsulated in assembler directives known collectively as *stab* (symbol table) directives, which are interspersed with the generated code. Stabs are the native format for debugging information in the a.out and XCOFF object file formats. The GNU tools can also emit stabs in the COFF and ECOFF object file formats.

The assembler adds the information from stabs to the symbol information it places by default in the symbol table and the string table of the '`.o`' file it is building. The linker consolidates the '`.o`' files into one executable file, with one symbol table and one string table. Debuggers use the symbol and string tables in the executable as a source of debugging information about the program.

1.2 Overview of Stab Format

There are three overall formats for stab assembler directives, differentiated by the first word of the stab. The name of the directive describes which combination of four possible data fields follows. It is either `.stabs` (string), `.stabn` (number), or `.stabd` (dot). IBM's XCOFF assembler uses `.stabx` (and some other directives such as `.file` and `.bi`) instead of `.stabs`, `.stabn` or `.stabd`.

The overall format of each class of stab is:

```
.stabs "string",type,other,desc,value
.stabn type,other,desc,value
.stabd type,other,desc
.stabx "string",value,type,sdb-type
```

For `.stabn` and `.stabd`, there is no *string* (the `n_strx` field is zero; see Chapter 6 [Symbol Tables], page 33). For `.stabd`, the *value* field is implicit and has the value of the current file location. For `.stabx`, the *sdb-type* field is unused for stabs and can always be set to zero. The *other* field is almost always unused and can be set to zero.

The number in the *type* field gives some basic information about which type of stab this is (or whether it *is* a stab, as opposed to an ordinary symbol). Each valid type number defines a different stab type; further, the stab type defines the exact interpretation of, and possible values for, any remaining *string*, *desc*, or *value* fields present in the stab. See Appendix A [Stab Types], page 47, for a list in numeric order of the valid *type* field values for stab directives.

1.3 The String Field

For most stabs the string field holds the meat of the debugging information. The flexible nature of this field is what makes stabs extensible. For some stab types the string field contains only a name. For other stab types the contents can be a great deal more complex.

The overall format of the string field for most stab types is:

"*name*:*symbol-descriptor type-information*"

name is the name of the symbol represented by the stab; it can contain a pair of colons (see Section 7.2 [Nested Symbols], page 37). *name* can be omitted, which means the stab represents an unnamed object. For example, '`:t10=*2`' defines type 10 as a pointer to type 2, but does not give the type a name. Omitting the *name* field is supported by AIX dbx and GDB after about version 4.8, but not other debuggers. GCC sometimes uses a single space as the name instead of omitting the name altogether; apparently that is supported by most debuggers.

The *symbol-descriptor* following the '`:`' is an alphabetic character that tells more specifically what kind of symbol the stab represents. If the *symbol-descriptor* is omitted, but type information follows, then the stab represents a local variable. For a list of symbol descriptors, see Appendix B [Symbol Descriptors], page 51. The '`c`' symbol descriptor is an exception in that it is not followed by type information. See Chapter 3 [Constants], page 13.

type-information is either a *type-number*, or '*type-number*`=`'. A *type-number* alone is a type reference, referring directly to a type that has already been defined.

The '*type-number*`=`' form is a type definition, where the number represents a new type which is about to be defined. The type definition may refer to other types by number, and those type numbers may be followed by '`=`' and nested definitions. Also, the Lucid compiler will repeat '*type-number*`=`' more than once if it wants to define several type numbers at once.

In a type definition, if the character that follows the equals sign is non-numeric then it is a *type-descriptor*, and tells what kind of type is about to be defined. Any other values following the *type-descriptor* vary, depending on the *type-descriptor*. See Appendix C [Type

Descriptors], page 53, for a list of *type-descriptor* values. If a number follows the '`=`' then the number is a *type-reference*. For a full description of types, Chapter 5 [Types], page 21.

A *type-number* is often a single number. The GNU and Sun tools additionally permit a *type-number* to be a pair (*file-number*,*filetype-number*) (the parentheses appear in the string, and serve to distinguish the two cases). The *file-number* is a number starting with 1 which is incremented for each seperate source file in the compilation (e.g., in C, each header file gets a different number). The *filetype-number* is a number starting with 1 which is incremented for each new type defined in the file. (Separating the file number and the type number permits the `N_BINCL` optimization to succeed more often; see Section 2.3 [Include Files], page 7).

There is an AIX extension for type attributes. Following the '`=`' are any number of type attributes. Each one starts with '`@`' and ends with '`;`'. Debuggers, including AIX's dbx and GDB 4.10, skip any type attributes they do not recognize. GDB 4.9 and other versions of dbx may not do this. Because of a conflict with C++ (see Chapter 7 [Cplusplus], page 37), new attributes should not be defined which begin with a digit, '`(`', or '`-`'; GDB may be unable to distinguish those from the C++ type descriptor '`@`'. The attributes are:

`a`*boundary*
: *boundary* is an integer specifying the alignment. I assume it applies to all variables of this type.

`p`*integer*
: Pointer class (for checking). Not sure what this means, or how *integer* is interpreted.

`P`
: Indicate this is a packed type, meaning that structure fields or array elements are placed more closely in memory, to save memory at the expense of speed.

`s`*size*
: Size in bits of a variable of this type. This is fully supported by GDB 4.11 and later.

`S`
: Indicate that this type is a string instead of an array of characters, or a bitstring instead of a set. It doesn't change the layout of the data being represented, but does enable the debugger to know which type it is.

All of this can make the string field quite long. All versions of GDB, and some versions of dbx, can handle arbitrarily long strings. But many versions of dbx (or assemblers or linkers, I'm not sure which) cretinously limit the strings to about 80 characters, so compilers which must work with such systems need to split the `.stabs` directive into several `.stabs` directives. Each stab duplicates every field except the string field. The string field of every stab except the last is marked as continued with a backslash at the end (in the assembly code this may be written as a double backslash, depending on the assembler). Removing the backslashes and concatenating the string fields of each stab produces the original, long string. Just to be incompatible (or so they don't have to worry about what the assembler does with backslashes), AIX can use '`?`' instead of backslash.

1.4 A Simple Example in C Source

To get the flavor of how stabs describe source information for a C program, let's look at the simple program:

```
main()
{
        printf("Hello world");
}
```

When compiled with '-g', the program above yields the following '.s' file. Line numbers have been added to make it easier to refer to parts of the '.s' file in the description of the stabs that follows.

1.5 The Simple Example at the Assembly Level

This simple "hello world" example demonstrates several of the stab types used to describe C language source files.

```
1  gcc2_compiled.:
2  .stabs "/cygint/s1/users/jcm/play/",100,0,0,Ltext0
3  .stabs "hello.c",100,0,0,Ltext0
4  .text
5  Ltext0:
6  .stabs "int:t1=r1;-2147483648;2147483647;",128,0,0,0
7  .stabs "char:t2=r2;0;127;",128,0,0,0
8  .stabs "long int:t3=r1;-2147483648;2147483647;",128,0,0,0
9  .stabs "unsigned int:t4=r1;0;-1;",128,0,0,0
10 .stabs "long unsigned int:t5=r1;0;-1;",128,0,0,0
11 .stabs "short int:t6=r1;-32768;32767;",128,0,0,0
12 .stabs "long long int:t7=r1;0;-1;",128,0,0,0
13 .stabs "short unsigned int:t8=r1;0;65535;",128,0,0,0
14 .stabs "long long unsigned int:t9=r1;0;-1;",128,0,0,0
15 .stabs "signed char:t10=r1;-128;127;",128,0,0,0
16 .stabs "unsigned char:t11=r1;0;255;",128,0,0,0
17 .stabs "float:t12=r1;4;0;",128,0,0,0
18 .stabs "double:t13=r1;8;0;",128,0,0,0
19 .stabs "long double:t14=r1;8;0;",128,0,0,0
20 .stabs "void:t15=15",128,0,0,0
21      .align 4
22 LC0:
23      .ascii "Hello, world!\12\0"
24      .align 4
25      .global _main
26      .proc 1
27 _main:
28 .stabn 68,0,4,LM1
29 LM1:
30      !#PROLOGUE# 0
31      save %sp,-136,%sp
32      !#PROLOGUE# 1
33      call ___main,0
34      nop
35 .stabn 68,0,5,LM2
36 LM2:
```

```
LBB2:
     sethi %hi(LC0),%o1
     or %o1,%lo(LC0),%o0
     call _printf,0
     nop
.stabn 68,0,6,LM3
LM3:
LBE2:
.stabn 68,0,6,LM4
LM4:
L1:
     ret
     restore
.stabs "main:F1",36,0,0,_main
.stabn 192,0,0,LBB2
.stabn 224,0,0,LBE2
```

2 Encoding the Structure of the Program

The elements of the program structure that stabs encode include the name of the main function, the names of the source and include files, the line numbers, procedure names and types, and the beginnings and ends of blocks of code.

2.1 Main Program

Most languages allow the main program to have any name. The `N_MAIN` stab type tells the debugger the name that is used in this program. Only the string field is significant; it is the name of a function which is the main program. Most C compilers do not use this stab (they expect the debugger to assume that the name is `main`), but some C compilers emit an `N_MAIN` stab for the `main` function. I'm not sure how XCOFF handles this.

2.2 Paths and Names of the Source Files

Before any other stabs occur, there must be a stab specifying the source file. This information is contained in a symbol of stab type `N_SO`; the string field contains the name of the file. The value of the symbol is the start address of the portion of the text section corresponding to that file.

With the Sun Solaris2 compiler, the desc field contains a source-language code.

Some compilers (for example, GCC2 and SunOS4 '`/bin/cc`') also include the directory in which the source was compiled, in a second `N_SO` symbol preceding the one containing the file name. This symbol can be distinguished by the fact that it ends in a slash. Code from the `cfront` C++ compiler can have additional `N_SO` symbols for nonexistent source files after the `N_SO` for the real source file; these are believed to contain no useful information.

For example:

```
.stabs "/cygint/s1/users/jcm/play/",100,0,0,Ltext0     # 100 is N_SO
.stabs "hello.c",100,0,0,Ltext0
        .text
Ltext0:
```

Instead of `N_SO` symbols, XCOFF uses a `.file` assembler directive which assembles to a `C_FILE` symbol; explaining this in detail is outside the scope of this document.

If it is useful to indicate the end of a source file, this is done with an `N_SO` symbol with an empty string for the name. The value is the address of the end of the text section for the file. For some systems, there is no indication of the end of a source file, and you just need to figure it ended when you see an `N_SO` for a different source file, or a symbol ending in `.o` (which at least some linkers insert to mark the start of a new `.o` file).

2.3 Names of Include Files

There are several schemes for dealing with include files: the traditional `N_SOL` approach, Sun's `N_BINCL` approach, and the XCOFF `C_BINCL` approach (which despite the similar name has little in common with `N_BINCL`).

An `N_SOL` symbol specifies which include file subsequent symbols refer to. The string field is the name of the file and the value is the text address corresponding to the end of the previous include file and the start of this one. To specify the main source file again, use an N_SOL symbol with the name of the main source file.

The N_BINCL approach works as follows. An N_BINCL symbol specifies the start of an include file. In an object file, only the string is significant; the linker puts data into some of the other fields. The end of the include file is marked by an N_EINCL symbol (which has no string field). In an object file, there is no significant data in the N_EINCL symbol. N_BINCL and N_EINCL can be nested.

If the linker detects that two source files have identical stabs between an N_BINCL and N_EINCL pair (as will generally be the case for a header file), then it only puts out the stabs once. Each additional occurance is replaced by an N_EXCL symbol. I believe the GNU linker and the Sun (both SunOS4 and Solaris) linker are the only ones which supports this feature.

A linker which supports this feature will set the value of a N_BINCL symbol to the total of all the characters in the stabs strings included in the header file, omitting any file numbers. The value of an N_EXCL symbol is the same as the value of the N_BINCL symbol it replaces. This information can be used to match up N_EXCL and N_BINCL symbols which have the same filename. The N_EINCL value, and the values of the other and description fields for all three, appear to always be zero.

For the start of an include file in XCOFF, use the '.bi' assembler directive, which generates a C_BINCL symbol. A '.ei' directive, which generates a C_EINCL symbol, denotes the end of the include file. Both directives are followed by the name of the source file in quotes, which becomes the string for the symbol. The value of each symbol, produced automatically by the assembler and linker, is the offset into the executable of the beginning (inclusive, as you'd expect) or end (inclusive, as you would not expect) of the portion of the COFF line table that corresponds to this include file. C_BINCL and C_EINCL do not nest.

2.4 Line Numbers

An N_SLINE symbol represents the start of a source line. The desc field contains the line number and the value contains the code address for the start of that source line. On most machines the address is absolute; for stabs in sections (see Appendix F [Stab Sections], page 61), it is relative to the function in which the N_SLINE symbol occurs.

GNU documents N_DSLINE and N_BSLINE symbols for line numbers in the data or bss segments, respectively. They are identical to N_SLINE but are relocated differently by the linker. They were intended to be used to describe the source location of a variable declaration, but I believe that GCC2 actually puts the line number in the desc field of the stab for the variable itself. GDB has been ignoring these symbols (unless they contain a string field) since at least GDB 3.5.

For single source lines that generate discontiguous code, such as flow of control statements, there may be more than one line number entry for the same source line. In this case there is a line number entry at the start of each code range, each with the same line number.

XCOFF does not use stabs for line numbers. Instead, it uses COFF line numbers (which are outside the scope of this document). Standard COFF line numbers cannot deal with include files, but in XCOFF this is fixed with the C_BINCL method of marking include files (see Section 2.3 [Include Files], page 7).

2.5 Procedures

All of the following stabs normally use the `N_FUN` symbol type. However, Sun's `acc` compiler on SunOS4 uses `N_GSYM` and `N_STSYM`, which means that the value of the stab for the function is useless and the debugger must get the address of the function from the non-stab symbols instead. On systems where non-stab symbols have leading underscores, the stabs will lack underscores and the debugger needs to know about the leading underscore to match up the stab and the non-stab symbol. BSD Fortran is said to use `N_FNAME` with the same restriction; the value of the symbol is not useful (I'm not sure it really does use this, because GDB doesn't handle this and no one has complained).

A function is represented by an '`F`' symbol descriptor for a global (extern) function, and '`f`' for a static (local) function. For a.out, the value of the symbol is the address of the start of the function; it is already relocated. For stabs in ELF, the SunPRO compiler version 2.0.1 and GCC put out an address which gets relocated by the linker. In a future release SunPRO is planning to put out zero, in which case the address can be found from the ELF (non-stab) symbol. Because looking things up in the ELF symbols would probably be slow, I'm not sure how to find which symbol of that name is the right one, and this doesn't provide any way to deal with nested functions, it would probably be better to make the value of the stab an address relative to the start of the file, or just absolute. See Section F.2 [ELF Linker Relocation], page 62 for more information on linker relocation of stabs in ELF files. For XCOFF, the stab uses the `C_FUN` storage class and the value of the stab is meaningless; the address of the function can be found from the csect symbol (XTY_LD/XMC_PR).

The type information of the stab represents the return type of the function; thus '`foo:f5`' means that foo is a function returning type 5. There is no need to try to get the line number of the start of the function from the stab for the function; it is in the next `N_SLINE` symbol.

Some compilers (such as Sun's Solaris compiler) support an extension for specifying the types of the arguments. I suspect this extension is not used for old (non-prototyped) function definitions in C. If the extension is in use, the type information of the stab for the function is followed by type information for each argument, with each argument preceded by '`;`'. An argument type of 0 means that additional arguments are being passed, whose types and number may vary ('`...`' in ANSI C). GDB has tolerated this extension (parsed the syntax, if not necessarily used the information) since at least version 4.8; I don't know whether all versions of dbx tolerate it. The argument types given here are not redundant with the symbols for the formal parameters (see Section 4.7 [Parameters], page 18); they are the types of the arguments as they are passed, before any conversions might take place. For example, if a C function which is declared without a prototype takes a `float` argument, the value is passed as a `double` but then converted to a `float`. Debuggers need to use the types given in the arguments when printing values, but when calling the function they need to use the types given in the symbol defining the function.

If the return type and types of arguments of a function which is defined in another source file are specified (i.e., a function prototype in ANSI C), traditionally compilers emit no stab; the only way for the debugger to find the information is if the source file where the function is defined was also compiled with debugging symbols. As an extension the Solaris compiler uses symbol descriptor '`P`' followed by the return type of the function, followed by the arguments, each preceded by '`;`', as in a stab with symbol descriptor '`f`' or '`F`'. This use of symbol descriptor 'P' can be distinguished from its use for register parameters (see Section 4.7.1 [Register Parameters], page 19) by the fact that it has symbol type N_FUN.

The AIX documentation also defines symbol descriptor 'J' as an internal function. I assume this means a function nested within another function. It also says symbol descriptor 'm' is a module in Modula-2 or extended Pascal.

Procedures (functions which do not return values) are represented as functions returning the void type in C. I don't see why this couldn't be used for all languages (inventing a void type for this purpose if necessary), but the AIX documentation defines 'I', 'P', and 'Q' for internal, global, and static procedures, respectively. These symbol descriptors are unusual in that they are not followed by type information.

The following example shows a stab for a function main which returns type number 1. The _main specified for the value is a reference to an assembler label which is used to fill in the start address of the function.

```
.stabs "main:F1",36,0,0,_main       # 36 is N_FUN
```

The stab representing a procedure is located immediately following the code of the procedure. This stab is in turn directly followed by a group of other stabs describing elements of the procedure. These other stabs describe the procedure's parameters, its block local variables, and its block structure.

If functions can appear in different sections, then the debugger may not be able to find the end of a function. Recent versions of GCC will mark the end of a function with an N_FUN symbol with an empty string for the name. The value is the address of the end of the current function. Without such a symbol, there is no indication of the address of the end of a function, and you must assume that it ended at the starting address of the next function or at the end of the text section for the program.

2.6 Nested Procedures

For any of the symbol descriptors representing procedures, after the symbol descriptor and the type information is optionally a scope specifier. This consists of a comma, the name of the procedure, another comma, and the name of the enclosing procedure. The first name is local to the scope specified, and seems to be redundant with the name of the symbol (before the ':'). This feature is used by GCC, and presumably Pascal, Modula-2, etc., compilers, for nested functions.

If procedures are nested more than one level deep, only the immediately containing scope is specified. For example, this code:

```
int
foo (int x)
{
  int bar (int y)
    {
      int baz (int z)
        {
          return x + y + z;
        }
      return baz (x + 2 * y);
    }
```

```
    return x + bar (3 * x);
}
```

produces the stabs:

```
.stabs "baz:f1,baz,bar",36,0,0,_baz.15         # 36 is N_FUN
.stabs "bar:f1,bar,foo",36,0,0,_bar.12
.stabs "foo:F1",36,0,0,_foo
```

2.7 Block Structure

The program's block structure is represented by the `N_LBRAC` (left brace) and the `N_RBRAC` (right brace) stab types. The variables defined inside a block precede the `N_LBRAC` symbol for most compilers, including GCC. Other compilers, such as the Convex, Acorn RISC machine, and Sun `acc` compilers, put the variables after the `N_LBRAC` symbol. The values of the `N_LBRAC` and `N_RBRAC` symbols are the start and end addresses of the code of the block, respectively. For most machines, they are relative to the starting address of this source file. For the Gould NP1, they are absolute. For stabs in sections (see Appendix F [Stab Sections], page 61), they are relative to the function in which they occur.

The `N_LBRAC` and `N_RBRAC` stabs that describe the block scope of a procedure are located after the `N_FUN` stab that represents the procedure itself.

Sun documents the desc field of `N_LBRAC` and `N_RBRAC` symbols as containing the nesting level of the block. However, dbx seems to not care, and GCC always sets desc to zero.

For XCOFF, block scope is indicated with `C_BLOCK` symbols. If the name of the symbol is '`.bb`', then it is the beginning of the block; if the name of the symbol is '`.be`'; it is the end of the block.

2.8 Alternate Entry Points

Some languages, like Fortran, have the ability to enter procedures at some place other than the beginning. One can declare an alternate entry point. The `N_ENTRY` stab is for this; however, the Sun FORTRAN compiler doesn't use it. According to AIX documentation, only the name of a `C_ENTRY` stab is significant; the address of the alternate entry point comes from the corresponding external symbol. A previous revision of this document said that the value of an `N_ENTRY` stab was the address of the alternate entry point, but I don't know the source for that information.

3 Constants

The 'c' symbol descriptor indicates that this stab represents a constant. This symbol descriptor is an exception to the general rule that symbol descriptors are followed by type information. Instead, it is followed by '=' and one of the following:

`b` *value*
: Boolean constant. *value* is a numeric value; I assume it is 0 for false or 1 for true.

`c` *value*
: Character constant. *value* is the numeric value of the constant.

`e` *type-information* , *value*
: Constant whose value can be represented as integral. *type-information* is the type of the constant, as it would appear after a symbol descriptor (see Section 1.3 [String Field], page 2). *value* is the numeric value of the constant. GDB 4.9 does not actually get the right value if *value* does not fit in a host `int`, but it does not do anything violent, and future debuggers could be extended to accept integers of any size (whether unsigned or not). This constant type is usually documented as being only for enumeration constants, but GDB has never imposed that restriction; I don't know about other debuggers.

`i` *value*
: Integer constant. *value* is the numeric value. The type is some sort of generic integer type (for GDB, a host `int`); to specify the type explicitly, use 'e' instead.

`r` *value*
: Real constant. *value* is the real value, which can be 'INF' (optionally preceded by a sign) for infinity, 'QNAN' for a quiet NaN (not-a-number), or 'SNAN' for a signalling NaN. If it is a normal number the format is that accepted by the C library function `atof`.

`s` *string*
: String constant. *string* is a string enclosed in either ''' (in which case ''' characters within the string are represented as '\'' or '"' (in which case '"' characters within the string are represented as '\"').

`S` *type-information* , *elements* , *bits* , *pattern*
: Set constant. *type-information* is the type of the constant, as it would appear after a symbol descriptor (see Section 1.3 [String Field], page 2). *elements* is the number of elements in the set (does this means how many bits of *pattern* are actually used, which would be redundant with the type, or perhaps the number of bits set in *pattern*? I don't get it), *bits* is the number of bits in the constant (meaning it specifies the length of *pattern*, I think), and *pattern* is a hexadecimal representation of the set. AIX documentation refers to a limit of 32 bytes, but I see no reason why this limit should exist. This form could probably be used for arbitrary constants, not just sets; the only catch is that *pattern* should be understood to be target, not host, byte order and format.

The boolean, character, string, and set constants are not supported by GDB 4.9, but it ignores them. GDB 4.8 and earlier gave an error message and refused to read symbols from the file containing the constants.

The above information is followed by ';'.

4 Variables

Different types of stabs describe the various ways that variables can be allocated: on the stack, globally, in registers, in common blocks, statically, or as arguments to a function.

4.1 Automatic Variables Allocated on the Stack

If a variable's scope is local to a function and its lifetime is only as long as that function executes (C calls such variables *automatic*), it can be allocated in a register (see Section 4.3 [Register Variables], page 16) or on the stack.

Each variable allocated on the stack has a stab with the symbol descriptor omitted. Since type information should begin with a digit, '-', or '(', only those characters precluded from being used for symbol descriptors. However, the Acorn RISC machine (ARM) is said to get this wrong: it puts out a mere type definition here, without the preceding '*type-number*='. This is a bad idea; there is no guarantee that type descriptors are distinct from symbol descriptors. Stabs for stack variables use the `N_LSYM` stab type, or `C_LSYM` for XCOFF.

The value of the stab is the offset of the variable within the local variables. On most machines this is an offset from the frame pointer and is negative. The location of the stab specifies which block it is defined in; see Section 2.7 [Block Structure], page 11.

For example, the following C code:

```
int
main ()
{
  int x;
}
```

produces the following stabs:

```
.stabs "main:F1",36,0,0,_main    # 36 is N_FUN
.stabs "x:1",128,0,0,-12         # 128 is N_LSYM
.stabn 192,0,0,LBB2              # 192 is N_LBRAC
.stabn 224,0,0,LBE2              # 224 is N_RBRAC
```

See Section 2.5 [Procedures], page 9 for more information on the `N_FUN` stab, and Section 2.7 [Block Structure], page 11 for more information on the `N_LBRAC` and `N_RBRAC` stabs.

4.2 Global Variables

A variable whose scope is not specific to just one source file is represented by the '`G`' symbol descriptor. These stabs use the `N_GSYM` stab type (C_GSYM for XCOFF). The type information for the stab (see Section 1.3 [String Field], page 2) gives the type of the variable.

For example, the following source code:

```
char g_foo = 'c';
```

yields the following assembly code:

```
.stabs "g_foo:G2",32,0,0,0     # 32 is N_GSYM
     .global _g_foo
     .data
```

```
_g_foo:
        .byte 99
```

The address of the variable represented by the N_GSYM is not contained in the N_GSYM stab. The debugger gets this information from the external symbol for the global variable. In the example above, the .global _g_foo and _g_foo: lines tell the assembler to produce an external symbol.

Some compilers, like GCC, output N_GSYM stabs only once, where the variable is defined. Other compilers, like SunOS4 /bin/cc, output a N_GSYM stab for each compilation unit which references the variable.

4.3 Register Variables

Register variables have their own stab type, N_RSYM (C_RSYM for XCOFF), and their own symbol descriptor, 'r'. The stab's value is the number of the register where the variable data will be stored.

AIX defines a separate symbol descriptor 'd' for floating point registers. This seems unnecessary; why not just just give floating point registers different register numbers? I have not verified whether the compiler actually uses 'd'.

If the register is explicitly allocated to a global variable, but not initialized, as in:

```
register int g_bar asm ("%g5");
```

then the stab may be emitted at the end of the object file, with the other bss symbols.

4.4 Common Blocks

A common block is a statically allocated section of memory which can be referred to by several source files. It may contain several variables. I believe Fortran is the only language with this feature.

A N_BCOMM stab begins a common block and an N_ECOMM stab ends it. The only field that is significant in these two stabs is the string, which names a normal (non-debugging) symbol that gives the address of the common block. According to IBM documentation, only the N_BCOMM has the name of the common block (even though their compiler actually puts it both places).

The stabs for the members of the common block are between the N_BCOMM and the N_ECOMM; the value of each stab is the offset within the common block of that variable. IBM uses the C_ECOML stab type, and there is a corresponding N_ECOML stab type, but Sun's Fortran compiler uses N_GSYM instead. The variables within a common block use the 'V' symbol descriptor (I believe this is true of all Fortran variables). Other stabs (at least type declarations using C_DECL) can also be between the N_BCOMM and the N_ECOMM.

4.5 Static Variables

Initialized static variables are represented by the 'S' and 'V' symbol descriptors. 'S' means file scope static, and 'V' means procedure scope static. One exception: in XCOFF, IBM's xlc compiler always uses 'V', and whether it is file scope or not is distinguished by whether the stab is located within a function.

In a.out files, `N_STSYM` means the data section, `N_FUN` means the text section, and `N_LCSYM` means the bss section. For those systems with a read-only data section separate from the text section (Solaris), `N_ROSYM` means the read-only data section.

For example, the source lines:

```
static const int var_const = 5;
static int var_init = 2;
static int var_noinit;
```

yield the following stabs:

```
.stabs "var_const:S1",36,0,0,_var_const      # 36 is N_FUN
...
.stabs "var_init:S1",38,0,0,_var_init        # 38 is N_STSYM
...
.stabs "var_noinit:S1",40,0,0,_var_noinit    # 40 is N_LCSYM
```

In XCOFF files, the stab type need not indicate the section; `C_STSYM` can be used for all statics. Also, each static variable is enclosed in a static block. A `C_BSTAT` (emitted with a '`.bs`' assembler directive) symbol begins the static block; its value is the symbol number of the csect symbol whose value is the address of the static block, its section is the section of the variables in that static block, and its name is '`.bs`'. A `C_ESTAT` (emitted with a '`.es`' assembler directive) symbol ends the static block; its name is '`.es`' and its value and section are ignored.

In ECOFF files, the storage class is used to specify the section, so the stab type need not indicate the section.

In ELF files, for the SunPRO compiler version 2.0.1, symbol descriptor '`S`' means that the address is absolute (the linker relocates it) and symbol descriptor '`V`' means that the address is relative to the start of the relevant section for that compilation unit. SunPRO has plans to have the linker stop relocating stabs; I suspect that their the debugger gets the address from the corresponding ELF (not stab) symbol. I'm not sure how to find which symbol of that name is the right one. The clean way to do all this would be to have a the value of a symbol descriptor '`S`' symbol be an offset relative to the start of the file, just like everything else, but that introduces obvious compatibility problems. For more information on linker stab relocation, See Section F.2 [ELF Linker Relocation], page 62.

4.6 Fortran Based Variables

Fortran (at least, the Sun and SGI dialects of FORTRAN-77) has a feature which allows allocating arrays with `malloc`, but which avoids blurring the line between arrays and pointers the way that C does. In stabs such a variable uses the '`b`' symbol descriptor.

For example, the Fortran declarations

```
real foo, foo10(10), foo10_5(10,5)
pointer (foop, foo)
pointer (foo10p, foo10)
pointer (foo105p, foo10_5)
```

produce the stabs

```
foo:b6
foo10:bar3;1;10;6
```

```
foo10_5:bar3;1;5;ar3;1;10;6
```

In this example, `real` is type 6 and type 3 is an integral type which is the type of the subscripts of the array (probably `integer`).

The '`b`' symbol descriptor is like '`V`' in that it denotes a statically allocated symbol whose scope is local to a function; see See Section 4.5 [Statics], page 17. The value of the symbol, instead of being the address of the variable itself, is the address of a pointer to that variable. So in the above example, the value of the `foo` stab is the address of a pointer to a real, the value of the `foo10` stab is the address of a pointer to a 10-element array of reals, and the value of the `foo10_5` stab is the address of a pointer to a 5-element array of 10-element arrays of reals.

4.7 Parameters

Formal parameters to a function are represented by a stab (or sometimes two; see below) for each parameter. The stabs are in the order in which the debugger should print the parameters (i.e., the order in which the parameters are declared in the source file). The exact form of the stab depends on how the parameter is being passed.

Parameters passed on the stack use the symbol descriptor '`p`' and the `N_PSYM` symbol type (or `C_PSYM` for XCOFF). The value of the symbol is an offset used to locate the parameter on the stack; its exact meaning is machine-dependent, but on most machines it is an offset from the frame pointer.

As a simple example, the code:

```
main (argc, argv)
     int argc;
     char **argv;
```

produces the stabs:

```
.stabs "main:F1",36,0,0,_main                   # 36 is N_FUN
.stabs "argc:p1",160,0,0,68                     # 160 is N_PSYM
.stabs "argv:p20=*21=*2",160,0,0,72
```

The type definition of `argv` is interesting because it contains several type definitions. Type 21 is pointer to type 2 (char) and `argv` (type 20) is pointer to type 21.

The following symbol descriptors are also said to go with `N_PSYM`. The value of the symbol is said to be an offset from the argument pointer (I'm not sure whether this is true or not).

```
pP (<<??>>)
pF Fortran function parameter
X  (function result variable)
```

4.7.1 Passing Parameters in Registers

If the parameter is passed in a register, then traditionally there are two symbols for each argument:

```
.stabs "arg:p1" . . .       ; N_PSYM
.stabs "arg:r1" . . .       ; N_RSYM
```

Debuggers use the second one to find the value, and the first one to know that it is an argument.

Because that approach is kind of ugly, some compilers use symbol descriptor 'P' or 'R' to indicate an argument which is in a register. Symbol type C_RPSYM is used in XCOFF and N_RSYM is used otherwise. The symbol's value is the register number. 'P' and 'R' mean the same thing; the difference is that 'P' is a GNU invention and 'R' is an IBM (XCOFF) invention. As of version 4.9, GDB should handle either one.

There is at least one case where GCC uses a 'p' and 'r' pair rather than 'P'; this is where the argument is passed in the argument list and then loaded into a register.

According to the AIX documentation, symbol descriptor 'D' is for a parameter passed in a floating point register. This seems unnecessary—why not just use 'R' with a register number which indicates that it's a floating point register? I haven't verified whether the system actually does what the documentation indicates.

On the sparc and hppa, for a 'P' symbol whose type is a structure or union, the register contains the address of the structure. On the sparc, this is also true of a 'p' and 'r' pair (using Sun cc) or a 'p' symbol. However, if a (small) structure is really in a register, 'r' is used. And, to top it all off, on the hppa it might be a structure which was passed on the stack and loaded into a register and for which there is a 'p' and 'r' pair! I believe that symbol descriptor 'i' is supposed to deal with this case (it is said to mean "value parameter by reference, indirect access"; I don't know the source for this information), but I don't know details or what compilers or debuggers use it, if any (not GDB or GCC). It is not clear to me whether this case needs to be dealt with differently than parameters passed by reference (see Section 4.7.3 [Reference Parameters], page 20).

4.7.2 Storing Parameters as Local Variables

There is a case similar to an argument in a register, which is an argument that is actually stored as a local variable. Sometimes this happens when the argument was passed in a register and then the compiler stores it as a local variable. If possible, the compiler should claim that it's in a register, but this isn't always done.

If a parameter is passed as one type and converted to a smaller type by the prologue (for example, the parameter is declared as a `float`, but the calling conventions specify that it is passed as a `double`), then GCC2 (sometimes) uses a pair of symbols. The first symbol uses symbol descriptor 'p' and the type which is passed. The second symbol has the type and location which the parameter actually has after the prologue. For example, suppose the following C code appears with no prototypes involved:

```
void
subr (f)
     float f;
{
```

if `f` is passed as a double at stack offset 8, and the prologue converts it to a float in register number 0, then the stabs look like:

```
.stabs "f:p13",160,0,3,8    # 160 is N_PSYM, here 13 is double
.stabs "f:r12",64,0,3,0     # 64 is N_RSYM, here 12 is float
```

In both stabs 3 is the line number where `f` is declared (see Section 2.4 [Line Numbers], page 8).

GCC, at least on the 960, has another solution to the same problem. It uses a single '`p`' symbol descriptor for an argument which is stored as a local variable but uses `N_LSYM` instead of `N_PSYM`. In this case, the value of the symbol is an offset relative to the local variables for that function, not relative to the arguments; on some machines those are the same thing, but not on all.

On the VAX or on other machines in which the calling convention includes the number of words of arguments actually passed, the debugger (GDB at least) uses the parameter symbols to keep track of whether it needs to print nameless arguments in addition to the formal parameters which it has printed because each one has a stab. For example, in

```
extern int fprintf (FILE *stream, char *format, ...);
...
fprintf (stdout, "%d\n", x);
```

there are stabs for `stream` and `format`. On most machines, the debugger can only print those two arguments (because it has no way of knowing that additional arguments were passed), but on the VAX or other machines with a calling convention which indicates the number of words of arguments, the debugger can print all three arguments. To do so, the parameter symbol (symbol descriptor '`p`') (not necessarily '`r`' or symbol descriptor omitted symbols) needs to contain the actual type as passed (for example, `double` not `float` if it is passed as a double and converted to a float).

4.7.3 Passing Parameters by Reference

If the parameter is passed by reference (e.g., Pascal `VAR` parameters), then the symbol descriptor is '`v`' if it is in the argument list, or '`a`' if it in a register. Other than the fact that these contain the address of the parameter rather than the parameter itself, they are identical to '`p`' and '`R`', respectively. I believe '`a`' is an AIX invention; '`v`' is supported by all stabs-using systems as far as I know.

4.7.4 Passing Conformant Array Parameters

Conformant arrays are a feature of Modula-2, and perhaps other languages, in which the size of an array parameter is not known to the called function until run-time. Such parameters have two stabs: a '`x`' for the array itself, and a '`C`', which represents the size of the array. The value of the '`x`' stab is the offset in the argument list where the address of the array is stored (it this right? it is a guess); the value of the '`C`' stab is the offset in the argument list where the size of the array (in elements? in bytes?) is stored.

5 Defining Types

The examples so far have described types as references to previously defined types, or defined in terms of subranges of or pointers to previously defined types. This chapter describes the other type descriptors that may follow the '=' in a type definition.

5.1 Builtin Types

Certain types are built in (`int`, `short`, `void`, `float`, etc.); the debugger recognizes these types and knows how to handle them. Thus, don't be surprised if some of the following ways of specifying builtin types do not specify everything that a debugger would need to know about the type—in some cases they merely specify enough information to distinguish the type from other types.

The traditional way to define builtin types is convolunted, so new ways have been invented to describe them. Sun's `acc` uses special builtin type descriptors ('b' and 'R'), and IBM uses negative type numbers. GDB accepts all three ways, as of version 4.8; dbx just accepts the traditional builtin types and perhaps one of the other two formats. The following sections describe each of these formats.

5.1.1 Traditional Builtin Types

This is the traditional, convoluted method for defining builtin types. There are several classes of such type definitions: integer, floating point, and `void`.

5.1.1.1 Traditional Integer Types

Often types are defined as subranges of themselves. If the bounding values fit within an `int`, then they are given normally. For example:

```
.stabs "int:t1=r1;-2147483648;2147483647;",128,0,0,0     # 128 is N_LSYM
.stabs "char:t2=r2;0;127;",128,0,0,0
```

Builtin types can also be described as subranges of `int`:

```
.stabs "unsigned short:t6=r1;0;65535;",128,0,0,0
```

If the lower bound of a subrange is 0 and the upper bound is -1, the type is an unsigned integral type whose bounds are too big to describe in an `int`. Traditionally this is only used for `unsigned int` and `unsigned long`:

```
.stabs "unsigned int:t4=r1;0;-1;",128,0,0,0
```

For larger types, GCC 2.4.5 puts out bounds in octal, with one or more leading zeroes. In this case a negative bound consists of a number which is a 1 bit (for the sign bit) followed by a 0 bit for each bit in the number (except the sign bit), and a positive bound is one which is a 1 bit for each bit in the number (except possibly the sign bit). All known versions of dbx and GDB version 4 accept this (at least in the sense of not refusing to process the file), but GDB 3.5 refuses to read the whole file containing such symbols. So GCC 2.3.3 did not output the proper size for these types. As an example of octal bounds, the string fields of the stabs for 64 bit integer types look like:

```
long int:t3=r1;001000000000000000000000;000777777777777777777777;
long unsigned int:t5=r1;000000000000000000000000;001777777777777777777777;
```

If the lower bound of a subrange is 0 and the upper bound is negative, the type is an unsigned integral type whose size in bytes is the absolute value of the upper bound. I believe this is a Convex convention for `unsigned long long`.

If the lower bound of a subrange is negative and the upper bound is 0, the type is a signed integral type whose size in bytes is the absolute value of the lower bound. I believe this is a Convex convention for `long long`. To distinguish this from a legitimate subrange, the type should be a subrange of itself. I'm not sure whether this is the case for Convex.

5.1.1.2 Traditional Other Types

If the upper bound of a subrange is 0 and the lower bound is positive, the type is a floating point type, and the lower bound of the subrange indicates the number of bytes in the type:

```
.stabs "float:t12=r1;4;0;",128,0,0,0
.stabs "double:t13=r1;8;0;",128,0,0,0
```

However, GCC writes `long double` the same way it writes `double`, so there is no way to distinguish.

```
.stabs "long double:t14=r1;8;0;",128,0,0,0
```

Complex types are defined the same way as floating-point types; there is no way to distinguish a single-precision complex from a double-precision floating-point type.

The C `void` type is defined as itself:

```
.stabs "void:t15=15",128,0,0,0
```

I'm not sure how a boolean type is represented.

5.1.2 Defining Builtin Types Using Builtin Type Descriptors

This is the method used by Sun's `acc` for defining builtin types. These are the type descriptors to define builtin types:

`b` *signed char-flag width* ; *offset* ; *nbits* ;

Define an integral type. *signed* is '`u`' for unsigned or '`s`' for signed. *char-flag* is '`c`' which indicates this is a character type, or is omitted. I assume this is to distinguish an integral type from a character type of the same size, for example it might make sense to set it for the C type `wchar_t` so the debugger can print such variables differently (Solaris does not do this). Sun sets it on the C types `signed char` and `unsigned char` which arguably is wrong. *width* and *offset* appear to be for small objects stored in larger ones, for example a `short` in an `int` register. *width* is normally the number of bytes in the type. *offset* seems to always be zero. *nbits* is the number of bits in the type.

Note that type descriptor '`b`' used for builtin types conflicts with its use for Pascal space types (see Section 5.2 [Miscellaneous Types], page 26); they can be distinguished because the character following the type descriptor will be a digit, '`(`', or '`-`' for a Pascal space type, or '`u`' or '`s`' for a builtin type.

`w`
: Documented by AIX to define a wide character type, but their compiler actually uses negative type numbers (see Section 5.1.3 [Negative Type Numbers], page 23).

`R` *fp-type* `;` *bytes* `;`
: Define a floating point type. *fp-type* has one of the following values:

  `1 (NF_SINGLE)`
  : IEEE 32-bit (single precision) floating point format.

  `2 (NF_DOUBLE)`
  : IEEE 64-bit (double precision) floating point format.

  `3 (NF_COMPLEX)`
  `4 (NF_COMPLEX16)`
  `5 (NF_COMPLEX32)`
  : These are for complex numbers. A comment in the GDB source describes them as Fortran `complex`, `double complex`, and `complex*16`, respectively, but what does that mean? (i.e., Single precision? Double precison?).

  `6 (NF_LDOUBLE)`
  : Long double. This should probably only be used for Sun format `long double`, and new codes should be used for other floating point formats (`NF_DOUBLE` can be used if a `long double` is really just an IEEE double, of course).

  bytes is the number of bytes occupied by the type. This allows a debugger to perform some operations with the type even if it doesn't understand *fp-type*.

`g` *type-information* `;` *nbits*
: Documented by AIX to define a floating type, but their compiler actually uses negative type numbers (see Section 5.1.3 [Negative Type Numbers], page 23).

`c` *type-information* `;` *nbits*
: Documented by AIX to define a complex type, but their compiler actually uses negative type numbers (see Section 5.1.3 [Negative Type Numbers], page 23).

The C `void` type is defined as a signed integral type 0 bits long:

```
.stabs "void:t19=bs0;0;0",128,0,0,0
```

The Solaris compiler seems to omit the trailing semicolon in this case. Getting sloppy in this way is not a swift move because if a type is embedded in a more complex expression it is necessary to be able to tell where it ends.

I'm not sure how a boolean type is represented.

5.1.3 Negative Type Numbers

This is the method used in XCOFF for defining builtin types. Since the debugger knows about the builtin types anyway, the idea of negative type numbers is simply to give a special type number which indicates the builtin type. There is no stab defining these types.

There are several subtle issues with negative type numbers.

One is the size of the type. A builtin type (for example the C types `int` or `long`) might have different sizes depending on compiler options, the target architecture, the ABI, etc. This issue doesn't come up for IBM tools since (so far) they just target the RS/6000; the sizes indicated below for each size are what the IBM RS/6000 tools use. To deal with differing sizes, either define separate negative type numbers for each size (which works but requires changing the debugger, and, unless you get both AIX dbx and GDB to accept the change, introduces an incompatibility), or use a type attribute (see Section 1.3 [String Field], page 2) to define a new type with the appropriate size (which merely requires a debugger which understands type attributes, like AIX dbx or GDB). For example,

```
.stabs "boolean:t10=@s8;-16",128,0,0,0
```

defines an 8-bit boolean type, and

```
.stabs "boolean:t10=@s64;-16",128,0,0,0
```

defines a 64-bit boolean type.

A similar issue is the format of the type. This comes up most often for floating-point types, which could have various formats (particularly extended doubles, which vary quite a bit even among IEEE systems). Again, it is best to define a new negative type number for each different format; changing the format based on the target system has various problems. One such problem is that the Alpha has both VAX and IEEE floating types. One can easily imagine one library using the VAX types and another library in the same executable using the IEEE types. Another example is that the interpretation of whether a boolean is true or false can be based on the least significant bit, most significant bit, whether it is zero, etc., and different compilers (or different options to the same compiler) might provide different kinds of boolean.

The last major issue is the names of the types. The name of a given type depends *only* on the negative type number given; these do not vary depending on the language, the target system, or anything else. One can always define separate type numbers—in the following list you will see for example separate `int` and `integer*4` types which are identical except for the name. But compatibility can be maintained by not inventing new negative type numbers and instead just defining a new type with a new name. For example:

```
.stabs "CARDINAL:t10=-8",128,0,0,0
```

Here is the list of negative type numbers. The phrase *integral type* is used to mean twos-complement (I strongly suspect that all machines which use stabs use twos-complement; most machines use twos-complement these days).

`-1` `int`, 32 bit signed integral type.

`-2` `char`, 8 bit type holding a character. Both GDB and dbx on AIX treat this as signed. GCC uses this type whether `char` is signed or not, which seems like a bad idea. The AIX compiler (`xlc`) seems to avoid this type; it uses -5 instead for `char`.

`-3` `short`, 16 bit signed integral type.

`-4` `long`, 32 bit signed integral type.

`-5` `unsigned char`, 8 bit unsigned integral type.

`-6` `signed char`, 8 bit signed integral type.

-7 `unsigned short`, 16 bit unsigned integral type.

-8 `unsigned int`, 32 bit unsigned integral type.

-9 `unsigned`, 32 bit unsigned integral type.

-10 `unsigned long`, 32 bit unsigned integral type.

-11 `void`, type indicating the lack of a value.

-12 `float`, IEEE single precision.

-13 `double`, IEEE double precision.

-14 `long double`, IEEE double precision. The compiler claims the size will increase in a future release, and for binary compatibility you have to avoid using `long double`. I hope when they increase it they use a new negative type number.

-15 `integer`. 32 bit signed integral type.

-16 `boolean`. 32 bit type. GDB and GCC assume that zero is false, one is true, and other values have unspecified meaning. I hope this agrees with how the IBM tools use the type.

-17 `short real`. IEEE single precision.

-18 `real`. IEEE double precision.

-19 `stringptr`. See Section 5.6 [Strings], page 29.

-20 `character`, 8 bit unsigned character type.

-21 `logical*1`, 8 bit type. This Fortran type has a split personality in that it is used for boolean variables, but can also be used for unsigned integers. 0 is false, 1 is true, and other values are non-boolean.

-22 `logical*2`, 16 bit type. This Fortran type has a split personality in that it is used for boolean variables, but can also be used for unsigned integers. 0 is false, 1 is true, and other values are non-boolean.

-23 `logical*4`, 32 bit type. This Fortran type has a split personality in that it is used for boolean variables, but can also be used for unsigned integers. 0 is false, 1 is true, and other values are non-boolean.

-24 `logical`, 32 bit type. This Fortran type has a split personality in that it is used for boolean variables, but can also be used for unsigned integers. 0 is false, 1 is true, and other values are non-boolean.

-25 `complex`. A complex type consisting of two IEEE single-precision floating point values.

-26 `complex`. A complex type consisting of two IEEE double-precision floating point values.

-27 `integer*1`, 8 bit signed integral type.

-28 `integer*2`, 16 bit signed integral type.

-29 `integer*4`, 32 bit signed integral type.

`-30` `wchar`. Wide character, 16 bits wide, unsigned (what format? Unicode?).

`-31` `long long`, 64 bit signed integral type.

`-32` `unsigned long long`, 64 bit unsigned integral type.

`-33` `logical*8`, 64 bit unsigned integral type.

`-34` `integer*8`, 64 bit signed integral type.

5.2 Miscellaneous Types

`b` *type-information* `;` *bytes*

Pascal space type. This is documented by IBM; what does it mean?

This use of the '`b`' type descriptor can be distinguished from its use for builtin integral types (see Section 5.1.2 [Builtin Type Descriptors], page 22) because the character following the type descriptor is always a digit, '`(`', or '`-`'.

`B` *type-information*

A volatile-qualified version of *type-information*. This is a Sun extension. References and stores to a variable with a volatile-qualified type must not be optimized or cached; they must occur as the user specifies them.

`d` *type-information*

File of type *type-information*. As far as I know this is only used by Pascal.

`k` *type-information*

A const-qualified version of *type-information*. This is a Sun extension. A variable with a const-qualified type cannot be modified.

`M` *type-information* `;` *length*

Multiple instance type. The type seems to composed of *length* repetitions of *type-information*, for example `character*3` is represented by '`M-2;3`', where '`-2`' is a reference to a character type (see Section 5.1.3 [Negative Type Numbers], page 23). I'm not sure how this differs from an array. This appears to be a Fortran feature. *length* is a bound, like those in range types; see Section 5.4 [Subranges], page 27.

`S` *type-information*

Pascal set type. *type-information* must be a small type such as an enumeration or a subrange, and the type is a bitmask whose length is specified by the number of elements in *type-information*.

In CHILL, if it is a bitstring instead of a set, also use the '`S`' type attribute (see Section 1.3 [String Field], page 2).

`*` *type-information*

Pointer to *type-information*.

5.3 Cross-References to Other Types

A type can be used before it is defined; one common way to deal with that situation is just to use a type reference to a type which has not yet been defined.

Another way is with the '`x`' type descriptor, which is followed by '`s`' for a structure tag, '`u`' for a union tag, or '`e`' for a enumerator tag, followed by the name of the tag, followed by '`:`'. If the name contains '`::`' between a '`<`' and '`>`' pair (for C++ templates), such a '`::`' does not end the name—only a single '`:`' ends the name; see Section 7.2 [Nested Symbols], page 37.

For example, the following C declarations:

```
struct foo;
struct foo *bar;
```

produce:

```
.stabs "bar:G16=*17=xsfoo:",32,0,0,0
```

Not all debuggers support the '`x`' type descriptor, so on some machines GCC does not use it. I believe that for the above example it would just emit a reference to type 17 and never define it, but I haven't verified that.

Modula-2 imported types, at least on AIX, use the '`i`' type descriptor, which is followed by the name of the module from which the type is imported, followed by '`:`', followed by the name of the type. There is then optionally a comma followed by type information for the type. This differs from merely naming the type (see Section 5.9 [Typedefs], page 31) in that it identifies the module; I don't understand whether the name of the type given here is always just the same as the name we are giving it, or whether this type descriptor is used with a nameless stab (see Section 1.3 [String Field], page 2), or what. The symbol ends with '`;`'.

5.4 Subrange Types

The '`r`' type descriptor defines a type as a subrange of another type. It is followed by type information for the type of which it is a subrange, a semicolon, an integral lower bound, a semicolon, an integral upper bound, and a semicolon. The AIX documentation does not specify the trailing semicolon, in an effort to specify array indexes more cleanly, but a subrange which is not an array index has always included a trailing semicolon (see Section 5.5 [Arrays], page 28).

Instead of an integer, either bound can be one of the following:

`A` *offset*
: The bound is passed by reference on the stack at offset *offset* from the argument list. See Section 4.7 [Parameters], page 18, for more information on such offsets.

`T` *offset*
: The bound is passed by value on the stack at offset *offset* from the argument list.

`a` *register-number*
: The bound is pased by reference in register number *register-number*.

`t` *register-number*
: The bound is passed by value in register number *register-number*.

`J` There is no bound.

Subranges are also used for builtin types; see Section 5.1.1 [Traditional Builtin Types], page 21.

5.5 Array Types

Arrays use the '`a`' type descriptor. Following the type descriptor is the type of the index and the type of the array elements. If the index type is a range type, it ends in a semicolon; otherwise (for example, if it is a type reference), there does not appear to be any way to tell where the types are separated. In an effort to clean up this mess, IBM documents the two types as being separated by a semicolon, and a range type as not ending in a semicolon (but this is not right for range types which are not array indexes, see Section 5.4 [Subranges], page 27). I think probably the best solution is to specify that a semicolon ends a range type, and that the index type and element type of an array are separated by a semicolon, but that if the index type is a range type, the extra semicolon can be omitted. GDB (at least through version 4.9) doesn't support any kind of index type other than a range anyway; I'm not sure about dbx.

It is well established, and widely used, that the type of the index, unlike most types found in the stabs, is merely a type definition, not type information (see Section 1.3 [String Field], page 2) (that is, it need not start with '*type-number*=' if it is defining a new type). According to a comment in GDB, this is also true of the type of the array elements; it gives '`ar1;1;10;ar1;1;10;4`' as a legitimate way to express a two dimensional array. According to AIX documentation, the element type must be type information. GDB accepts either.

The type of the index is often a range type, expressed as the type descriptor '`r`' and some parameters. It defines the size of the array. In the example below, the range '`r1;0;2;`' defines an index type which is a subrange of type 1 (integer), with a lower bound of 0 and an upper bound of 2. This defines the valid range of subscripts of a three-element C array.

For example, the definition:

```
char char_vec[3] = {'a','b','c'};
```

produces the output:

```
.stabs "char_vec:G19=ar1;0;2;2",32,0,0,0
     .global _char_vec
     .align 4
_char_vec:
     .byte 97
     .byte 98
     .byte 99
```

If an array is *packed*, the elements are spaced more closely than normal, saving memory at the expense of speed. For example, an array of 3-byte objects might, if unpacked, have each element aligned on a 4-byte boundary, but if packed, have no padding. One way to specify that something is packed is with type attributes (see Section 1.3 [String Field], page 2). In the case of arrays, another is to use the '`P`' type descriptor instead of '`a`'. Other than specifying a packed array, '`P`' is identical to '`a`'.

An open array is represented by the '`A`' type descriptor followed by type information specifying the type of the array elements.

An N-dimensional dynamic array is represented by

`D` *dimensions* `;` *type-information*

dimensions is the number of dimensions; *type-information* specifies the type of the array elements.

A subarray of an N-dimensional array is represented by

`E` *dimensions* `;` *type-information*

dimensions is the number of dimensions; *type-information* specifies the type of the array elements.

5.6 Strings

Some languages, like C or the original Pascal, do not have string types, they just have related things like arrays of characters. But most Pascals and various other languages have string types, which are indicated as follows:

`n` *type-information* `;` *bytes*
: *bytes* is the maximum length. I'm not sure what *type-information* is; I suspect that it means that this is a string of *type-information* (thus allowing a string of integers, a string of wide characters, etc., as well as a string of characters). Not sure what the format of this type is. This is an AIX feature.

`z` *type-information* `;` *bytes*
: Just like '`n`' except that this is a gstring, not an ordinary string. I don't know the difference.

`N`
: Pascal Stringptr. What is this? This is an AIX feature.

Languages, such as CHILL which have a string type which is basically just an array of characters use the '`S`' type attribute (see Section 1.3 [String Field], page 2).

5.7 Enumerations

Enumerations are defined with the '`e`' type descriptor.

The source line below declares an enumeration type at file scope. The type definition is located after the `N_RBRAC` that marks the end of the previous procedure's block scope, and before the `N_FUN` that marks the beginning of the next procedure's block scope. Therefore it does not describe a block local symbol, but a file local one.

The source line:

```
enum e_places {first,second=3,last};
```

generates the following stab:

```
.stabs "e_places:T22=efirst:0,second:3,last:4,;",128,0,0,0
```

The symbol descriptor ('`T`') says that the stab describes a structure, enumeration, or union tag. The type descriptor '`e`', following the '`22=`' of the type definition narrows it down to an enumeration type. Following the '`e`' is a list of the elements of the enumeration. The format is '*name*`:`*value*`,`'. The list of elements ends with '`;`'. The fact that *value* is specified as an integer can cause problems if the value is large. GCC 2.5.2 tries to output it in octal in that case with a leading zero, which is probably a good thing, although GDB

4.11 supports octal only in cases where decimal is perfectly good. Negative decimal values are supported by both GDB and dbx.

There is no standard way to specify the size of an enumeration type; it is determined by the architecture (normally all enumerations types are 32 bits). Type attributes can be used to specify an enumeration type of another size for debuggers which support them; see Section 1.3 [String Field], page 2.

Enumeration types are unusual in that they define symbols for the enumeration values (`first`, `second`, and `third` in the above example), and even though these symbols are visible in the file as a whole (rather than being in a more local namespace like structure member names), they are defined in the type definition for the enumeration type rather than each having their own symbol. In order to be fast, GDB will only get symbols from such types (in its initial scan of the stabs) if the type is the first thing defined after a '`T`' or '`t`' symbol descriptor (the above example fulfills this requirement). If the type does not have a name, the compiler should emit it in a nameless stab (see Section 1.3 [String Field], page 2); GCC does this.

5.8 Structures

The encoding of structures in stabs can be shown with an example.

The following source code declares a structure tag and defines an instance of the structure in global scope. Then a `typedef` equates the structure tag with a new type. Seperate stabs are generated for the structure tag, the structure `typedef`, and the structure instance. The stabs for the tag and the `typedef` are emited when the definitions are encountered. Since the structure elements are not initialized, the stab and code for the structure variable itself is located at the end of the program in the bss section.

```
struct s_tag {
  int   s_int;
  float s_float;
  char  s_char_vec[8];
  struct s_tag* s_next;
} g_an_s;

typedef struct s_tag s_typedef;
```

The structure tag has an `N_LSYM` stab type because, like the enumeration, the symbol has file scope. Like the enumeration, the symbol descriptor is '`T`', for enumeration, structure, or tag type. The type descriptor '`s`' following the '`16=`' of the type definition narrows the symbol type to structure.

Following the '`s`' type descriptor is the number of bytes the structure occupies, followed by a description of each structure element. The structure element descriptions are of the form *name:type, bit offset from the start of the struct, number of bits in the element.*

```
# 128 is N_LSYM
.stabs "s_tag:T16=s20s_int:1,0,32;s_float:12,32,32;
        s_char_vec:17=ar1;0;7;2,64,64;s_next:18=*16,128,32;;",128,0,0,0
```

In this example, the first two structure elements are previously defined types. For these, the type following the '*name*`:`' part of the element description is a simple type reference. The other two structure elements are new types. In this case there is a type definition embedded after the '*name*:'. The type definition for the array element looks just like a type definition for a standalone array. The `s_next` field is a pointer to the same kind of structure that the field is an element of. So the definition of structure type 16 contains a type definition for an element which is a pointer to type 16.

If a field is a static member (this is a C++ feature in which a single variable appears to be a field of every structure of a given type) it still starts out with the field name, a colon, and the type, but then instead of a comma, bit position, comma, and bit size, there is a colon followed by the name of the variable which each such field refers to.

If the structure has methods (a C++ feature), they follow the non-method fields; see Chapter 7 [Cplusplus], page 37.

5.9 Giving a Type a Name

To give a type a name, use the '`t`' symbol descriptor. The type is specified by the type information (see Section 1.3 [String Field], page 2) for the stab. For example,

```
.stabs "s_typedef:t16",128,0,0,0      # 128 is N_LSYM
```

specifies that `s_typedef` refers to type number 16. Such stabs have symbol type `N_LSYM` (or `C_DECL` for XCOFF). (The Sun documentation mentions using `N_GSYM` in some cases).

If you are specifying the tag name for a structure, union, or enumeration, use the '`T`' symbol descriptor instead. I believe C is the only language with this feature.

If the type is an opaque type (I believe this is a Modula-2 feature), AIX provides a type descriptor to specify it. The type descriptor is '`o`' and is followed by a name. I don't know what the name means—is it always the same as the name of the type, or is this type descriptor used with a nameless stab (see Section 1.3 [String Field], page 2)? There optionally follows a comma followed by type information which defines the type of this type. If omitted, a semicolon is used in place of the comma and the type information, and the type is much like a generic pointer type—it has a known size but little else about it is specified.

5.10 Unions

```
union u_tag {
  int  u_int;
  float u_float;
  char* u_char;
} an_u;
```

This code generates a stab for a union tag and a stab for a union variable. Both use the `N_LSYM` stab type. If a union variable is scoped locally to the procedure in which it is defined, its stab is located immediately preceding the `N_LBRAC` for the procedure's block start.

The stab for the union tag, however, is located preceding the code for the procedure in which it is defined. The stab type is `N_LSYM`. This would seem to imply that the union type is file scope, like the struct type `s_tag`. This is not true. The contents and position of the stab for `u_type` do not convey any infomation about its procedure local scope.

```
# 128 is N_LSYM
.stabs "u_tag:T23=u4u_int:1,0,32;u_float:12,0,32;u_char:21,0,32;;",
       128,0,0,0
```

The symbol descriptor 'T', following the 'name:' means that the stab describes an enumeration, structure, or union tag. The type descriptor 'u', following the '23=' of the type definition, narrows it down to a union type definition. Following the 'u' is the number of bytes in the union. After that is a list of union element descriptions. Their format is *name:type, bit offset into the union, number of bytes for the element;*.

The stab for the union variable is:

```
.stabs "an_u:23",128,0,0,-20     # 128 is N_LSYM
```

'-20' specifies where the variable is stored (see Section 4.1 [Stack Variables], page 15).

5.11 Function Types

Various types can be defined for function variables. These types are not used in defining functions (see Section 2.5 [Procedures], page 9); they are used for things like pointers to functions.

The simple, traditional, type is type descriptor 'f' is followed by type information for the return type of the function, followed by a semicolon.

This does not deal with functions for which the number and types of the parameters are part of the type, as in Modula-2 or ANSI C. AIX provides extensions to specify these, using the 'f', 'F', 'p', and 'R' type descriptors.

First comes the type descriptor. If it is 'f' or 'F', this type involves a function rather than a procedure, and the type information for the return type of the function follows, followed by a comma. Then comes the number of parameters to the function and a semicolon. Then, for each parameter, there is the name of the parameter followed by a colon (this is only present for type descriptors 'R' and 'F' which represent Pascal function or procedure parameters), type information for the parameter, a comma, 0 if passed by reference or 1 if passed by value, and a semicolon. The type definition ends with a semicolon.

For example, this variable definition:

```
int (*g_pf)();
```

generates the following code:

```
.stabs "g_pf:G24=*25=f1",32,0,0,0
    .common _g_pf,4,"bss"
```

The variable defines a new type, 24, which is a pointer to another new type, 25, which is a function returning `int`.

6 Symbol Information in Symbol Tables

This chapter describes the format of symbol table entries and how stab assembler directives map to them. It also describes the transformations that the assembler and linker make on data from stabs.

6.1 Symbol Table Format

Each time the assembler encounters a stab directive, it puts each field of the stab into a corresponding field in a symbol table entry of its output file. If the stab contains a string field, the symbol table entry for that stab points to a string table entry containing the string data from the stab. Assembler labels become relocatable addresses. Symbol table entries in a.out have the format:

```
struct internal_nlist {
  unsigned long n_strx;         /* index into string table of name */
  unsigned char n_type;         /* type of symbol */
  unsigned char n_other;        /* misc info (usually empty) */
  unsigned short n_desc;        /* description field */
  bfd_vma n_value;              /* value of symbol */
};
```

If the stab has a string, the `n_strx` field holds the offset in bytes of the string within the string table. The string is terminated by a NUL character. If the stab lacks a string (for example, it was produced by a `.stabn` or `.stabd` directive), the `n_strx` field is zero.

Symbol table entries with `n_type` field values greater than 0x1f originated as stabs generated by the compiler (with one random exception). The other entries were placed in the symbol table of the executable by the assembler or the linker.

6.2 Transformations on Symbol Tables

The linker concatenates object files and does fixups of externally defined symbols.

You can see the transformations made on stab data by the assembler and linker by examining the symbol table after each pass of the build. To do this, use '`nm -ap`', which dumps the symbol table, including debugging information, unsorted. For stab entries the columns are: *value*, *other*, *desc*, *type*, *string*. For assembler and linker symbols, the columns are: *value*, *type*, *string*.

The low 5 bits of the stab type tell the linker how to relocate the value of the stab. Thus for stab types like `N_RSYM` and `N_LSYM`, where the value is an offset or a register number, the low 5 bits are `N_ABS`, which tells the linker not to relocate the value.

Where the value of a stab contains an assembly language label, it is transformed by each build step. The assembler turns it into a relocatable address and the linker turns it into an absolute address.

6.2.1 Transformations on Static Variables

This source line defines a static variable at file scope:

```
static int s_g_repeat
```

The following stab describes the symbol:

```
.stabs "s_g_repeat:S1",38,0,0,_s_g_repeat
```

The assembler transforms the stab into this symbol table entry in the '.o' file. The location is expressed as a data segment offset.

```
00000084 - 00 0000 STSYM s_g_repeat:S1
```

In the symbol table entry from the executable, the linker has made the relocatable address absolute.

```
0000e00c - 00 0000 STSYM s_g_repeat:S1
```

6.2.2 Transformations on Global Variables

Stabs for global variables do not contain location information. In this case, the debugger finds location information in the assembler or linker symbol table entry describing the variable. The source line:

```
char g_foo = 'c';
```

generates the stab:

```
.stabs "g_foo:G2",32,0,0,0
```

The variable is represented by two symbol table entries in the object file (see below). The first one originated as a stab. The second one is an external symbol. The upper case 'D' signifies that the `n_type` field of the symbol table contains 7, `N_DATA` with local linkage. The stab's value is zero since the value is not used for `N_GSYM` stabs. The value of the linker symbol is the relocatable address corresponding to the variable.

```
00000000 - 00 0000  GSYM g_foo:G2
00000080 D _g_foo
```

These entries as transformed by the linker. The linker symbol table entry now holds an absolute address:

```
00000000 - 00 0000  GSYM g_foo:G2
...
0000e008 D _g_foo
```

6.2.3 Transformations of Stabs in separate sections

For object file formats using stabs in separate sections (see Appendix F [Stab Sections], page 61), use `objdump --stabs` instead of `nm` to show the stabs in an object or executable file. `objdump` is a GNU utility; Sun does not provide any equivalent.

The following example is for a stab whose value is an address is relative to the compilation unit (see Section F.2 [ELF Linker Relocation], page 62). For example, if the source line

```
static int ld = 5;
```

appears within a function, then the assembly language output from the compiler contains:

```
.Ddata.data:
...
        .stabs "ld:V(0,3)",0x26,0,4,.L18-Ddata.data    # 0x26 is N_STSYM
...
```

```
.L18:
        .align 4
        .word 0x5
```

Because the value is formed by subtracting one symbol from another, the value is absolute, not relocatable, and so the object file contains

```
Symnum n_type n_othr n_desc n_value  n_strx String
31     STSYM  0      4      00000004 680    ld:V(0,3)
```

without any relocations, and the executable file also contains

```
Symnum n_type n_othr n_desc n_value  n_strx String
31     STSYM  0      4      00000004 680    ld:V(0,3)
```

7 GNU C++ Stabs

7.1 C++ Class Names

In C++, a class name which is declared with `class`, `struct`, or `union`, is not only a tag, as in C, but also a type name. Thus there should be stabs with both '`t`' and '`T`' symbol descriptors (see Section 5.9 [Typedefs], page 31).

To save space, there is a special abbreviation for this case. If the '`T`' symbol descriptor is followed by '`t`', then the stab defines both a type name and a tag.

For example, the C++ code

```
struct foo {int x;};
```

can be represented as either

```
.stabs "foo:T19=s4x:1,0,32;;",128,0,0,0       # 128 is N_LSYM
.stabs "foo:t19",128,0,0,0
```

or

```
.stabs "foo:Tt19=s4x:1,0,32;;",128,0,0,0
```

7.2 Defining a Symbol Within Another Type

In C++, a symbol (such as a type name) can be defined within another type.

In stabs, this is sometimes represented by making the name of a symbol which contains '`::`'. Such a pair of colons does not end the name of the symbol, the way a single colon would (see Section 1.3 [String Field], page 2). I'm not sure how consistently used or well thought out this mechanism is. So that a pair of colons in this position always has this meaning, '`:`' cannot be used as a symbol descriptor.

For example, if the string for a stab is '`foo::bar::baz:t5=*6`', then `foo::bar::baz` is the name of the symbol, '`t`' is the symbol descriptor, and '`5=*6`' is the type information.

7.3 Basic Types For C++

<< the examples that follow are based on a01.C >>

C++ adds two more builtin types to the set defined for C. These are the unknown type and the vtable record type. The unknown type, type 16, is defined in terms of itself like the void type.

The vtable record type, type 17, is defined as a structure type and then as a structure tag. The structure has four fields: delta, index, pfn, and delta2. pfn is the function pointer.

<< In boilerplate $vtbl_ptr_type, what are the fields delta, index, and delta2 used for? >>

This basic type is present in all C++ programs even if there are no virtual methods defined.

```
.stabs "struct_name:sym_desc(type)type_def(17)=type_desc(struct)struct_bytes(8)
        elem_name(delta):type_ref(short int),bit_offset(0),field_bits(16);
        elem_name(index):type_ref(short int),bit_offset(16),field_bits(16);
        elem_name(pfn):type_def(18)=type_desc(ptr to)type_ref(void),
                                    bit_offset(32),field_bits(32);
        elem_name(delta2):type_def(short int);bit_offset(32),field_bits(16);;"
        N_LSYM, NIL, NIL
```

```
.stabs "$vtbl_ptr_type:t17=s8
        delta:6,0,16;index:6,16,16;pfn:18=*15,32,32;delta2:6,32,16;;"
        ,128,0,0,0
```

```
.stabs "name:sym_dec(struct tag)type_ref($vtbl_ptr_type)",N_LSYM,NIL,NIL,NIL
```

```
.stabs "$vtbl_ptr_type:T17",128,0,0,0
```

7.4 Simple Class Definition

The stabs describing C++ language features are an extension of the stabs describing C. Stabs representing C++ class types elaborate extensively on the stab format used to describe structure types in C. Stabs representing class type variables look just like stabs representing C language variables.

Consider the following very simple class definition.

```
class baseA {
public:
        int Adat;
        int Ameth(int in, char other);
};
```

The class `baseA` is represented by two stabs. The first stab describes the class as a structure type. The second stab describes a structure tag of the class type. Both stabs are of stab type `N_LSYM`. Since the stab is not located between an `N_FUN` and an `N_LBRAC` stab this indicates that the class is defined at file scope. If it were, then the `N_LSYM` would signify a local variable.

A stab describing a C++ class type is similar in format to a stab describing a C struct, with each class member shown as a field in the structure. The part of the struct format describing fields is expanded to include extra information relevent to C++ class members. In addition, if the class has multiple base classes or virtual functions the struct format outside of the field parts is also augmented.

In this simple example the field part of the C++ class stab representing member data looks just like the field part of a C struct stab. The section on protections describes how its format is sometimes extended for member data.

The field part of a C++ class stab representing a member function differs substantially from the field part of a C struct stab. It still begins with '`name:`' but then goes on to define a new type number for the member function, describe its return type, its argument types, its protection level, any qualifiers applied to the method definition, and whether the method is virtual or not. If the method is virtual then the method description goes on to give the vtable index of the method, and the type number of the first base class defining the method.

When the field name is a method name it is followed by two colons rather than one. This is followed by a new type definition for the method. This is a number followed by an equal sign and the type of the method. Normally this will be a type declared using the '#' type descriptor; see Section 7.7 [Method Type Descriptor], page 40; static member functions are declared using the 'f' type descriptor instead; see Section 5.11 [Function Types], page 32.

The format of an overloaded operator method name differs from that of other methods. It is 'op$::*operator-name*.' where *operator-name* is the operator name such as '+' or '+='. The name ends with a period, and any characters except the period can occur in the *operator-name* string.

The next part of the method description represents the arguments to the method, preceeded by a colon and ending with a semi-colon. The types of the arguments are expressed in the same way argument types are expressed in C++ name mangling. In this example an `int` and a `char` map to 'ic'.

This is followed by a number, a letter, and an asterisk or period, followed by another semicolon. The number indicates the protections that apply to the member function. Here the 2 means public. The letter encodes any qualifier applied to the method definition. In this case, 'A' means that it is a normal function definition. The dot shows that the method is not virtual. The sections that follow elaborate further on these fields and describe the additional information present for virtual methods.

```
.stabs "class_name:sym_desc(type)type_def(20)=type_desc(struct)struct_bytes(4)
        field_name(Adat):type(int),bit_offset(0),field_bits(32);

        method_name(Ameth)::type_def(21)=type_desc(method)return_type(int);
        :arg_types(int char);
        protection(public)qualifier(normal)virtual(no);;"
        N_LSYM,NIL,NIL,NIL
```

```
.stabs "baseA:t20=s4Adat:1,0,32;Ameth::21=##1;:ic;2A.;;",128,0,0,0

.stabs "class_name:sym_desc(struct tag)",N_LSYM,NIL,NIL,NIL

.stabs "baseA:T20",128,0,0,0
```

7.5 Class Instance

As shown above, describing even a simple C++ class definition is accomplished by massively extending the stab format used in C to describe structure types. However, once the class is defined, C stabs with no modifications can be used to describe class instances. The following source:

```
main () {
        baseA AbaseA;
}
```

yields the following stab describing the class instance. It looks no different from a standard C stab describing a local variable.

```
.stabs "name:type_ref(baseA)", N_LSYM, NIL, NIL, frame_ptr_offset
```

```
.stabs "AbaseA:20",128,0,0,-20
```

7.6 Method Definition

The class definition shown above declares Ameth. The C++ source below defines Ameth:

```
int
baseA::Ameth(int in, char other)
{
        return in;
};
```

This method definition yields three stabs following the code of the method. One stab describes the method itself and following two describe its parameters. Although there is only one formal argument all methods have an implicit argument which is the `this` pointer. The `this` pointer is a pointer to the object on which the method was called. Note that the method name is mangled to encode the class name and argument types. Name mangling is described in the ARM (*The Annotated C++ Reference Manual*, by Ellis and Stroustrup, ISBN 0-201-51459-1); '`gpcompare.texi`' in Cygnus GCC distributions describes the differences between GNU mangling and ARM mangling.

```
.stabs "name:symbol_desriptor(global function)return_type(int)",
        N_FUN, NIL, NIL, code_addr_of_method_start

.stabs "Ameth__5baseAic:F1",36,0,0,_Ameth__5baseAic
```

Here is the stab for the `this` pointer implicit argument. The name of the `this` pointer is always `this`. Type 19, the `this` pointer is defined as a pointer to type 20, `baseA`, but a stab defining `baseA` has not yet been emited. Since the compiler knows it will be emited shortly, here it just outputs a cross reference to the undefined symbol, by prefixing the symbol name with '`xs`'.

```
.stabs "name:sym_desc(register param)type_def(19)=
        type_desc(ptr to)type_ref(baseA)=
        type_desc(cross-reference to)baseA:",N_RSYM,NIL,NIL,register_number

.stabs "this:P19=*20=xsbaseA:",64,0,0,8
```

The stab for the explicit integer argument looks just like a parameter to a C function. The last field of the stab is the offset from the argument pointer, which in most systems is the same as the frame pointer.

```
.stabs "name:sym_desc(value parameter)type_ref(int)",
        N_PSYM,NIL,NIL,offset_from_arg_ptr

.stabs "in:p1",160,0,0,72
```

<< The examples that follow are based on A1.C >>

7.7 The '#' Type Descriptor

This is used to describe a class method. This is a function which takes an extra argument as its first argument, for the `this` pointer.

If the '`#`' is immediately followed by another '`#`', the second one will be followed by the return type and a semicolon. The class and argument types are not specified, and must be determined by demangling the name of the method if it is available.

Otherwise, the single '#' is followed by the class type, a comma, the return type, a comma, and zero or more parameter types separated by commas. The list of arguments is terminated by a semicolon. In the debugging output generated by gcc, a final argument type of `void` indicates a method which does not take a variable number of arguments. If the final argument type of `void` does not appear, the method was declared with an ellipsis.

Note that although such a type will normally be used to describe fields in structures, unions, or classes, for at least some versions of the compiler it can also be used in other contexts.

7.8 The '@' Type Descriptor

The '@' type descriptor is for a member (class and variable) type. It is followed by type information for the offset basetype, a comma, and type information for the type of the field being pointed to. (FIXME: this is acknowledged to be gibberish. Can anyone say what really goes here?).

Note that there is a conflict between this and type attributes (see Section 1.3 [String Field], page 2); both use type descriptor '@'. Fortunately, the '@' type descriptor used in this C++ sense always will be followed by a digit, '(', or '-', and type attributes never start with those things.

7.9 Protections

In the simple class definition shown above all member data and functions were publicly accessable. The example that follows contrasts public, protected and privately accessable fields and shows how these protections are encoded in C++ stabs.

If the character following the '*field-name*:' part of the string is '/', then the next character is the visibility. '0' means private, '1' means protected, and '2' means public. Debuggers should ignore visibility characters they do not recognize, and assume a reasonable default (such as public) (GDB 4.11 does not, but this should be fixed in the next GDB release). If no visibility is specified the field is public. The visibility '9' means that the field has been optimized out and is public (there is no way to specify an optimized out field with a private or protected visibility). Visibility '9' is not supported by GDB 4.11; this should be fixed in the next GDB release.

The following C++ source:

```
class vis {
private:
        int   priv;
protected:
        char  prot;
public:
        float pub;
};
```

generates the following stab:

```
# 128 is N_LSYM
.stabs "vis:T19=s12priv:/01,0,32;prot:/12,32,8;pub:12,64,32;;",128,0,0,0
```

'`vis:T19=s12`' indicates that type number 19 is a 12 byte structure named `vis` The `priv` field has public visibility ('`/0`'), type int ('`1`'), and offset and size '`,0,32;`'. The `prot` field has protected visibility ('`/1`'), type char ('`2`') and offset and size '`,32,8;`'. The `pub` field has type float ('`12`'), and offset and size '`,64,32;`'.

Protections for member functions are signified by one digit embeded in the field part of the stab describing the method. The digit is 0 if private, 1 if protected and 2 if public. Consider the C++ class definition below:

```
class all_methods {
private:
        int   priv_meth(int in){return in;};
protected:
        char  protMeth(char in){return in;};
public:
        float pubMeth(float in){return in;};
};
```

It generates the following stab. The digit in question is to the left of an '`A`' in each case. Notice also that in this case two symbol descriptors apply to the class name struct tag and struct type.

```
.stabs "class_name:sym_desc(struct tag&type)type_def(21)=
     sym_desc(struct)struct_bytes(1)
     meth_name::type_def(22)=sym_desc(method)returning(int);
     :args(int);protection(private)modifier(normal)virtual(no);
     meth_name::type_def(23)=sym_desc(method)returning(char);
     :args(char);protection(protected)modifier(normal)virual(no);
     meth_name::type_def(24)=sym_desc(method)returning(float);
     :args(float);protection(public)modifier(normal)virtual(no);;",
     N_LSYM,NIL,NIL,NIL
```

```
.stabs "all_methods:Tt21=s1priv_meth::22=##1;:i;0A.;protMeth::23=##2;:c;1A.;
        pubMeth::24=##12;:f;2A.;;",128,0,0,0
```

7.10 Method Modifiers (`const`, `volatile`, `const volatile`)

<< based on a6.C >>

In the class example described above all the methods have the normal modifier. This method modifier information is located just after the protection information for the method. This field has four possible character values. Normal methods use '`A`', const methods use '`B`', volatile methods use '`C`', and const volatile methods use '`D`'. Consider the class definition below:

```
class A {
public:
        int ConstMeth (int arg) const { return arg; };
        char VolatileMeth (char arg) volatile { return arg; };
        float ConstVolMeth (float arg) const volatile {return arg; };
};
```

This class is described by the following stab:

```
.stabs "class(A):sym_desc(struct)type_def(20)=type_desc(struct)struct_bytes(1)
```

```
meth_name(ConstMeth)::type_def(21)sym_desc(method)
returning(int);:arg(int);protection(public)modifier(const)virtual(no);
meth_name(VolatileMeth)::type_def(22)=sym_desc(method)
returning(char);:arg(char);protection(public)modifier(volatile)virt(no)
meth_name(ConstVolMeth)::type_def(23)=sym_desc(method)
returning(float);:arg(float);protection(public)modifer(const volatile)
virtual(no);;", ...
```

```
.stabs "A:T20=s1ConstMeth::21=##1;:i;2B.;VolatileMeth::22=##2;:c;2C.;
                ConstVolMeth::23=##12;:f;2D.;;",128,0,0,0
```

7.11 Virtual Methods

<< The following examples are based on a4.C >>

The presence of virtual methods in a class definition adds additional data to the class description. The extra data is appended to the description of the virtual method and to the end of the class description. Consider the class definition below:

```
class A {
public:
        int Adat;
        virtual int A_virt (int arg) { return arg; };
};
```

This results in the stab below describing class A. It defines a new type (20) which is an 8 byte structure. The first field of the class struct is 'Adat', an integer, starting at structure offset 0 and occupying 32 bits.

The second field in the class struct is not explicitly defined by the C++ class definition but is implied by the fact that the class contains a virtual method. This field is the vtable pointer. The name of the vtable pointer field starts with '$vf' and continues with a type reference to the class it is part of. In this example the type reference for class A is 20 so the name of its vtable pointer field is '$vf20', followed by the usual colon.

Next there is a type definition for the vtable pointer type (21). This is in turn defined as a pointer to another new type (22).

Type 22 is the vtable itself, which is defined as an array, indexed by a range of integers between 0 and 1, and whose elements are of type 17. Type 17 was the vtable record type defined by the boilerplate C++ type definitions, as shown earlier.

The bit offset of the vtable pointer field is 32. The number of bits in the field are not specified when the field is a vtable pointer.

Next is the method definition for the virtual member function `A_virt`. Its description starts out using the same format as the non-virtual member functions described above, except instead of a dot after the 'A' there is an asterisk, indicating that the function is virtual. Since is is virtual some addition information is appended to the end of the method description.

The first number represents the vtable index of the method. This is a 32 bit unsigned number with the high bit set, followed by a semi-colon.

The second number is a type reference to the first base class in the inheritence hierarchy defining the virtual member function. In this case the class stab describes a base class so the virtual function is not overriding any other definition of the method. Therefore the reference is to the type number of the class that the stab is describing (20).

This is followed by three semi-colons. One marks the end of the current sub-section, one marks the end of the method field, and the third marks the end of the struct definition.

For classes containing virtual functions the very last section of the string part of the stab holds a type reference to the first base class. This is preceeded by '~%' and followed by a final semi-colon.

```
.stabs "class_name(A):type_def(20)=sym_desc(struct)struct_bytes(8)
        field_name(Adat):type_ref(int),bit_offset(0),field_bits(32);
        field_name(A virt func ptr):type_def(21)=type_desc(ptr to)type_def(22)=
        sym_desc(array)index_type_ref(range of int from 0 to 1);
        elem_type_ref(vtbl elem type),
        bit_offset(32);
        meth_name(A_virt)::typedef(23)=sym_desc(method)returning(int);
        :arg_type(int),protection(public)normal(yes)virtual(yes)
        vtable_index(1);class_first_defining(A);;;~%first_base(A);",
        N_LSYM,NIL,NIL,NIL
```

```
.stabs "A:t20=s8Adat:1,0,32;$vf20:21=*22=ar1;0;1;17,32;
        A_virt::23=##1;:i;2A*-2147483647;20;;;~%20;",128,0,0,0
```

7.12 Inheritence

Stabs describing C++ derived classes include additional sections that describe the inheritence hierarchy of the class. A derived class stab also encodes the number of base classes. For each base class it tells if the base class is virtual or not, and if the inheritence is private or public. It also gives the offset into the object of the portion of the object corresponding to each base class.

This additional information is embeded in the class stab following the number of bytes in the struct. First the number of base classes appears bracketed by an exclamation point and a comma.

Then for each base type there repeats a series: a virtual character, a visibilty character, a number, a comma, another number, and a semi-colon.

The virtual character is '1' if the base class is virtual and '0' if not. The visibility character is '2' if the derivation is public, '1' if it is protected, and '0' if it is private. Debuggers should ignore virtual or visibility characters they do not recognize, and assume a reasonable default (such as public and non-virtual) (GDB 4.11 does not, but this should be fixed in the next GDB release).

The number following the virtual and visibility characters is the offset from the start of the object to the part of the object pertaining to the base class.

After the comma, the second number is a type_descriptor for the base type. Finally a semi-colon ends the series, which repeats for each base class.

The source below defines three base classes `A`, `B`, and `C` and the derived class `D`.

```
class A {
public:
        int Adat;
```

```
        virtual int A_virt (int arg) { return arg; };
};

class B {
public:
        int B_dat;
        virtual int B_virt (int arg) {return arg; };
};

class C {
public:
        int Cdat;
        virtual int C_virt (int arg) {return arg; };
};

class D : A, virtual B, public C {
public:
        int Ddat;
        virtual int A_virt (int arg ) { return arg+1; };
        virtual int B_virt (int arg)  { return arg+2; };
        virtual int C_virt (int arg)  { return arg+3; };
        virtual int D_virt (int arg)  { return arg; };
};
```

Class stabs similar to the ones described earlier are generated for each base class.

```
.stabs "A:T20=s8Adat:1,0,32;$vf20:21=*22=ar1;0;1;17,32;
        A_virt::23=##1;:i;2A*-2147483647;20;;;~%20;",128,0,0,0

.stabs "B:Tt25=s8Bdat:1,0,32;$vf25:21,32;B_virt::26=##1;
        :i;2A*-2147483647;25;;;~%25;",128,0,0,0

.stabs "C:Tt28=s8Cdat:1,0,32;$vf28:21,32;C_virt::29=##1;
        :i;2A*-2147483647;28;;;~%28;",128,0,0,0
```

In the stab describing derived class D below, the information about the derivation of this class is encoded as follows.

.stabs "derived_class_name:symbol_descriptors(struct tag&type)=
type_descriptor(struct)struct_bytes(32)!num_bases(3),
base_virtual(no)inheritence_public(no)base_offset(0),
base_class_type_ref(A);
base_virtual(yes)inheritence_public(no)base_offset(NIL),
base_class_type_ref(B);
base_virtual(no)inheritence_public(yes)base_offset(64),
base_class_type_ref(C); . . .

```
.stabs "D:Tt31=s32!3,000,20;100,25;0264,28;$vb25:24,128;Ddat:
        1,160,32;A_virt::32=##1;:i;2A*-2147483647;20;;B_virt:
        :32:i;2A*-2147483647;25;;C_virt::32:i;2A*-2147483647;
        28;;D_virt::32:i;2A*-2147483646;31;;;~%20;",128,0,0,0
```

7.13 Virtual Base Classes

A derived class object consists of a concatenation in memory of the data areas defined by each base class, starting with the leftmost and ending with the rightmost in the list of base classes. The exception to this rule is for virtual inheritence. In the example above, class D inherits virtually from base class B. This means that an instance of a D object will not contain its own B part but merely a pointer to a B part, known as a virtual base pointer.

In a derived class stab, the base offset part of the derivation information, described above, shows how the base class parts are ordered. The base offset for a virtual base class is always given as 0. Notice that the base offset for B is given as 0 even though B is not the first base class. The first base class A starts at offset 0.

The field information part of the stab for class D describes the field which is the pointer to the virtual base class B. The vbase pointer name is '$vb' followed by a type reference to the virtual base class. Since the type id for B in this example is 25, the vbase pointer name is '$vb25'.

```
.stabs "D:Tt31=s32!3,000,20;100,25;0264,28;$vb25:24,128;Ddat:1,
       160,32;A_virt::32=##1;:i;2A*-2147483647;20;;B_virt::32:i;
       2A*-2147483647;25;;C_virt::32:i;2A*-2147483647;28;;D_virt:
       :32:i;2A*-2147483646;31;;;~%20;",128,0,0,0
```

Following the name and a semicolon is a type reference describing the type of the virtual base class pointer, in this case 24. Type 24 was defined earlier as the type of the B class this pointer. The this pointer for a class is a pointer to the class type.

```
.stabs "this:P24=*25=xsB:",64,0,0,8
```

Finally the field offset part of the vbase pointer field description shows that the vbase pointer is the first field in the D object, before any data fields defined by the class. The layout of a D class object is a follows, Adat at 0, the vtable pointer for A at 32, Cdat at 64, the vtable pointer for C at 96, the virtual base pointer for B at 128, and Ddat at 160.

7.14 Static Members

The data area for a class is a concatenation of the space used by the data members of the class. If the class has virtual methods, a vtable pointer follows the class data. The field offset part of each field description in the class stab shows this ordering.

<< How is this reflected in stabs? See Cygnus bug #677 for some info. >>

Appendix A Table of Stab Types

The following are all the possible values for the stab type field, for a.out files, in numeric order. This does not apply to XCOFF, but it does apply to stabs in sections (see Appendix F [Stab Sections], page 61). Stabs in ECOFF use these values but add 0x8f300 to distinguish them from non-stab symbols.

The symbolic names are defined in the file '`include/aout/stabs.def`'.

A.1 Non-Stab Symbol Types

The following types are used by the linker and assembler, not by stab directives. Since this document does not attempt to describe aspects of object file format other than the debugging format, no details are given.

`0x0 N_UNDF` Undefined symbol

`0x2 N_ABS` File scope absolute symbol

`0x3 N_ABS | N_EXT` External absolute symbol

`0x4 N_TEXT` File scope text symbol

`0x5 N_TEXT | N_EXT` External text symbol

`0x6 N_DATA` File scope data symbol

`0x7 N_DATA | N_EXT` External data symbol

`0x8 N_BSS` File scope BSS symbol

`0x9 N_BSS | N_EXT` External BSS symbol

`0x0c N_FN_SEQ` Same as `N_FN`, for Sequent compilers

`0x0a N_INDR` Symbol is indirected to another symbol

`0x12 N_COMM` Common—visible after shared library dynamic link

`0x14 N_SETA`
`0x15 N_SETA | N_EXT`
Absolute set element

`0x16 N_SETT`
`0x17 N_SETT | N_EXT`
Text segment set element

`0x18 N_SETD`
`0x19 N_SETD | N_EXT`
Data segment set element

`0x1a N_SETB`
`0x1b N_SETB | N_EXT`
BSS segment set element

`0x1c N_SETV`
`0x1d N_SETV | N_EXT`
Pointer to set vector

`0x1e N_WARNING` Print a warning message during linking

`0x1f N_FN` File name of a '`.o`' file

A.2 Stab Symbol Types

The following symbol types indicate that this is a stab. This is the full list of stab numbers, including stab types that are used in languages other than C.

`0x20 N_GSYM` Global symbol; see Section 4.2 [Global Variables], page 15.

`0x22 N_FNAME` Function name (for BSD Fortran); see Section 2.5 [Procedures], page 9.

`0x24 N_FUN` Function name (see Section 2.5 [Procedures], page 9) or text segment variable (see Section 4.5 [Statics], page 17).

`0x26 N_STSYM` Data segment file-scope variable; see Section 4.5 [Statics], page 17.

`0x28 N_LCSYM` BSS segment file-scope variable; see Section 4.5 [Statics], page 17.

`0x2a N_MAIN` Name of main routine; see Section 2.1 [Main Program], page 7.

`0x2c N_ROSYM` Variable in `.rodata` section; see Section 4.5 [Statics], page 17.

`0x30 N_PC` Global symbol (for Pascal); see Section D.1 [N_PC], page 55.

`0x32 N_NSYMS` Number of symbols (according to Ultrix V4.0); see Section D.2 [N_NSYMS], page 55.

`0x34 N_NOMAP` No DST map; see Section D.3 [N_NOMAP], page 55.

`0x38 N_OBJ` Object file (Solaris2).

`0x3c N_OPT` Debugger options (Solaris2).

`0x40 N_RSYM` Register variable; see Section 4.3 [Register Variables], page 16.

`0x42 N_M2C` Modula-2 compilation unit; see Section D.4 [N_M2C], page 56.

`0x44 N_SLINE` Line number in text segment; see Section 2.4 [Line Numbers], page 8.

`0x46 N_DSLINE` Line number in data segment; see Section 2.4 [Line Numbers], page 8.

`0x48 N_BSLINE` Line number in bss segment; see Section 2.4 [Line Numbers], page 8.

`0x48 N_BROWS` Sun source code browser, path to '`.cb`' file; see Section D.5 [N_BROWS], page 56.

`0x4a N_DEFD` GNU Modula2 definition module dependency; see Section D.6 [N_DEFD], page 56.

`0x4c N_FLINE` Function start/body/end line numbers (Solaris2).

`0x50 N_EHDECL` GNU C++ exception variable; see Section D.7 [N_EHDECL], page 56.

`0x50 N_MOD2` Modula2 info "for imc" (according to Ultrix V4.0); see Section D.8 [N_MOD2], page 56.

`0x54 N_CATCH` GNU C++ `catch` clause; see Section D.9 [N_CATCH], page 57.

`0x60 N_SSYM`	Structure of union element; see Section D.10 [N_SSYM], page 57.
`0x62 N_ENDM`	Last stab for module (Solaris2).
`0x64 N_SO`	Path and name of source file; see Section 2.2 [Source Files], page 7.
`0x80 N_LSYM`	Stack variable (see Section 4.1 [Stack Variables], page 15) or type (see Section 5.9 [Typedefs], page 31).
`0x82 N_BINCL`	Beginning of an include file (Sun only); see Section 2.3 [Include Files], page 7.
`0x84 N_SOL`	Name of include file; see Section 2.3 [Include Files], page 7.
`0xa0 N_PSYM`	Parameter variable; see Section 4.7 [Parameters], page 18.
`0xa2 N_EINCL`	End of an include file; see Section 2.3 [Include Files], page 7.
`0xa4 N_ENTRY`	Alternate entry point; see Section 2.8 [Alternate Entry Points], page 11.
`0xc0 N_LBRAC`	Beginning of a lexical block; see Section 2.7 [Block Structure], page 11.
`0xc2 N_EXCL`	Place holder for a deleted include file; see Section 2.3 [Include Files], page 7.
`0xc4 N_SCOPE`	Modula2 scope information (Sun linker); see Section D.11 [N_SCOPE], page 57.
`0xe0 N_RBRAC`	End of a lexical block; see Section 2.7 [Block Structure], page 11.
`0xe2 N_BCOMM`	Begin named common block; see Section 4.4 [Common Blocks], page 16.
`0xe4 N_ECOMM`	End named common block; see Section 4.4 [Common Blocks], page 16.
`0xe8 N_ECOML`	Member of a common block; see Section 4.4 [Common Blocks], page 16.
`0xea N_WITH`	Pascal `with` statement: type,,0,0,offset (Solaris2).
`0xf0 N_NBTEXT`	Gould non-base registers; see Section D.12 [Gould], page 57.
`0xf2 N_NBDATA`	Gould non-base registers; see Section D.12 [Gould], page 57.
`0xf4 N_NBBSS`	Gould non-base registers; see Section D.12 [Gould], page 57.
`0xf6 N_NBSTS`	Gould non-base registers; see Section D.12 [Gould], page 57.
`0xf8 N_NBLCS`	Gould non-base registers; see Section D.12 [Gould], page 57.

Appendix B Table of Symbol Descriptors

The symbol descriptor is the character which follows the colon in many stabs, and which tells what kind of stab it is. See Section 1.3 [String Field], page 2, for more information about their use.

digit
`(`
`-` Variable on the stack; see Section 4.1 [Stack Variables], page 15.

`:` C++ nested symbol; see See Section 7.2 [Nested Symbols], page 37.

`a` Parameter passed by reference in register; see Section 4.7.3 [Reference Parameters], page 20.

`b` Based variable; see Section 4.6 [Based Variables], page 17.

`c` Constant; see Chapter 3 [Constants], page 13.

`C` Conformant array bound (Pascal, maybe other languages); Section 4.7.4 [Conformant Arrays], page 20. Name of a caught exception (GNU C++). These can be distinguished because the latter uses `N_CATCH` and the former uses another symbol type.

`d` Floating point register variable; see Section 4.3 [Register Variables], page 16.

`D` Parameter in floating point register; see Section 4.7.1 [Register Parameters], page 19.

`f` File scope function; see Section 2.5 [Procedures], page 9.

`F` Global function; see Section 2.5 [Procedures], page 9.

`G` Global variable; see Section 4.2 [Global Variables], page 15.

`i` See Section 4.7.1 [Register Parameters], page 19.

`I` Internal (nested) procedure; see Section 2.6 [Nested Procedures], page 10.

`J` Internal (nested) function; see Section 2.6 [Nested Procedures], page 10.

`L` Label name (documented by AIX, no further information known).

`m` Module; see Section 2.5 [Procedures], page 9.

`p` Argument list parameter; see Section 4.7 [Parameters], page 18.

`pP` See Section 4.7 [Parameters], page 18.

`pF` Fortran Function parameter; see Section 4.7 [Parameters], page 18.

`P` Unfortunately, three separate meanings have been independently invented for this symbol descriptor. At least the GNU and Sun uses can be distinguished by the symbol type. Global Procedure (AIX) (symbol type used unknown); see Section 2.5 [Procedures], page 9. Register parameter (GNU) (symbol type `N_PSYM`); see Section 4.7 [Parameters], page 18. Prototype of function referenced by this file (Sun `acc`) (symbol type `N_FUN`).

`Q` Static Procedure; see Section 2.5 [Procedures], page 9.

`R` Register parameter; see Section 4.7.1 [Register Parameters], page 19.

`r` Register variable; see Section 4.3 [Register Variables], page 16.

`S` File scope variable; see Section 4.5 [Statics], page 17.

`s` Local variable (OS9000).

`t` Type name; see Section 5.9 [Typedefs], page 31.

`T` Enumeration, structure, or union tag; see Section 5.9 [Typedefs], page 31.

`v` Parameter passed by reference; see Section 4.7.3 [Reference Parameters], page 20.

`V` Procedure scope static variable; see Section 4.5 [Statics], page 17.

`x` Conformant array; see Section 4.7.4 [Conformant Arrays], page 20.

`X` Function return variable; see Section 4.7 [Parameters], page 18.

Appendix C Table of Type Descriptors

The type descriptor is the character which follows the type number and an equals sign. It specifies what kind of type is being defined. See Section 1.3 [String Field], page 2, for more information about their use.

digit `(`	Type reference; see Section 1.3 [String Field], page 2.
`-`	Reference to builtin type; see Section 5.1.3 [Negative Type Numbers], page 23.
`#`	Method (C++); see Section 7.7 [Method Type Descriptor], page 40.
`*`	Pointer; see Section 5.2 [Miscellaneous Types], page 26.
`&`	Reference (C++).
`@`	Type Attributes (AIX); see Section 1.3 [String Field], page 2. Member (class and variable) type (GNU C++); see Section 7.8 [Member Type Descriptor], page 41.
`a`	Array; see Section 5.5 [Arrays], page 28.
`A`	Open array; see Section 5.5 [Arrays], page 28.
`b`	Pascal space type (AIX); see Section 5.2 [Miscellaneous Types], page 26. Builtin integer type (Sun); see Section 5.1.2 [Builtin Type Descriptors], page 22. Const and volatile qualfied type (OS9000).
`B`	Volatile-qualified type; see Section 5.2 [Miscellaneous Types], page 26.
`c`	Complex builtin type (AIX); see Section 5.1.2 [Builtin Type Descriptors], page 22. Const-qualified type (OS9000).
`C`	COBOL Picture type. See AIX documentation for details.
`d`	File type; see Section 5.2 [Miscellaneous Types], page 26.
`D`	N-dimensional dynamic array; see Section 5.5 [Arrays], page 28.
`e`	Enumeration type; see Section 5.7 [Enumerations], page 29.
`E`	N-dimensional subarray; see Section 5.5 [Arrays], page 28.
`f`	Function type; see Section 5.11 [Function Types], page 32.
`F`	Pascal function parameter; see Section 5.11 [Function Types], page 32
`g`	Builtin floating point type; see Section 5.1.2 [Builtin Type Descriptors], page 22.
`G`	COBOL Group. See AIX documentation for details.
`i`	Imported type (AIX); see Section 5.3 [Cross-References], page 27. Volatile-qualified type (OS9000).
`k`	Const-qualified type; see Section 5.2 [Miscellaneous Types], page 26.
`K`	COBOL File Descriptor. See AIX documentation for details.
`M`	Multiple instance type; see Section 5.2 [Miscellaneous Types], page 26.

`n` String type; see Section 5.6 [Strings], page 29.

`N` Stringptr; see Section 5.6 [Strings], page 29.

`o` Opaque type; see Section 5.9 [Typedefs], page 31.

`p` Procedure; see Section 5.11 [Function Types], page 32.

`P` Packed array; see Section 5.5 [Arrays], page 28.

`r` Range type; see Section 5.4 [Subranges], page 27.

`R` Builtin floating type; see Section 5.1.2 [Builtin Type Descriptors], page 22 (Sun). Pascal subroutine parameter; see Section 5.11 [Function Types], page 32 (AIX). Detecting this conflict is possible with careful parsing (hint: a Pascal subroutine parameter type will always contain a comma, and a builtin type descriptor never will).

`s` Structure type; see Section 5.8 [Structures], page 30.

`S` Set type; see Section 5.2 [Miscellaneous Types], page 26.

`u` Union; see Section 5.10 [Unions], page 31.

`v` Variant record. This is a Pascal and Modula-2 feature which is like a union within a struct in C. See AIX documentation for details.

`w` Wide character; see Section 5.1.2 [Builtin Type Descriptors], page 22.

`x` Cross-reference; see Section 5.3 [Cross-References], page 27.

`Y` Used by IBM's xlC C++ compiler (for structures, I think).

`z` gstring; see Section 5.6 [Strings], page 29.

Appendix D Expanded Reference by Stab Type

For a full list of stab types, and cross-references to where they are described, see Appendix A [Stab Types], page 47. This appendix just covers certain stabs which are not yet described in the main body of this document; eventually the information will all be in one place.

Format of an entry:

The first line is the symbol type (see 'include/aout/stab.def').

The second line describes the language constructs the symbol type represents.

The third line is the stab format with the significant stab fields named and the rest NIL.

Subsequent lines expand upon the meaning and possible values for each significant stab field.

Finally, any further information.

D.1 N_PC

N_PC .stabs

Global symbol (for Pascal).

```
"name" -> "symbol_name"  <<?>>
value  -> supposedly the line number (stab.def is skeptical)
```

'stabdump.c' says:

global pascal symbol: name,,0,subtype,line
<< subtype? >>

D.2 N_NSYMS

N_NSYMS .stabn

Number of symbols (according to Ultrix V4.0).

0, files,,funcs,lines (stab.def)

D.3 N_NOMAP

N_NOMAP .stabs

No DST map for symbol (according to Ultrix V4.0). I think this means a variable has been optimized out.

name, ,0,type,ignored (stab.def)

D.4 N_M2C

N_M2C `.stabs`

Modula-2 compilation unit.

```
"string" -> "unit_name,unit_time_stamp[,code_time_stamp]"
desc   -> unit_number
value  -> 0 (main unit)
          1 (any other unit)
```

See *Dbx and Dbxtool Interfaces*, 2nd edition, by Sun, 1988, for more information.

D.5 N_BROWS

N_BROWS `.stabs`

Sun source code browser, path to '`.cb`' file

<<?>> "path to associated '`.cb`' file"

Note: N_BROWS has the same value as N_BSLINE.

D.6 N_DEFD

N_DEFD `.stabn`

GNU Modula2 definition module dependency.

GNU Modula-2 definition module dependency. The value is the modification time of the definition file. The other field is non-zero if it is imported with the GNU M2 keyword `%INITIALIZE`. Perhaps `N_M2C` can be used if there are enough empty fields?

D.7 N_EHDECL

N_EHDECL `.stabs`

GNU C++ exception variable <<?>>.

"*string* is variable name"

Note: conflicts with `N_MOD2`.

D.8 N_MOD2

N_MOD2 `.stab?`

Modula2 info "for imc" (according to Ultrix V4.0)

Note: conflicts with `N_EHDECL` <<?>>

D.9 N_CATCH

N_CATCH `.stabn`

GNU C++ `catch` clause

GNU C++ `catch` clause. The value is its address. The desc field is nonzero if this entry is immediately followed by a `CAUGHT` stab saying what exception was caught. Multiple `CAUGHT` stabs means that multiple exceptions can be caught here. If desc is 0, it means all exceptions are caught here.

D.10 N_SSYM

N_SSYM `.stabn`

Structure or union element.

The value is the offset in the structure.

<<?looking at structs and unions in C I didn't see these>>

D.11 N_SCOPE

N_SCOPE `.stab?`

Modula2 scope information (Sun linker) <<?>>

D.12 Non-base registers on Gould systems

N_NBTEXT `.stab?`
N_NBDATA `.stab?`
N_NBBSS `.stab?`
N_NBSTS `.stab?`
N_NBLCS `.stab?`

These are used on Gould systems for non-base registers syms.

However, the following values are not the values used by Gould; they are the values which GNU has been documenting for these values for a long time, without actually checking what Gould uses. I include these values only because perhaps some someone actually did something with the GNU information (I hope not, why GNU knowingly assigned wrong values to these in the header file is a complete mystery to me).

```
240    0xf0    N_NBTEXT  ??
242    0xf2    N_NBDATA  ??
244    0xf4    N_NBBSS   ??
246    0xf6    N_NBSTS   ??
248    0xf8    N_NBLCS   ??
```

D.13 N_LENG

N_LENG `.stabn`

Second symbol entry containing a length-value for the preceding entry. The value is the length.

Appendix E Questions and Anomalies

- For GNU C stabs defining local and global variables (`N_LSYM` and `N_GSYM`), the desc field is supposed to contain the source line number on which the variable is defined. In reality the desc field is always 0. (This behavior is defined in '`dbxout.c`' and putting a line number in desc is controlled by '`#ifdef WINNING_GDB`', which defaults to false). GDB supposedly uses this information if you say '`list` *var*'. In reality, *var* can be a variable defined in the program and GDB says '`function` *var* `not defined`'.
- In GNU C stabs, there seems to be no way to differentiate tag types: structures, unions, and enums (symbol descriptor '`T`') and typedefs (symbol descriptor '`t`') defined at file scope from types defined locally to a procedure or other more local scope. They all use the `N_LSYM` stab type. Types defined at procedure scope are emited after the `N_RBRAC` of the preceding function and before the code of the procedure in which they are defined. This is exactly the same as types defined in the source file between the two procedure bodies. GDB overcompensates by placing all types in block #1, the block for symbols of file scope. This is true for default, '`-ansi`' and '`-traditional`' compiler options. (Bugs gcc/1063, gdb/1066.)
- What ends the procedure scope? Is it the proc block's `N_RBRAC` or the next `N_FUN`? (I believe its the first.)

Appendix F Using Stabs in Their Own Sections

Many object file formats allow tools to create object files with custom sections containing any arbitrary data. For any such object file format, stabs can be embedded in special sections. This is how stabs are used with ELF and SOM, and aside from ECOFF and XCOFF, is how stabs are used with COFF.

F.1 How to Embed Stabs in Sections

The assembler creates two custom sections, a section named `.stab` which contains an array of fixed length structures, one struct per stab, and a section named `.stabstr` containing all the variable length strings that are referenced by stabs in the `.stab` section. The byte order of the stabs binary data depends on the object file format. For ELF, it matches the byte order of the ELF file itself, as determined from the `EI_DATA` field in the `e_ident` member of the ELF header. For SOM, it is always big-endian (is this true??? FIXME). For COFF, it matches the byte order of the COFF headers. The meaning of the fields is the same as for a.out (see Section 6.1 [Symbol Table Format], page 33), except that the `n_strx` field is relative to the strings for the current compilation unit (which can be found using the synthetic N_UNDF stab described below), rather than the entire string table.

The first stab in the `.stab` section for each compilation unit is synthetic, generated entirely by the assembler, with no corresponding `.stab` directive as input to the assembler. This stab contains the following fields:

`n_strx` Offset in the `.stabstr` section to the source filename.

`n_type` `N_UNDF`.

`n_other` Unused field, always zero. This may eventually be used to hold overflows from the count in the `n_desc` field.

`n_desc` Count of upcoming symbols, i.e., the number of remaining stabs for this source file.

`n_value` Size of the string table fragment associated with this source file, in bytes.

The `.stabstr` section always starts with a null byte (so that string offsets of zero reference a null string), followed by random length strings, each of which is null byte terminated.

The ELF section header for the `.stab` section has its `sh_link` member set to the section number of the `.stabstr` section, and the `.stabstr` section has its ELF section header `sh_type` member set to `SHT_STRTAB` to mark it as a string table. SOM and COFF have no way of linking the sections together or marking them as string tables.

For COFF, the `.stab` and `.stabstr` sections may be simply concatenated by the linker. GDB then uses the `n_desc` fields to figure out the extent of the original sections. Similarly, the `n_value` fields of the header symbols are added together in order to get the actual position of the strings in a desired `.stabstr` section. Although this design obviates any need for the linker to relocate or otherwise manipulate `.stab` and `.stabstr` sections, it also requires some care to ensure that the offsets are calculated correctly. For instance, if the linker were to pad in between the `.stabstr` sections before concatenating, then the offsets to strings in the middle of the executable's `.stabstr` section would be wrong.

The GNU linker is able to optimize stabs information by merging duplicate strings and removing duplicate header file information (see Section 2.3 [Include Files], page 7). When some versions of the GNU linker optimize stabs in sections, they remove the leading `N_UNDF` symbol and arranges for all the `n_strx` fields to be relative to the start of the `.stabstr` section.

F.2 Having the Linker Relocate Stabs in ELF

This section describes some Sun hacks for Stabs in ELF; it does not apply to COFF or SOM.

To keep linking fast, you don't want the linker to have to relocate very many stabs. Making sure this is done for `N_SLINE`, `N_RBRAC`, and `N_LBRAC` stabs is the most important thing (see the descriptions of those stabs for more information). But Sun's stabs in ELF has taken this further, to make all addresses in the `n_value` field (functions and static variables) relative to the source file. For the `N_SO` symbol itself, Sun simply omits the address. To find the address of each section corresponding to a given source file, the compiler puts out symbols giving the address of each section for a given source file. Since these are ELF (not stab) symbols, the linker relocates them correctly without having to touch the stabs section. They are named `Bbss.bss` for the bss section, `Ddata.data` for the data section, and `Drodata.rodata` for the rodata section. For the text section, there is no such symbol (but there should be, see below). For an example of how these symbols work, See Section 6.2.3 [Stab Section Transformations], page 34. GCC does not provide these symbols; it instead relies on the stabs getting relocated. Thus addresses which would normally be relative to `Bbss.bss`, etc., are already relocated. The Sun linker provided with Solaris 2.2 and earlier relocates stabs using normal ELF relocation information, as it would do for any section. Sun has been threatening to kludge their linker to not do this (to speed up linking), even though the correct way to avoid having the linker do these relocations is to have the compiler no longer output relocatable values. Last I heard they had been talked out of the linker kludge. See Sun point patch 101052-01 and Sun bug 1142109. With the Sun compiler this affects '`S`' symbol descriptor stabs (see Section 4.5 [Statics], page 17) and functions (see Section 2.5 [Procedures], page 9). In the latter case, to adopt the clean solution (making the value of the stab relative to the start of the compilation unit), it would be necessary to invent a `Ttext.text` symbol, analogous to the `Bbss.bss`, etc., symbols. I recommend this rather than using a zero value and getting the address from the ELF symbols.

Finding the correct `Bbss.bss`, etc., symbol is difficult, because the linker simply concatenates the `.stab` sections from each '`.o`' file without including any information about which part of a `.stab` section comes from which '`.o`' file. The way GDB does this is to look for an ELF `STT_FILE` symbol which has the same name as the last component of the file name from the `N_SO` symbol in the stabs (for example, if the file name is '`../../gdb/main.c`', it looks for an ELF `STT_FILE` symbol named `main.c`). This loses if different files have the same name (they could be in different directories, a library could have been copied from one system to another, etc.). It would be much cleaner to have the `Bbss.bss` symbols in the stabs themselves. Having the linker relocate them there is no more work than having the linker relocate ELF symbols, and it solves the problem of having to associate the ELF and stab symbols. However, no one has yet designed or implemented such a scheme.

Symbol Types Index

Table of Contents