# The “stabs” debug format

**Julia Menapace**
**Cygnus Support**

Cygnus Support
Revision: 2.27
TEXinfo 2024-02-10.22

# 1 Overview of stabs

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

## 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 is translated by the assembler into a `.o` file, and then linked with other `.o` files and libraries to produce an executable file.

With the '`-g`' option, GCC puts additional debugging information in the `.s` file, 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 functions, their parameters and their scopes.

For some object file formats, the debugging information is encapsulated in assembler directives known collectively as 'stab' (symbol table) directives, 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 what combination of four possible data fields will follow. It is either `.stabs` (string), `.stabn` (number), or `.stabd` (dot).

The overall format of each class of stab is:

```
.stabs "string",type,0,desc,value
.stabn          type,0,desc,value
.stabd          type,0,desc
```

In general, in `.stabs` the *string* field contains name and type information. For `.stabd` the value field is implicit and has the value of the current file location. Otherwise the value field often contains a relocatable address, frame pointer offset, or register number, that maps to the source code element described by the stab.

The number in the type field gives some basic information about what type of stab this is (or whether it *is* a stab, as opposed to an ordinary symbol). Each possible type number defines a different stab type. The stab type further defines the exact interpretation of, and possible values for, any remaining "*string*", *desc*, or *value* fields present in the stab. Table A (see Section C.1 [Table A: Symbol types from stabs], page 47) lists in numeric order the possible type field values for stab directives. The reference section that follows Table A

describes the meaning of the fields for each stab type in detail. The examples that follow this overview introduce the stab types in terms of the source code elements they describe.

For `.stabs` the "*string*" field holds the meat of the debugging information. The generally unstructured 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 is of the "*string*" field is:

```
"name[:symbol_descriptor]
      [type_number[=type_descriptor ...]]"
```

*name* is the name of the symbol represented by the stab. *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.

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 Section C.3 [Table C: Symbol descriptors], page 48.

The '`c`' symbol descriptor is an exception in that it is not followed by type information. See Chapter 3 [Constants], page 9.

Type information is either a *type_number*, or a '*type_number*='. The *type_number* alone is a type reference, referring directly to a type that has already been defined.

The '*type_number*=' 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.

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*. If a number follows the '`=`' then the number is a *type_reference*. This is described more thoroughly in the section on types. See Section C.4 [Table D: Type Descriptors], page 49, for a list of *type_descriptor* values.

There is an AIX extension for type attributes. Following the '`=`' is any number of type attributes. Each one starts with '`@`' and ends with '`;`'. Debuggers, including AIX's dbx, skip any type attributes they do not recognize. The attributes are:

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

`s`*size*
: Size in bits of a variabe 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.

All 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 cretinously limit

the strings to about 80 characters, so compilers which must work with such DBX's need to split the `.stabs` directive into several `.stabs` directives. Each stab duplicates exactly all but the "*`string`*" field. The "*`string`*" field of every stab except the last is marked as continued with a double-backslash at the end. Removing the backslashes and concatenating the "*`string`*" fields of each stab produces the original, long string.

## 1.3 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.

## 1.4 The simple example at the assembly level

```
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:
37 LBB2:
38      sethi %hi(LC0),%o1
39      or %o1,%lo(LC0),%o0
40      call _printf,0
41      nop
42 .stabn 68,0,6,LM3
43 LM3:
44 LBE2:
45 .stabn 68,0,6,LM4
46 LM4:
47 L1:
48      ret
49      restore
50 .stabs "main:F1",36,0,0,_main
51 .stabn 192,0,0,LBB2
52 .stabn 224,0,0,LBE2
```

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

# 2 Encoding for the structure of the program

## 2.1 The path and name of the source file

**Directive:** `.stabs`

**Type:** `N_SO`

The first stabs in the .s file contain the name and path of the source file that was compiled to produce the .s file. This information is contained in two records of stab type N_SO (100).

```
        .stabs "path_name", N_SO, NIL, NIL, Code_address_of_program_start
        .stabs "file_name:", N_SO, NIL, NIL, Code_address_of_program_start
2       .stabs "/cygint/s1/users/jcm/play/",100,0,0,Ltext0
3       .stabs "hello.c",100,0,0,Ltext0
4           .text
5       Ltext0:
```

## 2.2 Line Numbers

**Directive:** `.stabn`

**Type:** `N_SLINE`

The start of source lines is represented by the `N_SLINE` (68) stab type.

`.stabn N_SLINE, NIL, `*`line`*`, address`

*line* is a source line number; *address* represents the code address for the start of that source line.

```
27 _main:
28 .stabn 68,0,4,LM1
29 LM1:
30      !#PROLOGUE# 0
```

## 2.3 Procedures

All of the following stabs use the '`N_FUN`' symbol type.

A function is represented by a '`F`' symbol descriptor for a global (extern) function, and '`f`' for a static (local) function. The next '`N_SLINE`' symbol can be used to find the line number of the start of the function. The value field is the address of the start of the function. 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.

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.

After the symbol descriptor and the type information, there is optionally a comma, followed by the name of the procedure, followed by a comma, followed by a name specifying the scope. The first name is local to the scope specified. I assume then that the name of the symbol (before the ':'), if specified, is some sort of global name. I assume the name specifying the scope is the name of a function specifying that scope. This feature is an AIX extension, and this information is based on the manual; I haven't actually tried it.

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.

```
48      ret
49      restore
```

The `.stabs` entry after this code fragment shows the *name* of the procedure (`main`); the type descriptor *desc* (`F`, for a global procedure); a reference to the predefined type `int` for the return type; and the starting *address* of the procedure.

Here is an exploded summary (with whitespace introduced for clarity), followed by line 50 of our sample assembly output, which has this form:

```
.stabs "name:
        desc   (global proc 'F')
        return_type_ref   (int)
       ",N_FUN, NIL, NIL,
       address
50 .stabs "main:F1",36,0,0,_main
```

## 2.4 Block Structure

**Directive:** `.stabn`

**Types:** `N_LBRAC`, `N_RBRAC`

The program's block structure is represented by the `N_LBRAC` (left brace) and the `N_RBRAC` (right brace) stab types. The following code range, which is the body of `main`, is labeled with '`LBB2:`' at the beginning and '`LBE2:`' at the end.

```
37 LBB2:
38      sethi %hi(LC0),%o1
39      or %o1,%lo(LC0),%o0
40      call _printf,0
41      nop
42 .stabn 68,0,6,LM3
43 LM3:
44 LBE2:
```

The `N_LBRAC` and `N_RBRAC` stabs that describe the block scope of the procedure are located after the `N_FUNC` stab that represents the procedure itself. The `N_LBRAC` uses the `LBB2` label as the code address in its value field, and the `N_RBRAC` uses `LBE2`.

```
50 .stabs "main:F1",36,0,0,_main
   .stabn N_LBRAC, NIL, NIL, left-brace-address
   .stabn N_RBRAC, NIL, NIL, right-brace-address
51 .stabn 192,0,0,LBB2
52 .stabn 224,0,0,LBE2
```

# 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*
: Enumeration constant. *type-information* is the type of the constant, as it would appear after a symbol descriptor (see Chapter 1 [Overview], page 1). *value* is the numeric value of the constant.

`i`*value*
: Integer constant. *value* is the numeric value.

`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 Chapter 1 [Overview], page 1). *elements* is the number of elements in the set (is this just the number of bits set in *pattern*? Or redundant with the type? 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.

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

This information is followed by ‘`;`’.

# 4 Simple types

## 4.1 Basic type definitions

**Directive:** `.stabs`

**Type:** `N_LSYM`

**Symbol Descriptor:**
`t`

The basic types for the language are described using the `N_LSYM` stab type. They are boilerplate and are emited by the compiler for each compilation unit. Basic type definitions are not always a complete description of the type and are sometimes circular. The debugger recognizes the type anyway, and knows how to read bits as that type.

Each language and compiler defines a slightly different set of basic types. In this example we are looking at the basic types for C emited by the GNU compiler targeting the Sun4. Here the basic types are mostly defined as range types.

## 4.2 Range types defined by min and max value

**Type Descriptor:**
`r`

When defining a range type, if the number after the first semicolon is smaller than the number after the second one, then the two numbers represent the smallest and the largest values in the range.

```
4  .text
5  Ltext0:

.stabs "name:
        descriptor  (type)
        type-def=
        type-desc
        type-ref;
        low-bound;
        high-bound;
       ",
       N_LSYM, NIL, NIL, NIL

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

Here the integer type (`1`) is defined as a range of the integer type (`1`). Likewise `char` is a range of `char`. This part of the definition is circular, but at least the high and low bound values of the range hold more information about the type.

Here short unsigned int is defined as type number 8 and described as a range of type `int`, with a minimum value of 0 and a maximum of 65535.

```
13 .stabs "short unsigned int:t8=r1;0;65535;",128,0,0,0
```

## 4.3 Range type defined by size in bytes

**Type Descriptor:**

`r`

In a range definition, if the first number after the semicolon is positive and the second is zero, then the type being defined is a floating point type, and the number after the first semicolon is the number of bytes needed to represent the type. Note that this does not provide a way to distinguish 8-byte real floating point types from 8-byte complex floating point types.

```
.stabs "name:
        desc
        type-def=
        type-desc
        type-ref;
        bit-count;
        0;
       ",
       N_LSYM, NIL, NIL, NIL

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
```

Cosmically enough, the `void` type is defined directly in terms of itself.

```
.stabs "name:
        symbol-desc
        type-def=
        type-ref
       ",N_LSYM,NIL,NIL,NIL

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

# 5 A Comprehensive Example in C

Now we'll examine a second program, `example2`, which builds on the first example to introduce the rest of the stab types, symbol descriptors, and type descriptors used in C. See Appendix A [Example2.c], page 41, for the complete `.c` source, and see Appendix B [Example2.s], page 43, for the `.s` assembly code. This description includes parts of those files.

## 5.1 Flow of control and nested scopes

**Directive:** `.stabn`

**Types:** `N_SLINE`, `N_LBRAC`, `N_RBRAC` (cont.)

Consider the body of `main`, from `example2.c`. It shows more about how `N_SLINE`, `N_RBRAC`, and `N_LBRAC` stabs are used.

```
20 {
21      static float s_flap;
22      int times;
23      for (times=0; times < s_g_repeat; times++){
24        int inner;
25        printf ("Hello world\n");
26      }
27 };
```

Here we have a single source line, the '`for`' line, that generates non-linear flow of control, and non-contiguous code. In this case, an `N_SLINE` stab with the same line number proceeds each block of non-contiguous code generated from the same source line.

The example also shows nested scopes. The `N_LBRAC` and `N_LBRAC` stabs that describe block structure are nested in the same order as the corresponding code blocks, those of the for loop inside those for the body of main.

This is the label for the `N_LBRAC` (left brace) stab marking the start of `main`.

```
57 LBB2:
```

In the first code range for C source line 23, the `for` loop initialize and test, `N_SLINE` (68) records the line number:

```
.stabn N_SLINE, NIL,
       line,
       address

58 .stabn 68,0,23,LM2
59 LM2:
60      st %g0,[%fp-20]
61 L2:
62      sethi %hi(_s_g_repeat),%o0
63      ld [%fp-20],%o1
64      ld [%o0+%lo(_s_g_repeat)],%o0
65      cmp %o1,%o0
```

```
66      bge L3
67      nop
```

label for the N_LBRAC (start block) marking the start of for loop

```
68 LBB3:
69 .stabn 68,0,25,LM3
70 LM3:
71      sethi %hi(LC0),%o1
72      or %o1,%lo(LC0),%o0
73      call _printf,0
74      nop
75 .stabn 68,0,26,LM4
76 LM4:
```

label for the N_RBRAC (end block) stab marking the end of the for loop

```
77 LBE3:
```

Now we come to the second code range for source line 23, the for loop increment and return. Once again, N_SLINE (68) records the source line number:

```
.stabn, N_SLINE, NIL,
        line,
        address
```

```
78 .stabn 68,0,23,LM5
79 LM5:
80 L4:
81      ld [%fp-20],%o0
82      add %o0,1,%o1
83      st %o1,[%fp-20]
84      b,a L2
85 L3:
86 .stabn 68,0,27,LM6
87 LM6:
```

label for the N_RBRAC (end block) stab marking the end of the for loop

```
88 LBE2:
89 .stabn 68,0,27,LM7
90 LM7:
91 L1:
92      ret
93      restore
94 .stabs "main:F1",36,0,0,_main
95 .stabs "argc:p1",160,0,0,68
96 .stabs "argv:p20=*21=*2",160,0,0,72
```

```
97 .stabs "s_flap:V12",40,0,0,_s_flap.0
98 .stabs "times:1",128,0,0,-20
```

Here is an illustration of stabs describing nested scopes. The scope nesting is reflected in the nested bracketing stabs (`N_LBRAC`, 192, appears here).

```
.stabn N_LBRAC,NIL,NIL,
       block-start-address

99  .stabn 192,0,0,LBB2      ## begin proc label
100 .stabs "inner:1",128,0,0,-24
101 .stabn 192,0,0,LBB3      ## begin for label
```

`N_RBRAC` (224), "right brace" ends a lexical block (scope).

```
.stabn N_RBRAC,NIL,NIL,
       block-end-address

102 .stabn 224,0,0,LBE3      ## end for label
103 .stabn 224,0,0,LBE2      ## end proc label
```

# 6 Variables

## 6.1 Locally scoped automatic variables

**Directive:** `.stabs`

**Type:** `N_LSYM`

**Symbol Descriptor:**

none

In addition to describing types, the `N_LSYM` stab type also describes locally scoped automatic variables. Refer again to the body of `main` in `example2.c`. It allocates two automatic variables: '`times`' is scoped to the body of `main`, and '`inner`' is scoped to the body of the `for` loop. '`s_flap`' is locally scoped but not automatic, and will be discussed later.

```
20 {
21      static float s_flap;
22      int times;
23      for (times=0; times < s_g_repeat; times++){
24        int inner;
25        printf ("Hello world\n");
26      }
27 };
```

The `N_LSYM` stab for an automatic variable is located just before the `N_LBRAC` stab describing the open brace of the block to which it is scoped.

`N_LSYM` (128): automatic variable, scoped locally to `main`

```
.stabs "name:
        type-ref",
       N_LSYM, NIL, NIL,
       frame-pointer-offset

98  .stabs "times:1",128,0,0,-20
99  .stabn 192,0,0,LBB2      ## begin `main' N_LBRAC
```

`N_LSYM` (128): automatic variable, scoped locally to the `for` loop

```
.stabs "name:
        type-ref",
        N_LSYM, NIL, NIL,
       frame-pointer-offset

100 .stabs "inner:1",128,0,0,-24
101 .stabn 192,0,0,LBB3      ## begin `for' loop N_LBRAC
```

Since the character in the string field following the colon is not a letter, there is no symbol descriptor. This means that the stab describes a local variable, and that the number after the colon is a type reference. In this case it a a reference to the basic type `int`. Notice also that the frame pointer offset is negative number for automatic variables.

## 6.2 Global Variables

**Directive:** `.stabs`

**Type:** `N_GSYM`

**Symbol Descriptor:**
`G`

Global variables are represented by the `N_GSYM` stab type. The symbol descriptor, following the colon in the string field, is '`G`'. Following the '`G`' is a type reference or type definition. In this example it is a type reference to the basic C type, `char`. The first source line in `example2.c`,

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

yields the following stab. The stab immediately precedes the code that allocates storage for the variable it describes.

`N_GSYM` (32): global symbol

```
.stabs "name:
        descriptor
        type-ref",
        N_GSYM, NIL, NIL, NIL

21 .stabs "g_foo:G2",32,0,0,0
22      .global _g_foo
23      .data
24 _g_foo:
25      .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.

## 6.3 Register variables

Register variables have their own stab type, `N_RSYM`, and their own symbol descriptor, `r`. The stab's value field contains the number of the register where the variable data will be stored.

The value is the register number.

AIX defines a separate symbol descriptor '`d`' for floating point registers. This seems incredibly stupid–why not just just give floating point registers different register numbers.

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

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

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

## 6.4 Initialized static variables

**Directive:** `.stabs`

**Type:** `N_STSYM`

**Symbol Descriptors:**
`S` (file scope), `V` (procedure scope)

Initialized static variables are represented by the `N_STSYM` stab type. The symbol descriptor part of the string field shows if the variable is file scope static ('`S`') or procedure scope static ('`V`'). The source line

```
3  static int s_g_repeat = 2;
```

yields the following code. The stab is located immediately preceding the storage for the variable it represents. Since the variable in this example is file scope static the symbol descriptor is '`S`'.

`N_STSYM` (38): initialized static variable (data seg w/internal linkage)

```
.stabs "name:
        descriptor
        type-ref",
       N_STSYM,NIL,NIL,
       address

26 .stabs "s_g_repeat:S1",38,0,0,_s_g_repeat
27      .align 4
28 _s_g_repeat:
29      .word 2
```

## 6.5 Un-initialized static variables

**Directive:** `.stabs`

**Type:** `N_LCSYM`

**Symbol Descriptors:**
`S` (file scope), `V` (procedure scope)

Un-initialized static variables are represented by the `N_LCSYM` stab type. The symbol descriptor part of the string shows if the variable is file scope static ('`S`') or procedure scope static ('`V`'). In this example it is procedure scope static. The source line allocating `s_flap` immediately follows the open brace for the procedure `main`.

```
20 {
21      static float s_flap;
```

The code that reserves storage for the variable `s_flap` precedes the body of body of `main`.

```
39      .reserve _s_flap.0,4,"bss",4
```

But since `s_flap` is scoped locally to `main`, its stab is located with the other stabs representing symbols local to `main`. The stab for `s_flap` is located just before the `N_LBRAC` for `main`.

`N_LCSYM` (40): uninitialized static var (BSS seg w/internal linkage)

```
.stabs "name:
```

```
        descriptor
        type-ref",
        N_LCSYM, NIL, NIL,
        address

97 .stabs "s_flap:V12",40,0,0,_s_flap.0
98 .stabs "times:1",128,0,0,-20
99 .stabn 192,0,0,LBB2                  # N_LBRAC for main.
```

## 6.6 Parameters

The symbol descriptor 'p' is used to refer to parameters which are in the arglist. Symbols have symbol type 'N_PSYM'. The value of the symbol is the offset relative to the argument list.

If the parameter is passed in a register, then the traditional way to do this is to provide two symbols for each argument:

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

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

Because this is kind of ugly, some compilers use symbol descriptor 'P' or 'R' to indicate an argument which is in a register. The symbol 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. Symbol type 'C_RPSYM' is used with 'R' and 'N_RSYM' is used with 'P'.

AIX, according to the documentation, uses 'D' for a parameter passed in a floating point register. This strikes me as incredibly bogus—why doesn't it 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.

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

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'/'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'/'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 below).

There is another case similar to an argument in a register, which is an argument which 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. Some compilers use the pair of symbols approach described above ("arg:p" followed by "arg:"); this includes gcc1 (not

gcc2) on the sparc when passing a small structure and gcc2 when the argument type is float and it is passed as a double and converted to float by the prologue (in the latter case the type of the "arg:p" symbol is double and the type of the "arg:" symbol is float). GCC, at least on the 960, 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).

If the parameter is passed by reference (e.g. Pascal VAR parameters), then type 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 other 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.

Conformant arrays refer to 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.

The following are also said to go with 'N_PSYM':

```
"name" -> "param_name:#type"
                        -> pP (<<??>>)
                        -> pF (<<??>>)
                        -> X  (function result variable)
                        -> b  (based variable)

value -> offset from the argument pointer (positive).
```

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.

# 7 Aggregate Types

Now let's look at some variable definitions involving complex types. This involves understanding better how types are described. In the examples so far types have been described as references to previously defined types or defined in terms of subranges of or pointers to previously defined types. The section that follows will talk about the various other type descriptors that may follow the = sign in a type definition.

## 7.1 Array types

**Directive:** `.stabs`

**Types:** `N_GSYM`, `N_LSYM`

**Symbol Descriptor:**
`T`

**Type Descriptor:**
`a`

As an example of an array type consider the global variable below.

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

Since the array is a global variable, it is described by the N_GSYM stab type. The symbol descriptor G, following the colon in stab's string field, also says the array is a global variable. Following the G is a definition for type (19) as shown by the equals sign after the type number.

After the equals sign is a type descriptor, a, which says that the type being defined is an array. Following the type descriptor for an array is the type of the index, a semicolon, and the type of the array elements.

The type of the index is often a range type, expressed as the letter r and some parameters. It defines the size of the array. In 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.

The array definition above generates the assembly language that follows.

<32> N_GSYM - global variable
.stabs "name:sym_desc(global)type_def(19)=type_desc(array)
index_type_ref(range of int from 0 to 2);element_type_ref(char)";
N_GSYM, NIL, NIL, NIL

```
32 .stabs "char_vec:G19=ar1;0;2;2",32,0,0,0
33      .global _char_vec
34      .align 4
35 _char_vec:
36      .byte 97
37      .byte 98
38      .byte 99
```

## 7.2 Enumerations

**Directive:** `.stabs`

**Type:** `N_LSYM`

**Symbol Descriptor:**
`T`

**Type Descriptor:**
`e`

The source line below declares an enumeration type. It is defined at file scope between the bodies of main and s_proc in example2.c. Because the N_LSYM 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, the N_LSYM does not describe a block local symbol, but a file local one. The source line:

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

generates the following stab, located just after the N_RBRAC (close brace stab) for main. The type definition is in an N_LSYM stab because type definitions are file scope not global scope.

```
<128> N_LSYM - local symbol
.stab "name:sym_dec(type)type_def(22)=sym_desc(enum)
      enum_name:value(0),enum_name:value(3),enum_name:value(4),;",
      N_LSYM, NIL, NIL, NIL
104 .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 type 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 a ;.

## 7.3 Structure Tags

**Directive:** `.stabs`

**Type:** `N_LSYM`

**Symbol Descriptor:**
`T`

**Type Descriptor:**
`s`

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. A seperate stab is 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 .common.

```
6  struct s_tag {
```

```
7    int   s_int;
8    float s_float;
9    char  s_char_vec[8];
10   struct s_tag* s_next;
11 } g_an_s;
12
13 typedef struct s_tag s_typedef;
```

The structure tag is an N_LSYM stab type because, like the enum, the symbol is file scope. Like the enum, the symbol descriptor is T, for enumeration, struct or tag type. The symbol descriptor s following the 16= of the type definition narrows the symbol type to struct.

Following the struct symbol descriptor is the number of bytes the struct 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, and number of bits in the element.

```
    <128> N_LSYM - type definition
    .stabs "name:sym_desc(struct tag) Type_def(16)=type_desc(struct type)
         struct_bytes
         elem_name:type_ref(int),bit_offset,field_bits;
         elem_name:type_ref(float),bit_offset,field_bits;
         elem_name:type_def(17)=type_desc(array)
index_type(range of int from 0 to 7);
         element_type(char),bit_offset,field_bits;;",
         N_LSYM,NIL,NIL,NIL

30 .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, two of the 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 an type definition for an element which is a pointer to type 16.

## 7.4 Typedefs

**Directive:** `.stabs`

**Type:** `N_LSYM`

**Symbol Descriptor:**
`t`

Here is the stab for the typedef equating the structure tag with a type.

```
    <128> N_LSYM - type definition
    .stabs "name:sym_desc(type name)type_ref(struct_tag)",N_LSYM,NIL,NIL,NIL
31 .stabs "s_typedef:t16",128,0,0,0
```

And here is the code generated for the structure variable.

<32> N_GSYM - global symbol
.stabs "name:sym_desc(global)type_ref(struct_tag)",N_GSYM,NIL,NIL,NIL

```
136 .stabs "g_an_s:G16",32,0,0,0
137     .common _g_an_s,20,"bss"
```

Notice that the structure tag has the same type number as the typedef for the structure tag. It is impossible to distinguish between a variable of the struct type and one of its typedef by looking at the debugging information.

## 7.5 Unions

**Directive:** `.stabs`

**Type:** `N_LSYM`

**Symbol Descriptor:**
`T`

**Type Descriptor:**
`u`

Next let's look at unions. In example2 this union type is declared locally to a procedure and an instance of the union is defined.

```
36   union u_tag {
37     int  u_int;
38     float u_float;
39     char* u_char;
40   } an_u;
```

This code generates a stab for the union tag and a stab for the union variable. Both use the N_LSYM stab type. Since the 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> N_LSYM - type
.stabs "name:sym_desc(union tag)type_def(22)=type_desc(union)
byte_size(4)
elem_name:type_ref(int),bit_offset(0),bit_size(32);
elem_name:type_ref(float),bit_offset(0),bit_size(32);
elem_name:type_ref(ptr to char),bit_offset(0),bit_size(32);;"
N_LSYM, NIL, NIL, NIL

```
105 .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, struct or type 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, and number of bytes for the element;.

The stab for the union variable follows. Notice that the frame pointer offset for local variables is negative.

```
<128> N_LSYM - local variable (with no symbol descriptor)
.stabs "name:type_ref(u_tag)", N_LSYM, NIL, NIL, frame_ptr_offset
130 .stabs "an_u:23",128,0,0,-20
```

## 7.6 Function types

type descriptor f

The last type descriptor in C which remains to be described is used for function types. Consider the following source line defining a global function pointer.

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

It generates the following code. Since the variable is not initialized, the code is located in the common area at the end of the file.

```
<32> N_GSYM - global variable
.stabs "name:sym_desc(global)type_def(24)=ptr_to(25)=
                        type_def(func)type_ref(int)
134 .stabs "g_pf:G24=*25=f1",32,0,0,0
135     .common _g_pf,4,"bss"
```

Since the variable is global, the stab type is N_GSYM and the symbol descriptor is G. The variable defines a new type, 24, which is a pointer to another new type, 25, which is defined as a function returning int.

# 8 Symbol information in symbol tables

This section examines more closely the format of symbol table entries and how stab assembler directives map to them. It also describes what transformations the assembler and linker make on data from stabs.

Each time the assembler encounters a stab in its input file it puts each field of the stab into corresponding fields 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 */
};
```

For .stabs directives, the n_strx field holds the character offset from the start of the string table to the string table entry containing the "string" field. For other classes of stabs (.stabn and .stabd) this field is null.

Symbol table entries with n_type fields containing a value greater or equal to 0x20 originated as stabs generated by the compiler (with one random exception). Those with n_type values less than 0x20 were placed in the symbol table of the executable by the assembler or the linker.

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, first the assemble and then the link.

To do this use nm with the -ap options. This 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.

There are a few important things to notice about symbol tables. Where the value field of a stab contains a frame pointer offset, or a register number, that value is unchanged by the rest of the build.

Where the value field 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. This source line defines a static variable at file scope:

```
3  static int s_g_repeat
```

The following stab describes the symbol.

```
26 .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.

```
21 00000084 - 00 0000 STSYM s_g_repeat:S1
```

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

```
22 0000e00c - 00 0000 STSYM s_g_repeat:S1
```

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:

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

generates the stab:

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

The variable is represented by the following two symbol table entries in the object file. 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 (see Table B). The value field following the file's line number is empty for the stab entry. For the linker symbol it contains the rellocatable address corresponding to the variable.

```
19 00000000 - 00 0000  GSYM g_foo:G2
20 00000080 D _g_foo
```

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

```
21 00000000 - 00 0000  GSYM g_foo:G2
...
215 0000e008 D _g_foo
```

# 9 GNU C++ stabs

### 9.0.1 Symbol descriptors added for C++ descriptions:

P - register parameter.

### 9.0.2 type descriptors added for C++ descriptions

`#` method type (two ## if minimal debug)

`xs` cross-reference

## 9.1 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
```

## 9.2 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 a 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 then the symbol descriptor '##', indicating a method type. This is followed by a type reference showing the return type of the method and a semi-colon.

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

## 9.3 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
```

## 9.4 Method defintion

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 not described by this document - Is there already such a doc? >>

```
.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 >>

## 9.5 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.

Protections for class member data are signified by two characters embeded in the stab defining the class type. These characters are located after the name: part of the string. /0 means private, /1 means protected, and /2 means public. If these characters are omited this means that the member is public. The following C++ source:

```
class all_data {
private:
        int   priv_dat;
protected:
        char  prot_dat;
public:
        float pub_dat;
};
```

generates the following stab to describe the class type all_data.

```
.stabs "class_name:sym_desc(type)type_def(19)=type_desc(struct)struct_bytes
      data_name:/protection(private)type_ref(int),bit_offset,num_bits;
      data_name:/protection(protected)type_ref(char),bit_offset,num_bits;
      data_name:(/num omited, private)type_ref(float),bit_offset,num_bits;;"
      N_LSYM,NIL,NIL,NIL
.stabs "all_data:t19=s12
       priv_dat:/01,0,32;prot_dat:/12,32,8;pub_dat:12,64,32;;",128,0,0,0
```

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
```

## 9.6 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
```

## 9.7 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*-214748364
```

## 9.8 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: two digits, a number, a comma, another number, and a semi-colon.

The first of the two digits is 1 if the base class is virtual and 0 if not. The second digit is 2 if the derivation is public and 0 if not.

The number following the first two digits 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
```

## 9.9 Virtual Base Classes

A derived class object consists of a concatination 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 it's 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 ase pointer for B at 128, and Ddat at 160.

## 9.10 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 Example2.c - source code for extended example

```
1  char g_foo = 'c';
2  register int g_bar asm ("%g5");
3  static int s_g_repeat = 2;
4  int (*g_pf)();
5
6  struct s_tag {
7    int   s_int;
8    float s_float;
9    char  s_char_vec[8];
10   struct s_tag* s_next;
11 } g_an_s;
12
13 typedef struct s_tag s_typedef;
14
15 char char_vec[3] = {'a','b','c'};
16
17 main (argc, argv)
18      int argc;
19      char* argv[];
20 {
21      static float s_flap;
22      int times;
23      for (times=0; times < s_g_repeat; times++){
24        int inner;
25        printf ("Hello world\n");
26      }
27 };
28
29 enum e_places {first,second=3,last};
30
31 static s_proc (s_arg, s_ptr_arg, char_vec)
32   s_typedef s_arg;
33   s_typedef* s_ptr_arg;
34   char* char_vec;
35 {
36   union u_tag {
37     int  u_int;
38     float u_float;
39     char* u_char;
40   } an_u;
41 }
42
43
```

# Appendix B Example2.s - assembly code for extended example

```
1  gcc2_compiled.:
2  .stabs "/cygint/s1/users/jcm/play/",100,0,0,Ltext0
3  .stabs "example2.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 .stabs "g_foo:G2",32,0,0,0
22      .global _g_foo
23      .data
24 _g_foo:
25      .byte 99
26 .stabs "s_g_repeat:S1",38,0,0,_s_g_repeat
27      .align 4
28 _s_g_repeat:
29      .word 2
30 .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
31 .stabs "s_typedef:t16",128,0,0,0
32 .stabs "char_vec:G19=ar1;0;2;2",32,0,0,0
33      .global _char_vec
34      .align 4
35 _char_vec:
36      .byte 97
37      .byte 98
38      .byte 99
39      .reserve _s_flap.0,4,"bss",4
40      .text
41      .align 4
42 LC0:
43      .ascii "Hello world\12\0"
44      .align 4
```

```
45      .global _main
46      .proc 1
47 _main:
48 .stabn 68,0,20,LM1
49 LM1:
50      !#PROLOGUE# 0
51      save %sp,-144,%sp
52      !#PROLOGUE# 1
53      st %i0,[%fp+68]
54      st %i1,[%fp+72]
55      call ___main,0
56      nop
57 LBB2:
58 .stabn 68,0,23,LM2
59 LM2:
60      st %g0,[%fp-20]
61 L2:
62      sethi %hi(_s_g_repeat),%o0
63      ld [%fp-20],%o1
64      ld [%o0+%lo(_s_g_repeat)],%o0
65      cmp %o1,%o0
66      bge L3
67      nop
68 LBB3:
69 .stabn 68,0,25,LM3
70 LM3:
71      sethi %hi(LC0),%o1
72      or %o1,%lo(LC0),%o0
73      call _printf,0
74      nop
75 .stabn 68,0,26,LM4
76 LM4:
77 LBE3:
78 .stabn 68,0,23,LM5
79 LM5:
80 L4:
81      ld [%fp-20],%o0
82      add %o0,1,%o1
83      st %o1,[%fp-20]
84      b,a L2
85 L3:
86 .stabn 68,0,27,LM6
87 LM6:
88 LBE2:
89 .stabn 68,0,27,LM7
90 LM7:
91 L1:
```

```
92      ret
93      restore
94 .stabs "main:F1",36,0,0,_main
95 .stabs "argc:p1",160,0,0,68
96 .stabs "argv:p20=*21=*2",160,0,0,72
97 .stabs "s_flap:V12",40,0,0,_s_flap.0
98 .stabs "times:1",128,0,0,-20
99 .stabn 192,0,0,LBB2
100 .stabs "inner:1",128,0,0,-24
101 .stabn 192,0,0,LBB3
102 .stabn 224,0,0,LBE3
103 .stabn 224,0,0,LBE2
104 .stabs "e_places:T22=efirst:0,second:3,last:4,;",128,0,0,0
105 .stabs "u_tag:T23=u4u_int:1,0,32;u_float:12,0,32;u_char:21,0,32;;",
128,0,0,0
106     .align 4
107     .proc 1
108 _s_proc:
109 .stabn 68,0,35,LM8
110 LM8:
111     !#PROLOGUE# 0
112     save %sp,-120,%sp
113     !#PROLOGUE# 1
114     mov %i0,%o0
115     st %i1,[%fp+72]
116     st %i2,[%fp+76]
117 LBB4:
118 .stabn 68,0,41,LM9
119 LM9:
120 LBE4:
121 .stabn 68,0,41,LM10
122 LM10:
123 L5:
124     ret
125     restore
126 .stabs "s_proc:f1",36,0,0,_s_proc
127 .stabs "s_arg:p16",160,0,0,0
128 .stabs "s_ptr_arg:p18",160,0,0,72
129 .stabs "char_vec:p21",160,0,0,76
130 .stabs "an_u:23",128,0,0,-20
131 .stabn 192,0,0,LBB4
132 .stabn 224,0,0,LBE4
133 .stabs "g_bar:r1",64,0,0,5
134 .stabs "g_pf:G24=*25=f1",32,0,0,0
135     .common _g_pf,4,"bss"
136 .stabs "g_an_s:G16",32,0,0,0
137     .common _g_an_s,20,"bss"
```

# Appendix C Quick reference

## C.1 Table A: Symbol types from stabs

Table A lists stab types sorted by type number. Stab type numbers are 32 and greater. This is the full list of stab numbers, including stab types that are used in languages other than C.

The #define names for these stab types are defined in: devo/include/aout/stab.def

```
type   type     #define   used to describe
dec    hex      name      source program feature
------------------------------------------------
32     0x20     N_GYSM    global symbol
34     0X22     N_FNAME   function name (for BSD Fortran)
36     0x24     N_FUN     function name or text segment variable for C
38     0x26     N_STSYM   static symbol (data segment w/internal linkage)
40     0x28     N_LCSYM   .lcomm symbol(BSS-seg variable w/internal linkage)
42     0x2a     N_MAIN    Name of main routine (not used in C)
48     0x30     N_PC      global symbol (for Pascal)
50     0x32     N_NSYMS   number of symbols (according to Ultrix V4.0)
52     0x34     N_NOMAP   no DST map for sym (according to Ultrix V4.0)
64     0x40     N_RSYM    register variable
66     0x42     N_M2C     Modula-2 compilation unit
68     0x44     N_SLINE   line number in text segment
70     0x46     N_DSLINE  line number in data segment

72     0x48     N_BSLINE  line number in bss segment
72     0x48     N_BROWS   Sun source code browser, path to .cb file

74     0x4a     N_DEFD    GNU Modula2 definition module dependency

80     0x50     N_EHDECL  GNU C++ exception variable
80     0x50     N_MOD2    Modula2 info "for imc" (according to Ultrix V4.0)

84     0x54     N_CATCH   GNU C++ "catch" clause
96     0x60     N_SSYM    structure of union element
100    0x64     N_SO      path and name of source file
128    0x80     N_LSYM    automatic var in the stack
                          (also used for type desc.)
130    0x82     N_BINCL   beginning of an include file (Sun only)
132    0x84     N_SOL     Name of sub-source (#include) file.
160    0xa0     N_PSYM    parameter variable
162    0xa2     N_EINCL   end of an include file
164    0xa4     N_ENTRY   alternate entry point
192    0xc0     N_LBRAC   beginning of a lexical block
194    0xc2     N_EXCL    place holder for a deleted include file
196    0xc4     N_SCOPE   modula2 scope information (Sun linker)
224    0xe0     N_RBRAC   end of a lexical block
226    0xe2     N_BCOMM   begin named common block
228    0xe4     N_ECOMM   end named common block
232    0xe8     N_ECOML   end common (local name)

       << used on Gould systems for non-base registers syms >>
240    0xf0     N_NBTEXT  ??
242    0xf2     N_NBDATA  ??
244    0xf4     N_NBBSS   ??
246    0xf6     N_NBSTS   ??
```

```
248    0xf8     N_NBLCS   ??
```

## C.2 Table B: Symbol types from assembler and linker

Table B shows the types of symbol table entries that hold assembler and linker symbols.

The #define names for these n_types values are defined in /include/aout/aout64.h

```
dec     hex     #define
n_type  n_type  name      used to describe
-----------------------------------------
1       0x0     N_UNDF    undefined symbol
2       0x2     N_ABS     absolute symbol -- defined at a particular address
3       0x3             extern " (vs. file scope)
4       0x4     N_TEXT    text symbol -- defined at offset in text segment
5       0x5             extern " (vs. file scope)
6       0x6     N_DATA    data symbol -- defined at offset in data segment
7       0x7             extern " (vs. file scope)
8       0x8     N_BSS     BSS symbol -- defined at offset in zero'd segment
9                       extern " (vs. file scope)

12      0x0C    N_FN_SEQ  func name for Sequent compilers (stab exception)

49      0x12    N_COMM    common sym -- visable after shared lib dynamic link
31      0x1f    N_FN      file name of a .o file
```

## C.3 Table C: Symbol descriptors

`(empty)` Local variable, See Section 6.1 [Automatic variables], page 17.

`a` Parameter passed by reference in register, See Section 6.6 [Parameters], page 20.

`c` Constant, See Chapter 3 [Constants], page 9.

`C` Conformant array bound, See Section 6.6 [Parameters], page 20.

`d` Floating point register variable, See Section 6.3 [Register variables], page 18.

`D` Parameter in floating point register, See Section 6.6 [Parameters], page 20.

`f` Static function, See Section 2.3 [Procedures], page 5.

`F` Global function, See Section 2.3 [Procedures], page 5.

`G` Global variable, See Section 6.2 [Global Variables], page 18.

`i` See Section 6.6 [Parameters], page 20.

`I` Internal (nested) procedure, See Section 2.3 [Procedures], page 5.

`J` Internal (nested) function, See Section 2.3 [Procedures], page 5.

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

`m` Module, See Section 2.3 [Procedures], page 5.

`p` Argument list parameter See Section 6.6 [Parameters], page 20.

`pP` See Section 6.6 [Parameters], page 20.

`pF` See Section 6.6 [Parameters], page 20.

`P` Global Procedure (AIX), See Section 2.3 [Procedures], page 5. Register parameter (GNU), See Section 6.6 [Parameters], page 20.

`Q` Static Procedure, See Section 2.3 [Procedures], page 5.

`R` Register parameter See Section 6.6 [Parameters], page 20.

`r` Register variable, See Section 6.3 [Register variables], page 18.

`S` Static file scope variable See Section 6.4 [Initialized statics], page 18, See Section 6.5 [Un-initialized statics], page 19.

`t` Type name, See Section 7.4 [Typedefs], page 25.

`T` enumeration, struct or union tag, See Section 7.5 [Unions], page 26.

`v` Call by reference, See Section 6.6 [Parameters], page 20.

`V` Static procedure scope variable See Section 6.4 [Initialized statics], page 18, See Section 6.5 [Un-initialized statics], page 19.

`x` Conformant array, See Section 6.6 [Parameters], page 20.

`X` Function return variable, See Section 6.6 [Parameters], page 20.

## C.4 Table D: Type Descriptors

`(digits)` Type reference, See Chapter 1 [Overview], page 1.

`*` Pointer type.

`@` Type Attributes (AIX), See Chapter 1 [Overview], page 1. Some C++ thing (GNU).

`a` Array type.

`e` Enumeration type.

`f` Function type.

`r` Range type.

`s` Structure type.

`u` Union specifications.

# Appendix D Expanded reference by stab type.

Format of an entry:

The first line is the symbol type expressed in decimal, hexadecimal, and as a #define (see devo/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. # stands in for the type descriptor.

Finally, any further information.

## D.1 32 - 0x20 - N_GYSM

Global variable.

.stabs "name", N_GSYM, NIL, NIL, NIL

```
"name" -> "symbol_name:#type"
                        # -> G
```

Only the "name" field is significant. The location of the variable is obtained from the corresponding external symbol.

## D.2 34 - 0x22 - N_FNAME

Function name (for BSD Fortran)

.stabs "name", N_FNAME, NIL, NIL, NIL

```
"name" -> "function_name"
```

Only the "name" field is significant. The location of the symbol is obtained from the corresponding extern symbol.

## D.3 36 - 0x24 - N_FUN

Function name (see Section 2.3 [Procedures], page 5) or text segment variable (see Chapter 6 [Variables], page 17).

*For functions:*

```
"name" -> "proc_name:#return_type"
                     #  -> F (global function)
                           f (local function)
desc  -> line num for proc start.  (GCC doesn't set and DBX doesn't miss it.)
value -> Code address of proc start.
```

*For text segment variables:*

```
<<How to create one?>>
```

## D.4 38 - 0x26 - N_STSYM

Initialized static symbol (data segment w/internal linkage).

.stabs "name", N_STSYM, NIL, NIL, value

```
"name" -> "symbol_name#type"
                       # -> S (scope global to compilation unit)
                         -> V (scope local to a procedure)
value  -> Data Address
```

## D.5 40 - 0x28 - N_LCSYM

Unitialized static (.lcomm) symbol(BSS segment w/internal linkage).

.stabs "name", N_LCLSYM, NIL, NIL, value

```
"name" -> "symbol_name#type"
                       # -> S (scope global to compilation unit)
                         -> V (scope local to procedure)
value  -> BSS Address
```

## D.6 42 - 0x2a - N_MAIN

Name of main routine (not used in C)

.stabs "name", N_MAIN, NIL, NIL, NIL

```
"name" -> "name_of_main_routine"
```

## D.7 48 - 0x30 - N_PC

Global symbol (for Pascal)

.stabs "name", N_PC, NIL, NIL, value

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

stabdump.c says:

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

## D.8 50 - 0x32 - N_NSYMS

Number of symbols (according to Ultrix V4.0)

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

## D.9 52 - 0x34 - N_NOMAP

no DST map for sym (according to Ultrix V4.0)

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

## D.10 64 - 0x40 - N_RSYM

register variable

.stabs "name:type",N_RSYM,0,RegSize,RegNumber (Sun doc)

## D.11 66 - 0x42 - N_M2C

Modula-2 compilation unit

.stabs "name", N_M2C, 0, desc, value

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

## D.12 68 - 0x44 - N_SLINE

Line number in text segment

.stabn N_SLINE, 0, desc, value

```
desc  -> line_number
value -> code_address (relocatable addr where the corresponding code starts)
```

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

## D.13 70 - 0x46 - N_DSLINE

Line number in data segment

.stabn N_DSLINE, 0, desc, value

```
desc  -> line_number
value -> data_address (relocatable addr where the corresponding code
starts)
```

See comment for N_SLINE above.

## D.14 72 - 0x48 - N_BSLINE

Line number in bss segment

.stabn N_BSLINE, 0, desc, value

```
desc  -> line_number
value -> bss_address (relocatable addr where the corresponding code
starts)
```

See comment for N_SLINE above.

## D.15 72 - 0x48 - N_BROWS

Sun source code browser, path to .cb file

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

Note: type field value overlaps with N_BSLINE

## D.16 74 - 0x4a - N_DEFD

GNU Modula2 definition module dependency

GNU Modula-2 definition module dependency. Value is the modification time of the definition file. Other 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.17 80 - 0x50 - N_EHDECL

GNU C++ exception variable <<?>>

"name is variable name"

Note: conflicts with N_MOD2.

## D.18 80 - 0x50 - N_MOD2

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

Note: conflicts with N_EHDECL <<?>>

## D.19 84 - 0x54 - N_CATCH

GNU C++ "catch" clause

GNU C++ 'catch' clause. Value is its address. Desc 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.20 96 - 0x60 - N_SSYM

Structure or union element

Value is offset in the structure.

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

## D.21 100 - 0x64 - N_SO

Path and name of source file containing main routine

.stabs "name", N_SO, NIL, NIL, value

```
"name" -> /source/directory/
       -> source_file

value  -> the starting text address of the compilation.
```

These are found two in a row. The name field of the first N_SO contains the directory that the source file is relative to. The name field of the second N_SO contains the name of the source file itself.

Only some compilers (e.g. gcc2, Sun cc) include the directory; this symbol can be distinguished by the fact that it ends in a slash. According to a comment in GDB's partial-stab.h, other compilers (especially unnamed C++ compilers) put out useless N_SO's for nonexistent source files (after the N_SO for the real source file).

## D.22 128 - 0x80 - N_LSYM

Automatic var in the stack (also used for type descriptors.)

.stabs "name" N_LSYM, NIL, NIL, value

*For stack based local variables:*

```
"name" -> name of the variable
value  -> offset from frame pointer (negative)
```

*For type descriptors:*

```
"name"   -> "name_of_the_type:#type"
                              # -> t

type     -> type_ref (or) type_def

type_ref -> type_number
type_def -> type_number=type_desc etc.
```

Type may be either a type reference or a type definition. A type reference is a number that refers to a previously defined type. A type definition is the number that will refer to this type, followed by an equals sign, a type descriptor and the additional data that defines the type. See the Table D for type descriptors and the section on types for what data follows each type descriptor.

## D.23 130 - 0x82 - N_BINCL

Beginning of an include file (Sun only)

Beginning of an include file. Only Sun uses this. In an object file, only the name is significant. The Sun linker puts data into some of the other fields.

## D.24 132 - 0x84 - N_SOL

Name of a sub-source file (#include file). Value is starting address of the compilation. <<?>>

## D.25 160 - 0xa0 - N_PSYM

Parameter variable. See Section 6.6 [Parameters], page 20.

## D.26 162 - 0xa2 - N_EINCL

End of an include file. This and N_BINCL act as brackets around the file's output. In an ojbect file, there is no significant data in this entry. The Sun linker puts data into some of the fields. <<?>>

## D.27 164 - 0xa4 - N_ENTRY

Alternate entry point. Value is its address. <<?>>

## D.28 192 - 0xc0 - N_LBRAC

Beginning of a lexical block (left brace). The variable defined inside the block precede the N_LBRAC symbol. Or can they follow as well as long as a new N_FUNC was not encountered. <<?>>

.stabn N_LBRAC, NIL, NIL, value

`value -> code address of block start.`

## D.29 194 - 0xc2 - N_EXCL

Place holder for a deleted include file. Replaces a N_BINCL and everything up to the corresponding N_EINCL. The Sun linker generates these when it finds multiple indentical copies of the symbols from an included file. This appears only in output from the Sun linker. <<?>>

## D.30 196 - 0xc4 - N_SCOPE

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

## D.31 224 - 0xe0 - N_RBRAC

End of a lexical block (right brace)

.stabn N_RBRAC, NIL, NIL, value

`value -> code address of the end of the block.`

## D.32 226 - 0xe2 - N_BCOMM

Begin named common block.

Only the name is significant. <<?>>

## D.33 228 - 0xe4 - N_ECOMM

End named common block.

Only the name is significant and it should match the N_BCOMM <<?>>

## D.34 232 - 0xe8 - N_ECOML

End common (local name)

value is address. <<?>>

## D.35 Non-base registers on Gould systems

<< used on Gould systems for non-base registers syms, values assigned at random, need real info from Gould. >> <<?>>

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

## D.36 - 0xfe - N_LENG

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 behavour 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.)
- The comment in xcoff.h says DBX_STATIC_CONST_VAR_CODE is used for static const variables. DBX_STATIC_CONST_VAR_CODE is set to N_FUN by default, in dbxout.c. If included, xcoff.h redefines it to N_STSYM. But testing the default behaviour, my Sun4 native example shows N_STSYM not N_FUN is used to describe file static initialized variables. (the code tests for TREE_READONLY(decl) && !TREE_THIS_VOLATILE(decl) and if true uses DBX_STATIC_CONST_VAR_CODE).
- Global variable stabs don't have location information. This comes from the external symbol for the same variable. The external symbol has a leading underbar on the _name of the variable and the stab does not. How do we know these two symbol table entries are talking about the same symbol when their names are different?
- Can gcc be configured to output stabs the way the Sun compiler does, so that their native debugging tools work? <NO?> It doesn't by default. GDB reads either format of stab. (gcc or SunC). How about dbx?

# Appendix F Differences between GNU stabs in a.out and GNU stabs in xcoff

(The AIX/RS6000 native object file format is xcoff with stabs). This appendix only covers those differences which are not covered in the main body of this document.

- Instead of .stabs, xcoff uses .stabx.
- The data fields of an xcoff .stabx are in a different order than an a.out .stabs. The order is: string, value, type. The desc and null fields present in a.out stabs are missing in xcoff stabs. For N_GSYM the value field is the name of the symbol.
- BSD a.out stab types correspond to AIX xcoff storage classes. In general the mapping is N_STABTYPE becomes C_STABTYPE. Some stab types in a.out are not supported in xcoff. See Table E. for full mappings.

  exception: initialised static N_STSYM and un-initialized static N_LCSYM both map to the C_STSYM storage class. But the destinction is preserved because in xcoff N_STSYM and N_LCSYM must be emited in a named static block. Begin the block with .bs s[RW] data_section_name for N_STSYM or .bs s bss_section_name for N_LCSYM. End the block with .es
- xcoff stabs describing tags and typedefs use the N_DECL (0x8c)instead of N_LSYM stab type.
- xcoff uses N_RPSYM (0x8e) instead of the N_RSYM stab type for register variables. If the register variable is also a value parameter, then use R instead of P for the symbol descriptor.

  6. xcoff uses negative numbers as type references to the basic types. There are no boilerplate type definitions emited for these basic types. << make table of basic types and type numbers for C >>
- xcoff .stabx sometimes don't have the name part of the string field.
- xcoff uses a .file stab type to represent the source file name. There is no stab for the path to the source file.
- xcoff uses a .line stab type to represent source lines. The format is: .line line_number.
- xcoff emits line numbers relative to the start of the current function. The start of a function is marked by .bf. If a function includes lines from a seperate file, then those line numbers are absolute line numbers in the <<sub-?>> file being compiled.
- The start of current include file is marked with: .bi "filename" and the end marked with .ei "filename"
- If the xcoff stab is a N_FUN (C_FUN) then follow the string field with ,. instead of just ,

(I think that's it for .s file differences. They could stand to be better presented. This is just a list of what I have noticed so far. There are a *lot* of differences in the information in the symbol tables of the executable and object files.)

Table E: mapping a.out stab types to xcoff storage classes

```
stab type       storage class
-------------------------------
N_GSYM          C_GSYM
N_FNAME         unknown
```

```
N_FUN           C_FUN
N_STSYM         C_STSYM
N_LCSYM         C_STSYM
N_MAIN          unkown
N_PC            unknown
N_RSYM          C_RSYM
N_RPSYM (0x8e)  C_RPSYM
N_M2C           unknown
N_SLINE         unknown
N_DSLINE        unknown
N_BSLINE        unknown
N_BROWSE        unchanged
N_CATCH         unknown
N_SSYM          unknown
N_SO            unknown
N_LSYM          C_LSYM
N_DECL  (0x8c)  C_DECL
N_BINCL         unknown
N_SOL           unknown
N_PSYM          C_PSYM
N_EINCL         unknown
N_ENTRY         C_ENTRY
N_LBRAC         unknown
N_EXCL          unknown
N_SCOPE         unknown
N_RBRAC         unknown
N_BCOMM         C_BCOMM
N_ECOMM         C_ECOMM
N_ECOML         C_ECOML

N_LENG          unknown
```

# Appendix G Differences between GNU stabs and Sun native stabs.

- GNU C stabs define *all* types, file or procedure scope, as N_LSYM. Sun doc talks about using N_GSYM too.
- Stabs describing block scopes, N_LBRAC and N_RBRAC are supposed to contain the nesting level of the block in the desc field, re Sun doc. GNU stabs always have 0 in that field. dbx seems not to care.
- Sun C stabs use type number pairs in the format (a,b) where a is a number starting with 1 and incremented for each sub-source file in the compilation. b is a number starting with 1 and incremented for each new type defined in the compilation. GNU C stabs use the type number alone, with no source file number.

# Table of Contents