Stabs Interface

Sun™ Studio 9

(Valid for Sun Studio 10, Sun Studio 11,
and Sun Studio 12)

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

Please
Recycle

Contents

CHAPTER 1

Introduction

The command line debugger, `dbx`, depends on two kinds of information generated by compilers and the linker to aid the user in debugging programs. The first type of information is exactly the same information that the linker uses to combine object files and that the loader uses to execute a program. The second type is generated by the compilers specifically to support debugging. This information is stored in a format known as *stabs*, which stands for symbol table entries. This document describes how these stabs are created, stored, modified, and interpreted.

The debugger supports the ELF format for object files (generated under the Solaris™ operating environment; for a description, see Chapter 3). In an ELF file, the stabs are stored in separate sections from the symbol table generated by the compiler and linker.

Although `dbx` is the most common interpreter of stabs, other programs (for example, the Performance Analyzer) use the information in stabs in one form or another.

This document tells you:

- How a compiler describes the program in stabs
- How the stabs relate to the linker symbol table entries stored in the object file or executable file
- What happens to stabs when the linker processes an object file
- How `dbx` interprets the stabs

A glossary is included to define the various terms used.

The examples of stab output from the compilers are not intended to define the stab specification, but are provided for clarification of the specification.

CHAPTER 2

Overview

`dbx` evolved from `Pdx`, a Pascal source debugger developed by Mark A. Linton as a Master's project at the University of California at Berkeley. Linton extended the linker symbol table entry to contain descriptions of variables, functions, and types, by encoding this information in the symbol name field. To avoid confusion with the existing linker stabs, additional stabs description codes were defined.

Since type and variable information is encoded as strings, stabs are easy to extend to support additional languages or other features. Additional stab types can be (and have been) created to meet changing needs without affecting the processing of existing stab types. Stabs have evolved significantly to:

- Support C++, Fortran 77, Fortran 95, and C
- Reduce the size of executable files
- Support additional operating system features such as dynamically loaded shared libraries
- Improve debugger performance

Most compilers translate the source into assembler instructions and pass this to the assembler. The assembler generates linker stab entries for files and non-local symbols. The compilers generate debugging stabs when the `-g` option is specified by including either `.stabs` or `.stabn` directives in the source passed to the assembler. These have the following formats:

`.stabs` *"string"*`,` *type*`,` *other*`,` *desc*`,` *value*

and

`.stabn` *type*`,` *other*`,` *desc*`,` *value*

where:

string contains the name and description of a symbol and, in general, consists of a name followed by a colon, a symbol descriptor (one or more characters), and descriptor specific information.

type specifies the type of the stab entry.

other is used in some stab types for miscellaneous inforamtion.

desc is used in some stab types to further describe the symbol.

value contains an offset or other value.

In an ELF file, debugging stabs are stored in the `.stab` section with the text of the strings in `.stabstr`.

There are two additional assembler directives:

`.xstabs "`*section*`", "`*string*`",` *type*`,` *other*`,` *desc*`,` *value*

`.xcstabs "`*section*`", "`*string*`",` *type*`,` *other*`,` *desc*`,` *value*

where:

section is the name of the section in which to place the stab.

string, *type*, *other*, *desc*, and *value* have the same meanings described for the `.stabs` and `.stabn` directives..

The `.xstabs` directive and .`xcstabs` directive can be used to direct the stab into a different section in an ELF file. The `.xstabs` directive is usually used to create index stabs, described in Chapter 3 and Appendix D, but is also used in other special situations. The `.xcstabs` directive is used to create COMDAT index stabs, which are described in Chapter 3.

The string in the stabs directives can be of any length, up to the string size limit imposed by the assembler. (Currently the assembler does not impose a limit, so the compilers can generate a stab string of any length.) To ease generation of stabs directives, the string may be continued from one stabs directive to the next by terminating the string with a backslash (\). The continuation stabs directive must have the same *type*, *desc*, and *value*. For example,

```
.stabs "boolean:t(0,2)=efalse:0,\\", 0x20, 0, 0, 0
.stabs "true:1,", 0x20, 0, 0, 0
```

is equivalent to

```
.stabs "boolean:t(0,2)=efalse:0,true:1,", 0x20, 0, 0, 0
```

There may be any number of continuation lines.

Each stabs directive contains a stab type that describes what is contained in the string part of the stab. When the stab describes a symbol, the stab type specifies whether the symbol is a local or global symbol, a function description, static variable, and so forth. In the preceding example, the stab type of `0x20` indicates that this stab describes a global symbol.

The stabs that describe symbols use the string to contain three pieces of information:

- Symbol name
- Symbol descriptor
- Type description

The name of the symbol starts the string and is followed by a colon. The symbol descriptor immediately follows the colon and describes what the symbol represents. In the example above, the symbol `boolean` is described to be a type definition by the `t` symbol type. Descriptions of local variables omit the symbol descriptor.

The actual description of the type is contained following the symbol descriptor. This may be a reference to a previously defined type or it may be a new definition of a type as indicated by a type number pair followed by an equal sign, as is done in the description of `boolean`.

Symbol names that also represent ELF level synbols normally retain their ELF spellings. One exception is Fortran 77, which omits the trailing underscore on the function names in the stabs.

CHAPTER 3

ELF Object File Format

ELF is the Executable and Linking Format used in System V and is the native executable file format for the Solaris operating environment. It is extensively described in the System V ABI[1]. This chapter briefly describes the format, focusing on the aspects that affect stabs.

Each symbol table entry has the following format (defined in `stab.h`):

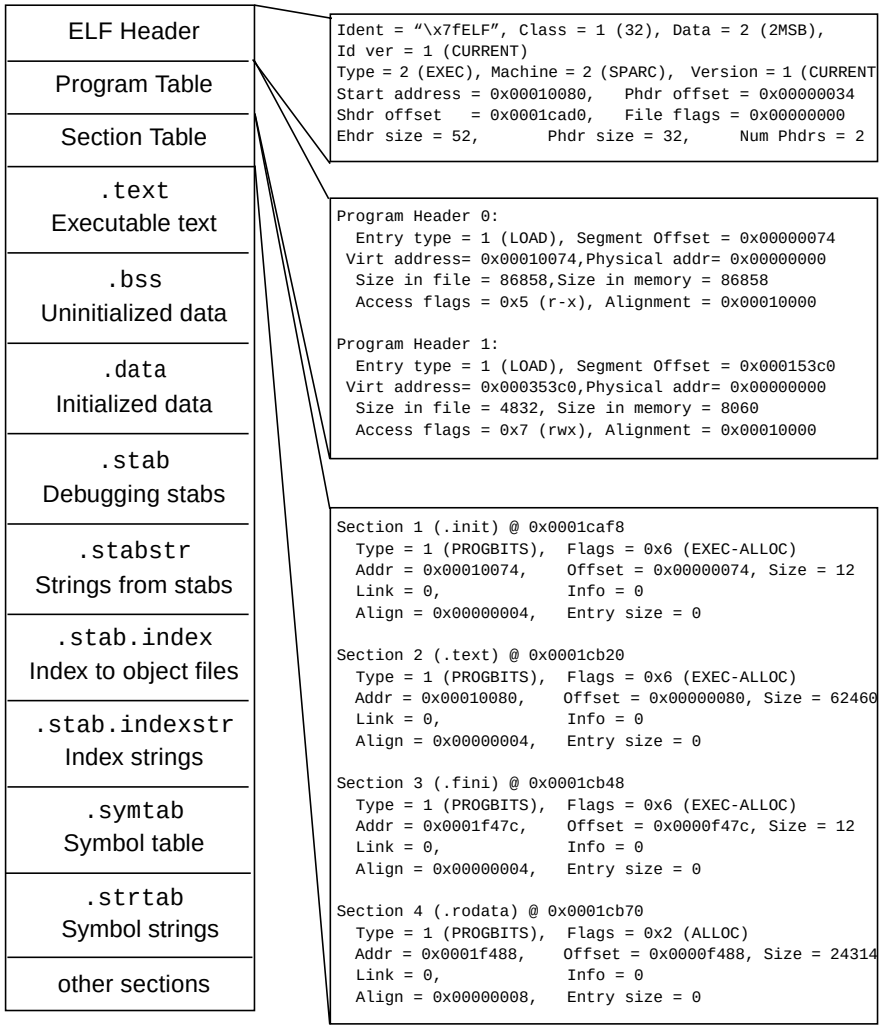
```
struct stab {
    unsigned       n_strx;      /* file String table index */
    unsigned char  n_type;      /* Stab type */
    char           n_other;     /* used by N_SLINE stab */
    short          n_desc;      /* Desc value */
    unsigned       n_value;     /* Offset or value */
};
```

The `n_strx` field is the offset of the string in the symbol string table. All strings are terminated by a null byte. Previous stabs versions defined struct `nlist` in `nlist.h`. This has been changed to avoid conflict with the system header file and struct of that name. The fields of struct stab were also redefined so that their sizes would not change when compiled for 64-bit programs.

An ELF file, whether an object file, an executable file, or a library file, is a highly structured file that consists of a header, a program table, a section table, and a number of named sections. The section table is an index to the sections, describing each section's name, type, storage address, length, and offset in the ELF file. Several sections have predefined names and contents. For example, the `.text` section contains the executable instructions of the program and the `.data` section contains initialized data. An ELF file may contain additional sections that have contents specified by the vendor, such as the `.stab` section. The following figure shows the layout of an ELF executable file.

1. 1. *AT&T: System V Application Binary Interface and SPARC Processor Supplement*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1990

ELF Executable Layout



The symbol table is in the section named .symtab and the strings are stored in the .strtab section. Each entry in the symbol table consists of:

- A name
- A value (which is usually an address)
- A size
- Type and binding flags
- A reference to the section to which it is related

The symbol table entries for an object file start with a symbol with the STT_FILE type with the name of the source file used to create the object file (the file name only, not the entire path to the source file). This is followed by local symbols which are defined in the file, functions are identified with type STT_FUNC and variables with type STT_OBJECT.

The symbol table entries for an executable file or shared library are concatenated from the object files that were linked. The first STT_FILE type has the name of the executable or shared library. The next STT_FILE should be the source file of the first object file linked. The next STT_FILE should be the source file of the second object file linked, and so on.

Following the last object file are the global symbols. Both function and variable symbols are collected at the end in random order. The following figure shows parts of the symbol table for a small "hello world" program. The STT_FILE entry for hello.c is followed by two local data objects, sa and sb, while the STT_FUNC entry for main appears much later in the symbol table. Nothing in the symbol table indicates which source file contains any global function.

ELF Symbol Table

```
Symbol table -- 479 entries  value    size         info      shndx

                     crti.s  00000000  00000000  LOCAL    FILE  ABS
                     crt1.s  00000000  00000000  LOCAL    FILE  ABS
                 values-Xt.c 00000000  00000000  LOCAL    FILE  ABS
                     hello.c 00000000  00000000  LOCAL    FILE  ABS
                          sa  000353e0  00000004  LOCAL  OBJECT    6
                          sb  000353e4  00000004  LOCAL  OBJECT    6
                    atexit.c  00000000  00000000  LOCAL    FILE  ABS
                     exitfns  000366a0  00000094  LOCAL  OBJECT    8
                   numexitfns 0003542c  00000004  LOCAL  OBJECT    6
                    printf.c  00000000  00000000  LOCAL    FILE  ABS
                    doprnt.c  00000000  00000000  LOCAL    FILE  ABS
                     _zeroes  00035465  00000015  LOCAL  OBJECT    6
                                    .
                                    .
                                    .
                        .mul  00018a84  0000022c GLOBAL    FUNC    2
                       __iob  000363c8  00000140 GLOBAL  OBJECT    6
                        open  0001d7f0  00000000   WEAK    FUNC    2
                        main  00010120  00000044 GLOBAL    FUNC    2
                    getwidth  000196b4  00000064 GLOBAL    FUNC    2
                        read  0001cd54  00000000   WEAK    FUNC    2
          double_to_decimal   00018510  00000000   WEAK    FUNC    2
                      malloc  0001e400  00000298 GLOBAL    FUNC    2
                        _iob  000363c8  00000000   WEAK  OBJECT    6
                     __ctype  00035554  00000209 GLOBAL  OBJECT    6
                      wctomb  00019494  00000128 GLOBAL    FUNC    2
                      getpid  0001f440  00000000   WEAK    FUNC    2
                           a  000353c0  00000001 GLOBAL  OBJECT    6
                           b  000353c2  00000002 GLOBAL  OBJECT    6
```

The organization of the debugging stabs in an ELF file is not defined in the ABI, but is specific to the Solaris operating environment.

Debugging stabs may be stored in several sections in the ELF file. In an object file (not an executable file or library) the stabs are stored in either the .stab section (with strings in .stabstr) or in the .stab.excl (with strings in .stab.exclstr). If auto-load stab processing is to be used (that is, the object file will be retained rather than deleted after linking) then stabs are placed in the .stab.excl section. The Solaris linker will not copy the .stab.excl or .stab.exclstr sections to the executable. If the object file is not to be retained, stabs are stored in the .stab (and .stabstr) section and will be copied to the executable file.

The .stab.index section (and .stab.indexstr containing the strings) contain a reduced set of stabs that are used to support auto-load stab processing. These stabs specify the names of global functions and data that are contained in the object file (since this information is not available in the symbol table) and where to find the object file so that the debugger can read the stabs. For a more complete description, see Appendix D.

Stabs and the strings from each object file are concatenated to form these sections in the executable file. The stabs from each object file are preceded with an N_UNDF stab. The n_value field in this stab contains the total length of the strings for the object file. The n_strx field of the N_UNDF stab contains the offset to the file name string (usually a source file). The linker does not relocate or modify the offset to the strings. The order of N_UNDF stabs and the Elf STT_FILE symbols should match, preserved by the linker. dbx expects this order to be preserved.

When the COMDAT feature of the linker is used by a compiler, it is necessary to place some index stabs in COMDAT sections: .stab.index%function and the corresponding string section .stab.indexstr%function where function is the linker name of the function as given by the compiler. Unlike all other stab sections in an object file, these do not begin with an N_UNDF stab. Instead, each stab is encoded with n_other == 1 to identify it as a COMDAT stab. If multiple sections with the same name exist, the linker chooses one for the executable. The function's code and its index stab are always chosen together, so dbx always knows which function was chosen.

The non-COMDAT .stab.index and .stab.indexstr sections must always appear in the object file before any COMDAT stab sections, so that if the COMDAT sections are chosen by the linker, they always follow the N_UNDF stab in the .stab.index section. The order of the .stab.index%* COMDAT stab sections in the object file dictates the order of the corresponding .stab.indexstr%* COMDAT stab string sections.

The chosen COMDAT index stabs are concatenated into the executable along with the regular index stabs; the chosen COMDAT index stab strings are concatenated with the regular index stab strings. The N_UNDF stab count and string table size do not include the COMDAT stabs. The debugger must explicitly look for former COMDAT stabs when reading the index stabs of the executable. It must adjust for incorrect n_strx fields, which neither the compiler nor linker can adjust.

The following figure shows the various stab sections in an executable file:

ELF Stabs Sections

```
Excluded Stab table -- 38 entries

   0:  .stabs "busy.c",N_UNDF,0x0,0x25,0x309
   1:  .stabs "/usr/src/play/",N_SO,0x0,0x0,0x0
   2:  .stabs "busy.c",N_SO,0x0,0x3,0x0
   3:  .stabs "",N_OBJ,0x0,0x0,0x0
   4:  .stabs "",N_OBJ,0x0,0x0,0x0
   5:  .stabs "V=8.0;DBG_GEN=4.0.143;Xa;g;R=Forte Developer 7 C 5.4
2002/03/09;G=$XAY9kkBSUQm8ymc.",N_OPT,0x0,0x0,0x3c990512
   6:  .stabs "busy.c",N_SOL,0x0,0x0,0x0
   7:  .stabs "char:t(0,1)=bsc1;0;8",N_LSYM,0x0,0x0,0x0
   8:  .stabs "short:t(0,2)=bs2;0;16",N_LSYM,0x0,0x0,0x0
   9:  .stabs "int:t(0,3)=bs4;0;32",N_LSYM,0x0,0x0,0x0
                           .
                           .
23:  .stabs "float:t(0,17)=R1;4",N_LSYM,0x0,0x0,0x0
  24:  .stabs "double:t(0,18)=R2;8",N_LSYM,0x0,0x0,0x0
  25:  .stabs "long double:t(0,19)=R6;16",N_LSYM,0x0,0x0,0x0
  26:  .stabs "void:t(0,20)=bs0;0;0",N_LSYM,0x0,0x0,0x0
  27:  .stabs "main:F(0,3)",N_FUN,0x0,0x0,0x0
  28:  .stabs "main",N_MAIN,0x0,0x0,0x0
  29:  .stabn N_LBRAC,0x0,0x1,0x4
  30:  .stabs "i:(0,3)",N_LSYM,0x0,0x4,0xfffffff8
  31:  .stabs
"$XBY9kkBSUQm8ymc.main.__func__:V(0,21)=ar(0,4);0;4;(0,22)=k(0,1)",N_
ROSYM,0x0,0x5,0x0
  32:  .stabs "lazy:P(0,3)",N_FUN,0x0,0x0,0x0
  33:  .stabn N_SLINE,0x0,0x3,0x4
  34:  .stabn N_SLINE,0x0,0x4,0x10
  35:  .stabn N_SLINE,0x0,0x5,0x24
  36:  .stabn N_RBRAC,0x0,0x1,0x24
  37:  .stabn N_ENDM,0x0,0x0,0x0
```

.

ELF Stabs Sections (continued)

```
Index Stab table -- 29 entries

   0:  .stabs "busy.c",N_UNDF,0x0,0x9,0x123
   1:  .stabs "/usr/src/play/",N_SO,0x0,0x0,0x0
   2:  .stabs "busy.c",N_SO,0x0,0x3,0x0
   3:  .stabs "/usr/src/play",N_OBJ,0x0,0x0,0x0
   4:  .stabs "busy.o",N_OBJ,0x0,0x0,0x0
  5:  .stabs "V=8.0;DBG_GEN=4.0.143;Xa;g;R=Forte Developer 7 C 5.4 2002/
03/09;G=$XAY9kkBSUQm8ymc.",N_OPT,0x0,0x0,0x3c990512
   6:  .stabs "/usr/src/play; /opt/SUNWspro/bin/../prod/bin/cc -c -g
busy.c -W0,-xp\$XAY9kkBSUQm8ymc.",N_CMDLINE,0x0,0x0,0x0
   7:  .stabs "main",N_MAIN,0x0,0x0,0x0
   8:  .stabs "main",N_FUN,0x0,0x0,0x0
   9:  .stabs "busy.c",N_SOL,0x0,0x0,0x0
  10:  .stabs "lazy.c",N_UNDF,0x0,0x8,0x11e
  11:  .stabs "/usr/src/play/",N_SO,0x0,0x0,0x0
  12:  .stabs "lazy.c",N_SO,0x0,0x3,0x0
  13:  .stabs "/usr/src/play",N_OBJ,0x0,0x0,0x0
  14:  .stabs "lazy.o",N_OBJ,0x0,0x0,0x0
  15:  .stabs "V=8.0;DBG_GEN=4.0.143;Xa;g;R=Forte Developer 7 C 5.4
2002/03/09;G=$XAY9kkBhUQm8Snc.",N_OPT,0x0,0x0,0x3c990521
  16:  .stabs "/usr/src/play; /opt/SUNWspro/bin/../prod/bin/cc -c -g
lazy.c -W0,-xp\$XAY9kkBhUQm8Snc.",N_CMDLINE,0x0,0x0,0x0
  17:  .stabs "lazy",N_FUN,0x0,0x0,0x0
  18:  .stabs "lazy.c",N_SOL,0x0,0x0,0x0
  19:  .stabn N_ILDPAD,0x0,0x0,0x0
  20:  .stabn N_ILDPAD,0x0,0x0,0x0
  21:  .stabn N_ILDPAD,0x0,0x0,0x0
  22:  .stabn N_ILDPAD,0x0,0x0,0x0
  23:  .stabn N_ILDPAD,0x0,0x0,0x0
  24:  .stabn N_ILDPAD,0x0,0x0,0x0
  25:  .stabn N_ILDPAD,0x0,0x0,0x0
  26:  .stabn N_ILDPAD,0x0,0x0,0x0
  27:  .stabn N_ILDPAD,0x0,0x0,0x0
  28:  .stabn N_ILDPAD,0x0,0x0,0x0
```

CHAPTER 4

Debugger Stabs

The stabs directives may be interspersed in the assembly code generated by a compiler with relatively limited constraints on their ordering. These constraints are described with each stab type and summarized in this chapter.

The stabs are described in alphabetic order by type name. For convenience, the `a.out` linker stab codes are also listed.

N_ALIAS - Symbol Alias (0x6c)

The `N_ALIAS` stab introduces an alias for a function, variable, Fortran namelist, "external redefine" or namespace. It does not make forward references. For example:

```
.stabs "newname:Foldname", N_ALIAS, 0, 0, 0
```

is used to indicate that *newname* is another name for the function named *oldname*. The `F` prefix on *oldname* indicates it is a function name. Variables are treated similarly and indicated by a `V` prefix, as in:

```
.stabs "newname:Voldname", N_ALIAS, 0, 0, 0
```

Fortran namelists are indicated by a `N` prefix to *oldname*. For example:

```
.stabs "newname:Noldname", N_ALIAS, 0, 0, 0
```

Here, *newname* is the name of the namelist and *oldname* is the body of the namelist definition. For example, the statement:

```
namelist /xx/a,b,c
```

produces:

```
.stabs “xx:Na,b,c”, N_ALIAS, 0, 0, 0
```

Note – Do not use the following descriptor unless you are sure you need to because of unpredictable output. Do not use with C++.

External redefines are indicated by a `R` prefix to *sourcename*. In the following:

```
.stabs “externalname:Rsourcename”, N_ALIAS, 0, 0, 0
```

externalname is the name of an existing external symbol, function, or variable. *sourcename* is the new name by which that external may be referenced. The primary difference between this and the `F` descriptor is that the *externalname* may exist in a different object file from this stab, and that the name translation applies to the complete load object in which the *externalname* exists.

This stab is a no-op when:

- The *externalname* does not exist.
- This stab is a second (or further) occurrence with the same *externalname* specified. If the specified sourcename already exists, then overloading will occur.

Namespace aliases result from a statement of the form:

```
namespace foo = bar;
```

This makes `foo` a namespace alias for `bar`. Namespace aliases are handled by `N_ALIAS` stabs with a prefix `S`. The stabs can be position dependent or position independent, depending on the scopes in which the aliases occur.

1. (Function) Local namespace alias, position dependent

   `.stabs “`*newname*`:S` *typeid-of-newname=typeid-of-oldname*`”, N_ALIAS, 0,0,0`

2. namespace, class, global namespace alias, position independent

`.stabs "`*mangled_newname*`:S`*typeid-of-newname*`=`*typeid-of-oldname*`:`*typeid-of enclosing-scope*`", N_ALIAS, 0,0,0`

Fortran OpenMP threadprivate variables are indicated by a T prefix to *sourcename*. The format is:

`.stabs "` *compname* `: T` *sourcename* `", N_ALIAS, 0, 0, 0`

where:

compname is a compiler-generated pointer to the threadprivate value(s) of *sourcename*, the user-defined source variable. For example, in the following stab, `xxx_` is the linker name of a user-defined OpenMP threadprivate variable, and `__tls_ptr_xxx_` is the compiler-generated linker symbol that points to a static area that contains addresses for all the threadprivate copies of `xxx_`.

```
.stabs "__tls_ptr_xxx_:Txxx_", N_ALIAS, 0, 0, 0
```

N_BCOMM — Begin Common Block (0xe2)

`.stabs "`*BlockName*`", N_BCOMM, 0, 0,` *HashValue*

A `N_BCOMM` stab introduces a named common block and precedes the listing of symbols contained in the common block. The common block is named *BlockName*. Subsequent stabs preceding an `N_ECOMM` stab specify the variables in the common block. On the Solaris operating environment, the address of the common block is found in the ELF symbol table.

An `N_ECOMM` stab terminates the listing of symbols within the named common block.

Only `N_GSYM` stabs with `V` symbol type may appear between the `N_BCOMM` and `N_ECOMM`.

`N_BCOMM` and `N_ECOMM` may not be nested.

HashValue is a hash value which, with *BlockName*, uniquely identifies the common block. If *HashValue* is zero, no `N_XCOMM` stab may reference this common block.

Unnamed common is given a *BlockName* of `"__BLNK__"`.

The following common declaration:

```
common /blk/ a, b, c
```

generates the following stabs:

```
.stabs  "blk_",N_BCOMM,0,0,0
.stabs  "a:V6",N_GSYM,0,0,0
.stabs  "b:V6",N_GSYM,0,0,4
.stabs  "c:V6",N_GSYM,0,0,8
.stabs  "blk_",N_ECOMM,0,0,0
```

The first stab is the N_BCOMM, which starts the common block. The next three stabs are N_GSYM (global symbol) stabs, which describe the three variables defined in the common block. The last stab is the N_ECOMM, which ends the common block.

N_BINCL — Begin Include File (0x82)

```
.stabs "FilePath", N_BINCL, 0, 0, HashValue
```

The N_BINCL stab begins the symbol information defined in an include file. *FilePath* is the path to the file from the source directory specified in the N_SO stab. Stabs following the N_BINCL are generated by statements in the include file.

The stabs for the include file are terminated by an N_EINCL stab.

The N_BINCL stab must follow an N_SO stab for a source file and precede the matching N_ENDM stab. N_BINCL stabs and N_EINCL stabs may be nested within other N_BINCL/N_EINCL pairs.

The *HashValue* is a number computed by the linker that is unique to this occurrence of this named include file. Since different occurrences of an include file may actually define different symbols (most commonly as the result of #ifdef/#endif statements in the include file), the *HashValue* will be different for occurrences of an include file that are not identical.

It is an error for two N_BINCL stabs to appear in the same executable or shared library with the same *FilePath* and *HashValue*.

If file h.h in the compilation directory contains the following:

```
int a;
float b;
```

when it is included the object file will contain the following stabs:

```
.stabs "./h.h",N_BINCL,0x0,0x0,0x151
.stabs "a:G(0,1)",N_GSYM,0x0,0x4,0x0
.stabs "b:G(0,9)",N_GSYM,0x0,0x4,0x0
.stabn N_EINCL,0x0,0x0,0x0
```

The first stab is the N_BINCL stab, which specifies the path to the include file h.h from the source compilation directory. The *HashValue* of 0x151 is calculated by the linker, the stab generated by the compiler contains a zero.

The next two stabs are the N_GSYM (global symbol) stabs for the two variables in the include file. The last stab is the N_EINCL stab, which ends the include file.

N_BROWS — Source Browser (0x48)

```
.stabs "bdfile", N_BROWS, 0, 0, 0
```

The N_BROWS stab specifies the path to the .bd file created to support the Source Browser. In the Solaris operating environment, this stab is entered into the .stab.sbfocus section.

N_CMDLINE - Compilation Command Line (0x34)

To support fix and continue, the N_CMDLINE stab stores the command line for compilation:

```
.stabs "cwd;driver options prefix", N_CMDLINE, 0, 0, 0
```

where:

cwd is the full path name of the working directory as specified by the -cwd option from the compiler driver.

driver is a full path to the compiler driver.

options is the list of options passed into the compiler. This list should be modified to contain only the current source file and to use the -c option to cause a .o file to be produced.

prefix is the set of options of the form needed to pass the current global prefix back into the compiler. This is usually done via -Qoption, as in the example below. Ensure that multiple fix and continue runs do not keep duplicating prefix.

Together, *driver options prefix*, make up a complete reconstructs the complete command line. This command line was passed to /bin/sh, so all special characters must have been quoted with a backslash (for example., "/$").

For example, if myfile.cc was compiled in /tmp as follows:

```
CC -I../include -g -o myprog mtfile.cc
```

the resulting string for this stab would look something like:

```
"/tmp;/path/CC -I../include -g myfile.cc -c -Qoption ccfe
   -prefix -Qoption ccfe prefix
```

N_CODETAG - Code Generation Detail (0xd8)

The N_CODETAG stab provides information about the generated code needed for certain kinds of debugging. A subset of its functionality replaces the N_PATCH stab for run time checking information (load/store and structure copy).

```
.stabn N_CODETAG, marker, 0, addr
```

where:

marker is:

0x1 CODETAG_BITFIELD	Load/store of a bit field
0x2 CODETAG_SPILL	Spill of registers
0x3 CODETAG_SCOPY	Structure copy load/store
0x4 CODETAG_FSTART	(not used) Points to first inst of new frame (0==leaf)
0x5 CODETAG_END_CTORS	(not used) End of calls to super-class constructors
0x6 (not used in stabs)	*** DW_ATCF_SUN_branch_target in dwarf ***
0x7 CODETAG_STACK_PROBE	Marks insns that probe the stack memory

addr is a byte offset of an instruction from the function label. In addition, for any module that emitted one or more N_CODETAG stabs, one index stab should be emitted of the form:

```
.xstabs "", N_CODETAG, marker, 0x0, 0x0
```

For modules that have emitted no N_CODETAG stabs, an N_CODETAG index stab should not be emitted. Currently, there are four types of N_CODETAG stabs in use:

CODETAG_BITFIELD

The CODETAG_BITFIELD stab gives the byte offset of the address of a load instruction from the function label to which the instruction belongs. This is very similar to the N_SLINE stab, but instead of the source line offset from the beginning of the function, the byte offset from the beginning of the function is emitted. Emitting a CODETAG_BITFIELD N_CODETAG stab allows runtime checking not to check for Read Uninitialized Data for that load.

Consider this C example:

```
struct s { int a:1; char b; } s1;
int i;
i = s1.a;
s1.a = 1;
```

which generates the following N_CODETAG stabs:

```
.stabs "",N_CODETAG,0x1,0x0,0x0 # index stab
.stabn N_CODETAG,0x1,0x0,0x4
.stabn N_CODETAG,0x1,0x0,0x1c
```

The statement `s1.a = 1` is considered a Bitfield Insertion, while the statement `i = s1.a` is considered a Bitfield Extraction. An N_CODETAG stab should be issued for each load instruction that is part of bitfield operation (either insertion or extraction) that load more data than the bitfield in question. That is, if the load is for the exact size of the bitfield, then an N_CODETAG stab need not be emitted.

If an insertion does not involve a load (because a store of the exact size can be done), then no N_CODETAG stab should be emitted.

CODETAG_SPILL

For each load or store instruction that is generated for register spills and unspills, one N_CODETAG stab should be emitted. This includes floating-point spills and unspills as well. These should always be emitted in matched pairs (for the spill and unspill). Emitting a CODETAG_SPILL N_CODETAG stab allows runtime checking not to perform any checking on these load or store instructions.

N_CODETAG stabs should be grouped among all the other stabs for a given function, just like N_SLINE stabs. If there are N_LBRAC and N_RBRAC stabs for a given function, then the N_CODETAG stabs should be between those stabs, as appropriate.

CODETAG_SCOPY

For each load instruction that is generated to do a structure copy, one N_CODETAG stab should be emitted. Emitting a CODETAG_SCOPY N_CODETAG stab allows runtime checking not to perform any checking on these load instructions.

N_CODETAG stabs should be grouped among all the other stabs for a given function, just like N_SLINE stabs. If there are N_LBRAC and N_RBRAC stabs for a given function, then the N_CODETAG stabs should be between those stabs, as appropriate. For example, the following C program:

```
struct s {
   char c;
   int i;
}

main() {
   struct s ss, tmp;
   ss.c = 0;
   ss.i = 1;
   tmp = ss;
}
```

generates the following N_CODETAG stabs:

```
.stabs "",N_CODETAG,0x3,0x0,0x0 # index stab
.stabn N_CODETAG,0x3,0x0,0x18
.stabn N_CODETAG,0x3,0x0,0x20
```

for the two loads in the structure copy of `tmp = ss`.

CODETAG_STACK_PROBE

For each instruction that probes stack memory before creating a frame, one N_CODETAG stab should be emitted. Emitting a CODETAG_STACK_PROBE N_CODETAG stab allows runtime checking not to perform any checking on these stack-checking instructions.

N_CODETAG stabs should be grouped among all the other stabs for a given function, just like N_SLINE stabs. If there are N_LBRAC and N_RBRAC stabs for a given function, then the N_CODETAG stabs should be between those stabs, as appropriate.

N_CONSTRUCT - Constructor Description (0xd2)

```
.stabs  "Var:State",N_CONSTRUCT,0,End-Start,Start-Func
```

Each local variable whose destruction requires a call to a destructor will cause the generation of a pair of stabs (N_CONSTRUCT and N_DESTRUCT) describing the lifetime of this variable. This includes constructor and destructor calls which are inlined. dbx needs this information in order to implement its fix and continue feature. The N_CONSTRUCT stab is associated with the location just after the construction of the variable, and the N_DESTRUCT stab is associated with the location immediately before the object's destruction is begun.

For the N_CONSTRUCT stab:

Var	is the name (if any) of the variable being constructed
State	is the new destructor state number (the state is mapped by dbx into a unique set of destructors that must be called). This is the state after the specified instructions have been executed
End	is the location of the instruction immediately following the object's construction (regardless of whether it's a call or is done inline)
Start	*is* the location of the first instruction of the construction. So *End–Start* is the number of bytes of constructor code at this spot.
Func	is the name of the current function—thus, the expression *Start–Func* is a function-relative offset.

When more than one local variable exists, the N_CONSTRUCT, N_DESTRUCT pairs are nested and the state is incremented by 1 as each constructor is seen. After executing a particular constructor, the state is *State*.

N_CPROF - Cache Profile Feedback (0xf0)

(Reserved for future use)

N_DESTRUCT - Destructor Description (0xd4)

```
.stabs "FromState:ToState",N_DESTRUCT,0,End-Start,Start-Func
```

The N_DESTRUCT stab does not mention the variable being destroyed. Instead, it merely indicates the new "destructor state" after completion of the destruction described here.

For the N_DESTRUCT stab:

S*tate* is the new destructor state number (the state is mapped by dbx into a unique set of destructors that must be called to implement its *pop* instruction or to do some kinds of *continue* operations).

End is the address of the instruction following the last instruction of destructor code.

Start is the address of the first instruction of destructor code. Thus, *End - Start* specifies the number of bytes of destructor code.

Func is the name of the current function—thus, the expression *Start – Func* is a function-relative offset.

N_ECOMM — End Common Block (0xe4)

```
.stabs "BlockName", N_ECOMM, 0, 0, 0
```

A N_ECOMM stab terminates the listing of symbols defined within the named common block that was begun by an N_BCOMM stab.

N_BCOMM and N_ECOMM stabs may not be nested.

Unnamed common is given a *BlockName* of "__BLNK__".

For an example, see "N_BCOMM — Begin Common Block (0xe2)" on page 23.

N_EINCL — End Included File (0xa2)

```
.stabn N_EINCL, 0, 0, 0
```

The N_EINCL stab terminates the specification of stabs contained within an include file specified by the closest preceding N_BINCL stab. N_BINCL and N_EINCL stabs may be nested.

For an example, see "N_BINCL — Begin Include File (0x82)" on page 24.

N_EMOD - Fortran90 Module End

```
.stabs "Name", N_EMOD, 0, 0, 0
```

The N_EMOD stab ends the sequence of stabs that belong to the named module, which must start with a corresponding N_MOD stab.

See "N_MOD - Fortran 95 Module Begin" on page 45 for an example.

N_ENDM — End Module (0x62)

```
.stabn N_ENDM, 0, 0, 0
```

The N_ENDM stab terminates the stabs for the source file started by an N_SO. It must follow all of the stabs for the source file; no file-related stabs may follow the N_ENDM. This stab is required even if no other debugging stabs are generated.

The C file:

```
foo()
{
}
```

generates (in part) the following stabs:

```
.stabs "/tmp/",N_SO,0x0,0x0,0x0
.stabs "c.c",N_SO,0x0,0x3,0x0
.stabn N_ENDM,0x0,0x0,0x0
```

The first two N_SO stabs give the compilation directory and file name. The last stab is the N_ENDM, which ends the stab listing for the object file.

N_ENTRY — Fortran Alternate Entry (0xa4)

```
.stabs "Name : e RtnType; FunName ; ;", N_ENTRY, 0, Line, 0
```

The N_ENTRY stab is generated for an alternate entry into a Fortran function named *Name*. *RtnType* is the type of the value returned from the function. *FunName* is the name of the function that contains this entry point. If *Line* is not zero, this is the line number of the ENTRY statement.

Parameters passed to the entry are represented by N_PSYM (parameter) stabs following the N_ENTRY stab.

The following Fortran subroutine:

```
subroutine a(i)
print *, i
return

entry b(f)
print *, f
return
end
```

generates the following stabs:

```
.stabs "a:F14",N_FUN,0,0,_a_
.stabs "i:v3",N_PSYM,0,0,-8
.stabs “__entry:3”,N_LSYM,0,4,0xffffffb0
.stabs "b:e14;a_;;",N_ENTRY,0,5,0
.stabs "f:v6",N_PSYM,0,0,-4
```

The first N_FUN stab describes the subroutine a and is followed by an N_PSYM stab for its argument i. The third stab is generated, by convention, with the special name __entry. This specially named variable contains the address through which this procedure was entered (useful in determining a correct traceback) . This special symbol is present only in procedures with secondary entry points. The fourth stab is the N_ENTRY stab, which indicates that b is an entry point within the subroutine a. The name given in the stab is that of the label generated for the function, rather than the name that the user entered. It is followed by an N_PSYM stab for the argument to the entry.

N_ESYM — Position-independent External Data Type (0xc8)

```
.stabs "Name : SymDesc Type", N_ESYM, 0, Desc, Value
```

The N_ESYM stab is used in place of the N_LSYM stab in situations where the position of the stab in relation to the other stabs does not reflect the proper scoping. This occurs for C++ 5.0 templates and is a consequence of the compiler’s “on-demand” method of generating stabs for types. For example, if template instance stack<int> references a type “ctype” that is defined outside the template (in another file), an N_LSYM stab could not properly convey the scope in which ctype was defined.

The N_ESYM stab is used for data types that have external linkage (class, struct, union, enum). There is a similar N_ISYM stab for types with internal linkage. N_ESYM stabs, being external, also appear in the index stabs section.

N_ESYM uses the same syntax as N_LSYM; it differs only in that its name is always mangled. The name is demangled to obtain the scope information. If there is no “::” in the demangled name, the data type is in the global scope.

N_FLSYM -- Fragmented Data Symbol (0x2e)

```
.stabs " Name : SymDesc Type ", N_FLSYM, OpenMP, Size, 0
```

The N_FLSYM stab defines a global symbol with the name *Name*. The symbol represents a static local variable that has become global to the linker because it has been placed in a separate ELF section by the compiler.

SymDesc is a symbol descriptor and may be one of the following:

- G Global weak (C++)
- S File
- V Local

If the N_FLSYM stab appears outside of a function, only *SymDesc* S is permitted.

OpenMP is equal to N_SYM_OMP_TLS (see stab.h) when an OpenMP variable has been declared THREAD_PRIVATE. It is zero otherwise.

Size is the size of the variable in bytes.

The following C function, compiled with -xF=lcldata:

```
void foo ()
{
static int x;
}
```

generates the following stab for x:

```
.stabs "$XBY9kkB4DBZ_SXV.foo.x:V(0,3)",N_FLSYM,0x0,0x4,0x0
```

where $XBY9kkB4DBZ_SXV.foo.x is the globalized name for x.

N_FUN — Function or Procedure Definition (0x24)

```
.stabs "Name : SymDesc RtnType [ ; ArgType ]*",     N_FUN, FunKind,
Line, 0
```

An N_FUN stab defines the beginning of a function or procedure, or it describes a prototype for the specified function or procedure.

For the N_FUN stab:

Name is the name of the function or procedure.

SymDesc is a symbol descriptor and may be one of the following:

F	Global function or procedure
f	Local function or procedure
M	Module function (Fortran 95)
J	Internal procedure (Fortran 95)
P	Prototype for function or procedure
Y	C++ specification (see “C++ Specification (Y)” on page 88)

RtnType is the type of the returned value.

An optional list of *ArgType* entries separated by semicolons may follow the *RtnType*. These are the (possibly) promoted types of the formal arguments to the function, starting with the left-most argument. A 0 (zero) may be entered as the last *ArgType* to specify that this and subsequent argument types are unspecified and that all following types or number of arguments are valid.

ArgType is the (possibly) promoted type of the formal argument to the function. These may be different from the type of the formal argument. For example, in non-ANSI C, when not using prototypes, integer types shorter than an int are promoted to int. The *ArgType* shows that an int is passed to the function while the N_PSYM stab for the formal argument shows the formal argument’s declared type.

FunKind is currently used to qualify the kinds of Fortran 95 subprograms as follows:

1 = Pure
2 = Elemental
4 = Recursive

More than one kind may be specified, they are not all mutually exclusive.

In an index stab, *FunKind* `== 1` indicates that the stab came from a COMDAT section.

FunKind is normally zero, and all other possible values are reserved for future use.

Line is the source line on which the function definition begins.

All of the stabs that follow the `N_FUN` stab describe symbols and types that are defined within the function. The function's stabs are ended by the next `N_FUN`, `N_ENDM`, `N_ENTRY` stab or by the closing `N_RBRAC` for the function (one that specifies level zero).

The following C program:

```
#include <stdio.h>

int main (int argc, char **argv)
{
   printf ("%d %s\n", argc, *argv);
}
```

generates the following stabs:

```
.stabs  "main:F(0,3);(0,3);(0,20)=*(0,21)=*(0,1)",
        N_FUN,0,0,_main
.stabs  "argc:p(0,3)",N_PSYM,0,4,68
.stabs  "argv:p(0,20)",N_PSYM,0,4,72
.stabs  "printf:P(0,3);(0,22)=*(0,1);0",N_FUN,0,0,0
```

The first stab is an `N_FUN` stab, which describes `main` as a function that returns an `int (type (0,3))` and takes two arguments: an `int`, and a pointer to a pointer to a `char (type (0,1))`. The next two stabs are `N_PSYM` (parameter) stabs that describe the two arguments. The last `N_FUN` stab gives a prototype of the function `printf`.

N_FUN_CHILD -- Function Child (0xd9)

```
.stabs "Name", N_FUN_CHILD, 0, 0, 0
```

The N_FUN_CHILD stab is created when a nesting relationship between functions needs to be communicated to the debugger. This stab should appear after the parent function N_FUN stab and the *Name* in this N_FUN_CHILD stab should refer to the nested (child) function. This child function's definition would appear in a separate N_FUN stab set. Any function's stabs set can contain any number of N_FUN_CHILD stabs, one for every unique child function.

The following Fortran95 program:

```
        call sub
        end

        subroutine sub
        real a(100,100)
        integer i, j, k
        i = 100
        j = i
        k = i*j
!$omp parallel
!$omp do
        do i = 1,100
            do j = 1,100
                a(i,j) = i + j*100
            enddo
        enddo
$omp end do
        print *,i
$omp end parallel
        print *,k
        end
```

when compiled with `-g -xopenmp=noopt`, produces (in part) these stabs:

```
39:  .stabs "sub_:F1",N_FUN,0x0,0x4,0x0
40:  .stabn N_LBRAC,0x0,0x1,0x0
41:  .stabs "_$p1B10.sub_",N_FUN_CHILD,0x0,0x0,0x0
51:  .stabn N_RBRAC,0x0,0x1,0x214
52:  .stabs "_$d1A11.sub_:f1",N_FUN,0x0,0x0,0x0
53:  .stabn N_LBRAC,0x0,0x1,0x0
61:  .stabn N_RBRAC,0x0,0x1,0x170
62:  .stabs "_$p1B10.sub_:f1",N_FUN,0x0,0x0,0x0
63:  .stabn N_LBRAC,0x0,0x1,0x0
64:  .stabs "_$d1A11.sub_",N_FUN_CHILD,0x0,0x0,0x0
67:  .stabn N_RBRAC,0x0,0x1,0x17c
```

which shows that `_$p1B10.sub_` is a child of `sub_` (stab 41) and `_$d1A11.sub_` is a child of `_$p1B10.sub_` (stab 64). Both of these are compiler-generated OpenMP functions.

N_GSYM — Global Symbol (0x20)

```
.stabs "Name : SymDesc Type", N_GSYM, 0, Desc, Value
```

The `N_GSYM` stabs defines a global symbol with the name *Name*. A symbol may be a type name or a variable name. The *Name* field is followed by a colon and type specification.

The exact meaning of the *Desc* and *Value* fields depends on the value of *SymDesc*.

The C statements (outside of a function):

```
struct S {
   int a;
   int b;
   } Z;
int X;
```

generate the following stabs:

```
.stabs  "S:T(0,20)=s8a:(0,3),0,32;b:(0,3),32,32;",N_LSYM,0,8,1
.stabs  "Z:G(0,20)",N_GSYM,0,8,0
.stabs  "X:G(0,3)”,N_GSYM,0,4,0
```

The first stab is an `N_LSYM` (local symbol) stab, which describes the structure. The next two `N_GSYM` stabs specify that `Z` is an occurrence of that structure and that `X` is an `int (type (0,3))`.

`N_ILDPAD` - Incremental Link Padding (`0x4c`)

The `N_ILDPAD` stab is a linker stab that indicates that the string table position should be modified.

```
.stabs “Objname”, N_ILDPAD, 0, 0, BytesOfStringTable
```

The stab indicates that the string table position should be adjusted by *BytesOfStringTable* bytes.

`N_ISYM` — Position-independent Internal Data Type (`0xc6`)

```
.stabs "Name : SymDesc Type", N_ISYM, 0, Desc, Value
```

The `N_ISYM` stab is used in place of the `N_LSYM` stab in situations where the position of the stab in relation to the other stabs does not reflect the proper scoping. This occurs for C++ 5.0 templates and is a consequence of the compiler’s “on-demand” method of generating stabs for types. For example, if template instance `stack<int>` references a type “ctype” that is defined outside the template (in another file), an `N_LSYM` stab could not properly convey the scope in which ctype was defined.

The N_ISYM stab is used for data types that have internal linkage (typedefs). There is a similar N_ESYM stab for types with external linkage.

N_ISYM uses the same syntax as N_LSYM; it differs only in that its name is always mangled. The name is demangled to obtain the scope information. If there is no "::" in the demangled name, the data type is in file scope.

N_LBRAC — Begin Scope (0xc0)

```
.stabn N_LBRAC, 0, Level, Offset
```

The N_LBRAC stab indicates the start of a scope. This is usually a group of statements delimited by curly braces in C or by BEGIN and END in Pascal. The scope is terminated by an N_RBRAC stab, or by an N_FUN stab which implicitly closes all scopes opened by a N_LBRAC.

Level is the nesting level of the scope, with the outermost scope considered to be level zero. *Level* will always be one or more.

The N_LBRAC stab must appear within a function (started by an N_FUN stab). All stabs that appear between the N_LBRAC and N_RBRAC stabs are considered to be within the defined scope (with the exception of global scope stabs, such as N_GSYM).

N_LBRAC and N_RBRAC stabs may be nested.

Nested functions (Pascal, Fortran 95) are linearized nested-most first, because an N_FUN stab of a nested function would cancel the N_LBRAC stab of the enclosing function. The *Level* of such a nested function's N_LBRAC stab is always 1.

In an a.out file, *Offset* is the byte offset from the start of the object file; in an ELF file, it is the byte offset from the start of the function.

The following program:

```
main ()
{
    int i = 5;
    {
       float i = 5.5;
       printf ("%f\n", i);
    }
    printf ("%d\n", i);
}
```

generates the following stabs:

```
.stabs "main:F(0,3)",N_FUN,0,0,main
.stabs "main", N_MAIN, 0,0,0
.stabn N_LBRAC,0,1,.LL1-main
.stabs "i:(0,3)",N_LSYM,0,4,-4
.stabn N_SLINE,0,3,.LL2-main
.stabn N_LBRAC,0,2,.LL3-main
.stabs "i:(0,16)",N_LSYM,0,4,-8
.stabn N_SLINE,0,6,.LL4-main
.stabn N_SLINE,0,7,.LL5-main
.stabn N_RBRAC,0,2,.LL6-main
.stabn N_SLINE,0,10,.LL7-main
.stabn N_SLINE,0,11,.LL8-main
.stabn N_RBRAC,0,1,.LL9-main
```

The first stab is an `N_FUN` (function) stab, which starts stabs for the function `main`. It is followed by an `N_MAIN` stab, which indicates that this is the main function. These are followed by an `N_LBRAC` `stab`, which specifies level one and gives the address of the first instruction in the scope. This is followed by an `N_LSYM` (local symbol) stab of the `int` variable `i` defined in this scope, and an `N_SLINE` (line number) stab for the line containing the first call to `printf`.

The sixth stab is a second `N_LBRAC` stab, indicating scoping-level two. It is followed by another `N_LSYM` stab for a new declaration of `i` and two `N_SLINE` stabs. These are followed by an `N_RBRAC` (end scope) stab for level two, more `N_SLINE` stabs, and a final `N_RBRAC` stab for scope-level one.

`N_LCSYM` — Uninitialized Static Symbol (`0x28`)

```
.stabs "Name : SymDesc Type", N_LCSYM, OpenMP, Size, Offset
```

An `N_LCSYM` stab describes an uninitialized static variable.

SymDesc is a symbol descriptor and may be one of the following:

`G`	Global weak (C++)
`S`	File
`V`	Local
`b`	Fortran “based” variable

OpenMP is equal to `N_SYM_OMP_TLS` (see `stab.h`) when an OpenMP variable has been declared `THREAD_PRIVATE`. It is zero otherwise.

Size is the size of the symbol in bytes.

In an ELF file, *Offset* is the number of bytes into the object file’s uninitialized data (`bss`) area. This is identified by a linker symbol table entry for the local symbol "`Bbss.bss`" that has the address of the first byte of the `bss` area used by the object file. This symbol can be defined by the following assembly statements:

```
    .section ".bss"
Bbss.bss:
```

This creates a symbol table entry for `Bbss.bss` with the type `STT_NOTYPE`.

If the *Name* is globalized, as can happen when the object file has been built for the fix and continue feature of `dbx`, the *Name* will be found in the ELF symbol table and the *Offset* will be zero.

The only exception is for Fortran 95 pointer-based variables, where *Name* is the simple user name and *Offset* is the number of bytes from the start of `bss`.

If the `N_LCSYM` stab appears outside of a function, only *SymDesc* `S` is permitted.

The following C function:

```
foo ()
{
    static int x;
}
```

generates the following stab for `x:`

```
.stabs "x:V(0,3)",N_LCSYM,0,4,.L15-Bbss.bss
```

N_LSYM — Local Symbol (0x80)

```
.stabs "Name : SymDesc Type", N_LSYM, 0, Desc, Value
```

The N_LSYM stab describes either a file local or a function local symbol. A symbol may be a type name or a variable name. The *Name* field is a followed by a colon and a type specification.

The exact meaning of the *Desc* and *Value* fields depend on the type descriptor *Type*.

In the following C file:

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

generates (in part) the following stabs:

```
.stabs "char:t(0,1)=bsc1;0;8;",N_LSYM,0,0,0
.stabs "short:t(0,2)=bs2;0;16;",N_LSYM,0,0,0
.stabs "int:t(0,3)=bs4;0;32;",N_LSYM,0,0,0
.stabs "foo:F(0,3)",N_FUN,0,0,_foo
.stabs "x:(0,3)",N_LSYM,0,4,-4
.stabn N_ENDM, 0,0,0
```

The first three N_LSYM stabs describe standard int, short, and char types. Because these stabs appear before the N_FUN (function) stab that starts the stabs for function foo, they are file-local symbols. The N_LSYM stab for x appears after the N_FUN and before the N_ENDM (end module) stab, so it is local to the function foo.

N_MAIN — Main Routine Name (0x2a)

```
.stabs "Name", N_MAIN, 0, 0, 0
```

The N_MAIN stab specifies the name of the first user function executed in the program. In a C program, this is usually "main"; in Pascal, "program"; in Fortran "MAIN". This stab must always be generated for the main routine. It must also be generated in the .stab.index section.

There may only be a single N_MAIN stab in any executable file.

N_MOD - Fortran 95 Module Begin

```
.stabs "Name: MemberList", N_MOD, 0, 0, 0
```

The N_MOD stab begins the definition of a Fortran 95 module. All stabs after an N_MOD stab and before the terminating N_EMOD stab define the variables and subprograms contained within the module.N_MOD/N_EMOD pairs cannot be nested.

MemberList is a sequence of members separated by semicolons. A double semicolon ";;" terminates the list. Each member consists of a ppp-code followed by the member's name. See "The Components of the Class Stab" on page 75 for a description of ppp-codes. Currently, only ppp-codes "A" (private) and"C" (public) are used with module members.

For example:

```
 module bbb
   integer:: l_bbb=9, m_bbb=99, n_bbb=999, n1_bbb=9999
subroutine s1_bbb
print *, 's1_bbb'
end subroutine s1_bbb
end module bbb
```

produces the following stabs:

```
 .stabs
"bbb:Cbbb.l_bbb_;Cbbb.m_bbb_;Cbbb.n1_bbb_;Cbbb.n_bbb_;;",N_MOD,0
x0,0x0,0x0
.stabs "l_bbb:V(0,4)",N_STSYM,0x0,0x4,0x4
.stabs "m_bbb:V(0,4)",N_STSYM,0x0,0x4,0x8
.stabs "n1_bbb:V(0,4)",N_STSYM,0x0,0x4,0xc
.stabs "n_bbb:V(0,4)",N_STSYM,0x0,0x4,0x10
.stabs "bbb.s1_bbb:F(0,1)",N_FUN,0x0,0x0,0x0
.stabs "s1_bbb:W(0,0);bbb.s1_bbb_ ;;;",N_LSYM,0x0,0x0,0x0
.stabs "bbb.f90",N_SOL,0x0,0x0,0x0
.stabn N_SLINE,0x0,0x7,0x4
.stabn N_SLINE,0x0,0x8,0x48
.stabn N_LBRAC,0x0,0x1,0x0
.stabn N_RBRAC,0x0,0x1,0x48
.stabs "bbb",N_EMOD,0x0,0x0,0x0
```

N_OBJ — Object Directory and File (0x38)

```
.stabs "ObjectDir", N_OBJ, 0, 0, 0
.stabs "ObjectFile", N_OBJ, 0, 0, 0
```

Two N_OBJ stabs identify the current working directory where the linker was executed and the path to the object file from that directory. The N_OBJ stabs are generated by the compiler with null strings for the directory and file path. These are later filled by the linker, which places the current working directory, without trailing slash, in the first stab, and the file in the second.

When the linker brings in an object file from an archive library, *ObjectDir* is the name of the directory for the archive library, and the name of the object file is the name of archive library followed by the name of the object file in parentheses.

N_OPT — Options (0x3c)

```
.stabs "Options", N_OPT, 0, 0, TimeStamp
```

The N_OPT stab specifies various options that were used to compile the source file and the time the object file was created. *TimeStamp* is in the format returned by time(2).

The options are strings separated by semicolons. Options may have an argument that is specified after an equal sign. Blanks may precede or follow the option.

The options that may be specified are listed below:

A=2	Compiled ABI2 (the C++ 5.0 default)
dbl	Fortran dbl flag
dm	C++ differential mangling
cd	COMDAT used
DBGGEN=*version*	*version* of DBGGEN used
g	Debugging stabs were generated for this object file
F	Fragmented
G=<g>	Global prefix is <g>
nu	No underscores added to Fortran symbols
O	Optimized code
p	-Kpic used
P	-KPIC used
R=xx.xx<r>	Compiler release number is xx.xx (“r” is a version string)
r8	Fortran r8 flag
U	Fortran mixed case variables
V=<v>	Stab version is <v> (must be a numeric string)
Xa	ANSI C promotions

Every object file must have an N_OPT stab. If debugging stabs were generated for the object file, the -g option must be specified. A version number must also be specified (this example reflects version 2.0):

```
.stabs "Xt ; g ; V=2.0", N_OPT, 0, 0, 0x02040608
```

The same time stamp must be in the N_OPT stab in the .stab.index section.

N_OUTL - Outlined Function

```
.stabs "Name", N_OUTL, 0, 0, 0
```

The N_OUTL stab is produced by an optimizing code generator when a portion of the generated code is separate from the main body of the function, and source line information for the code exists in the form of N_SLINE stabs.

The offsets in the N_SLINE stabs are from the start of the outlined function "*Name*".

The *Name*, by convention, is related to the name of the original function.

N_PATCH - Patch Run Time Checker (0xd0)

The N_PATCH stab provides information to the run time checker to inhibit the checking of load and store instructions which are generated purely for internal reasons (not associated with user code).

```
.stabn N_PATCH, 0, patchtype, addr
```

where:

patchtype is:

`0x1` `P_BITFIELD`: read of bitfield container

`0x2` `P_SPILL`: register spill or unspill

`0x3` `P_SCOPY`: load used for structure copy

addr is a byte offset of an instruction from the function label. In addition, for any module that emitted one or more `N_PATCH` stabs, *one* index stab should be emitted of the form:

```
.xstabs “”, N_PATCH, 0x0, 0x0, 0x0
```

For modules which have emitted no `N_PATCH` stabs, an `N_PATCH` index stab should *not* be emitted. Currently, there are three types of `N_PATCH` stabs:

P_BITFIELD

The `P_BITFIELD` stab gives the byte offset of the address of a load instruction from the function label to which the instruction belongs. This is very similar to the `N_SLINE` stab, but instead of the source line offset from the beginning of the function, the byte offset from the beginning of the function is emitted. Emitting a `P_BITFIELD N_PATCH` stab allows runtime checking not to check for “Read Uninitialized Data” for that load.

Consider this C example:

```
struct s { int a:1; char b; } s1;
int i;
i = s1.a;
s1.a = 1;
```

The statement `s1.a = 1` is considered a *Bitfield Insertion*, while the statement `i = s1.a` is considered a *Bitfield Extraction*. An `N_PATCH` stab should be issued for each load instruction which is part of bitfield operation (either insertion or extraction) that load more data than the bitfield in question. That is:

- If the load is for the exact size of the bitfield, then an `N_PATCH` stab need not be emitted.
- If an insertion does not involve a load (because a store of the exact size can be done), then no `N_PATCH` stab should be emitted.

P_SPILL

For each load or store instruction which is generated for register spills and unspills, one N_PATCH stab should be emitted. This includes floating-point spills/unspills as well. These should always be emitted in matched pairs (for the spill and unspill). Emitting a P_SPILL N_PATCH stab will allow runtime checking not to perform any checking on these load or store instructions.

N_PATCH stabs should be grouped among all the other stabs for a given function, just like N_SLINE stabs. If there are N_LBRAC and N_RBRAC stabs for a given function, then the N_PATCH stabs should be between those stabs, as appropriate.

P_SCOPY

For each load instruction which is generated to do a structure copy, one N_PATCH stab should be emitted. Emitting a P_SCOPY N_PATCH stab will allow runtime checking not to perform any checking on these load instructions.

N_PATCH stabs should be grouped among all the other stabs for a given function, just like N_SLINE stabs. If there are L_BRAC and R_BRAC stabs for a given function, then the N_PATCH stabs should be between those stabs, as appropriate. For example, the following C program:

```
struct s {
    char c;
    int i;
}

main() {
    struct s ss, tmp;
    ss.c = 0;
    ss.i = 1;
    tmp = ss;
}
```

generates the following N_PATCH stabs:

```
.stabs “”,N_PATCH,0x0,0x0,0x0
.stabn N_PATCH,0x0,0x3,0x1c
.stabn N_PATCH,0x0,0x3,0x24
```

for the two loads in the structure copy of tmp = ss.

N_PSYM — Formal Parameter (0xa0)

```
.stabs "Name : SymDesc Type[;snumber]", N_PSYM, 0, 0, Offset
```

An N_PSYM stab describes a procedure or function parameter, giving its name, kind, type and argument offset.

SymDesc is a symbol descriptor that describes the type of the parameter. It may be one of the following:

b	Based variable (offset refers to address; implies an extra level of indirection)
C	Conformant array bound
d	Allocatable array (dope vector)
p	Value parameter
v	Fortran variable parameter by reference
x	Value parameter by reference
X	Function result variable

If the optional ;s*number* is specified, it indicates a C99 static size situation such as the following example, which means that x is passed as a pointer to the first element, and is always guaranteed to be 33 elements long:

```
int foo(int x[static33]) {...}
```

The N_PSYM stab follows the N_FUN stab, which describes a function or procedure. For register parameters an N_RSYM is also generated.

For an example of the N_PSYM stab, see "N_FUN — Function or Procedure Definition (0x24)" on page 36.

N_RBRAC — End Scope (0xe0)

```
.stabn N_RBRAC, 0, Level, Offset
```

The N_RBRAC stab ends a scope that was initiated by an N_LBRAC stab. *Level* is the nesting level of the scope, and must be the same as the matching N_LBRAC stab. If an N_LBRAC stab appears following an N_FUN (and subsequent N_PSYM stabs), it is taken to start the scope of the function. The matching N_RBRAC ends the scope of the function.

An N_SLINE stab should be generated before each N_RBRAC stab that ends a loop or a function. This permits "stopping" before exiting the scope.

The N_LBRAC and N_RBRAC stabs may be nested.

In an a.out file, *Offset* is the number of bytes from the start of the object file; in an ELF file, it is the number of bytes from the start of the function.

For an example of the N_RBRAC stab, see "N_LBRAC — Begin Scope (0xc0)" on page 41.

N_READ_MOD - Fortran 95 Module Use

```
.stabs "Name [ : ] [ only; ] [ NameList ]
```

The N_READ_MOD stab describes a Fortran 95 use statement. It consists of the *Name* of a module, an optional "only;" indicator, and an optional *NameList*. The colon after the name is required if either or both of the optional parts are present.

NameList is a sequence of zero or more name associations separated by commas. Each name association is either a single identifier, or a pair of identifiers separated by a space. The pair is in the order "local_name" followed by "original_name". There is no termination character.

For example:

```
use aaa
```

produces:

```
.stabs "aaa",N_READ_MOD,0x0,0x0,0x0
```

```
use bbb,only: l_bbb, m_bbb, my_n=>n_bbb
```

produces:

```
 .stabs "bbb:only;l_bbb,m_bbb,my_n n_bbb",N_READ_MOD,0x0,0x0,0x0
```

N_ROSYM — Read-Only Static Symbol (0x2c)

```
.stabs "Name : SymDesc Type", N_ROSYM, Flag, 0, Offset
```

An N_ROSYM stab describes a read-only initialized static variable.

SymDesc is a symbol descriptor and may be one of the following:

G	Global weak (not used)
S	File variable
V	Local static variable
b	FORTRAN based variables

If *Flag* is 0, *Offset* is the number of bytes into the object file's read-only data area. This is identified by a linker symbol table entry for the local symbol Drodata.rodata that has the address of the first byte of the data area used by the object file. This symbol can be defined by the following assembly statements:

```
    .section ".rodata"
Drodata.rodata:
```

If *Flag* is 1, Offset is the number of bytes into the object file's position independent area, identified by the linker symbol `Dpicdata.picdata`, which can be defined by the following assembly statements:

```
    .section ".picdata"
Dpicdata.picdata
```

This creates a symbol table entry with the type `STT_NOTYPE`.

If the *Name* is globalized, it will be found in the ELF symbol table and *Offset* will be zero.

In an `a.out` file, *Offset* contains the address of the variable.

If the `N_ROSYM` stab appears outside of a function, only file variables (*SymDesc* is `S`) are permitted. The address of the symbol is found from the linker stab with the same name. *Offset* is ignored.

`N_RSYM` — Register Symbol (`0x40`)

```
.stabs "Name : SymDesc Type", N_RSYM, 0, Size, Number
```

An `N_RSYM` stab describes a register variable or parameter, giving its name, size in bytes, type, and register number. A formal parameter also has an `N_PSYM` stab or an `N_RSYM` stab with symbol descriptor `p` (parameter).

SymDesc is a symbol descriptor and may be one of the following:

`p`	Register parameter
`r`	Register variable

The register numbers for the SPARC platform are assigned as follows:

Integer global registers (`g0-g7`)	0 - 7
Integer out registers (`o0-07`)	8 - 15

Integer local registers (`l0-l7`)	16 - 23
Integer in registers (`i0-i7`)	24 - 31
Floating point registers (`f0-f31`)	32 - 63

The following C function

```
foo(register int j)
{
   register int x = j;
}
```

generates the following stabs:

```
.stabs "foo:F(0,3);(0,3)",N_FUN,0,0,_foo
.stabs "j:p(0,3)",N_RSYM,0,4,24
.stabs "j:r(0,3)",N_RSYM,0,4,24
.stabs "x:r(0,3)",N_RSYM,0,4,29
```

The first stab is an `N_FUN` (function) stab that starts function `foo`. It is followed by two `N_RSYM` stabs for the parameter `j`, the first of which describes it as a parameter, and the second describes it as a register variable. The last stab describes `x` as a register variable.

`N_SLINE` — Source Line (`0x44`)

```
.stabn N_SLINE, DestructorInfo, Line, Offset
```

The `N_SLINE` stab indicates the location of the leading instruction of a contiguous block of instructions generated for a source line. *Line* specifies the number of the line in the source file described by the closest preceding `N_SO` or `N_SOL` stab, numbering from line one. In an ELF file, *Offset* is the number of bytes from the start of the enclosing function described by the preceding `N_FUN`. In an a.out file, *Offset* is the instruction address.

DestructorInfo is encoded into the *other* field of the stab, as a signed four-bit value, the low-order four bits in that field. It indicates a change in the destructor state number. This change should be added to the current state number to get the destructor state at this location. See the `N_CONSTRUCT` section for context. [The other

(upper) four bits of the *other* field are reserved for future use.] A value of 0x88 in the entire field indicates an overflow in the *DestructorInfo* field; in that case, the correct value is given as a decimal number in the string for this N_SLINE stab, along with something else:

```
.stabs "0:DestructorInfo",N_SLINE,0x88, Line, Offset
```

N_SLINE stabs may only appear within functions and must be in ascending order by *Offset*. There may be more than one N_SLINE stab generated for a given source line, and they may be in any order by line. If a source line does not have any executable code associated with it, there may not be an N_SLINE stab generated for the line.

The number 0 is not allowed in the *Line* field. If the compiler cannot attribute an instruction to any particular source line within a function, then the first source line of the function definition should be used.

An N_SLINE stab offset must be generated for an N_RBRAC offset that represents the end of a loop or end of a function. This supports stopping before exiting the scope.

Offset is the number of bytes from the start of the enclosing function described by the preceding N_FUN stab or N_OUTL stab. N_SLINE stabs following an N_ENTRY are associated with that entry, but the *Offset* is still from the N_FUN stab.

For example:

```
function fuzzy()
integer fuzzy, eeee
fuzzy = 0
return
entry eeee()
eeee = 1
return
end
```

produces the following stabs:

```
...
.stabs "/home/dmf/dbx_stuff/",N_SO,0x0,0x0,0x0
.stabs "sline.f",N_SO,0x0,0x7,0x0
.stabs "fuzzy:F(0,4)",N_FUN,0x0,0x0,0x0
.stabn N_SLINE,0x0,0x3,0x10
.stabn N_SLINE,0x0,0x4,0x14
.stabs "eeee:e(0,4);fuzzy_;;",N_ENTRY,0x0,0x2,0x0
.stabn N_SLINE,0x0,0x5,0x2c
.stabn N_SLINE,0x0,0x6,0x40
.stabn N_SLINE,0x0,0x7,0x48
.stabn N_SLINE,0x0,0x8,0x50
.stabn N_LBRAC,0x0,0x1,0x0
.stabs "__entry:(0,22)",N_LSYM,0x0,0x4,0xfffffffc
.stabs "fuzzy:(0,4)",N_LSYM,0x0,0x4,0xfffffff8
.stabs "eeee:(0,4)",N_LSYM,0x0,0x4,0xfffffff8
.stabn N_RBRAC,0x0,0x1,0x58
...
```

N_SO — Source Directory and File (0x64)

```
.stabs "SourceDir", N_SO, 0, 0, 0
.stabs "SourceFile", N_SO, 0, LangCode, 0
```

Two N_SO stabs identify the current working directory where the code was compiled, and the path to the source file that produced this object code. They must be the first two debugging stabs generated for an object file that contains debugging stabs. The stabs for the object file are terminated by an N_ENDM stab.

SourceDir is the name of the directory, terminated by a slash.

SourceFile is the name of the source file in the form it was given to the compiler.

LangCode describes the language of the source file. If it is zero, the debugger is expected to identify the source language by other means, for example, by the suffix of the source file name.

The language codes are as follows:

N_SO_AS	1	assembler source
N_SO_C	2	K & R C source
N_SO_ANSI_C	3	ANSI C source
N_SO_CC	4	C++ source
N_SO_FORTRAN	5	Fortran source
N_SO_PASCAL	6	Pascal source (not used)
N_SO_FORTRAN90	7	Fortran 95 source
N_SO_JAVA	8	Java source
N_SO_C99	9	C99 source

For example, if the current working directory is /usr/example/test and we compile the source file ../src/ex.cc, then the following stabs will be generated before the start of the other stabs for the file:

```
.stabs "/usr/example/test/",N_SO,0,0,0
.stabs"../src/ex.cc",N_SO,0,4,0
```

There may only be a single pair of N_SO stabs generated for each object file; they must precede all of the stabs for that object file except the N_UNDF stab, which is always first.

When using a translator with another compiler (such as `cfront` generating C code for the C compiler), the translator and compiler need to insure that only one pair of `N_SO` stabs is generated. In general, this means that the translator should generate the `N_SO` stabs, and the compiler should not generate any `N_SO` stabs.

`N_SOL` — Included File (`0x84`)

```
.stab "FilePath", N_SOL, 0, Line, 0
```

The `N_SOL` stab specifies the actual source file that generated executable code, type definitions, and variable definitions described by subsequent `N_SLINE` stabs. The *FilePath* is relative to the *SourceDir* specified in the preceding `N_SO` stabs.

`N_SOL` index stabs appear only in the case of executable code, and do not have subsequent stabs which depend upon them. These index stabs aid the debugger in its algorithms for locating code in which to set breakpoints, etc.

Line is the source line number of the #*include* directive that caused this file to be included. The line is assumed to be in the file specified by the preceding `N_SOL` or `N_SO` stab. It is not used in an `N_SOL` index stab.

The `N_SOL` stab must be generated for code generated within an include file. It may also be used to indicate that the original source is different from the source file passed to the compiler, for example, to specify the name of a file processed by `yacc`. It is usually generated immediately before any stab that contains a source line not from the file named in the preceding `N_SOL` or `N_SO` stab. This includes, but is not limited to, `N_SLINE`, `N_FUN`, `N_ENTRY`, and `N_SOL` stabs.

If the include file `h.h` contains the line:

```
i = 5;
```

and the C source file contains:

```
main()
{
   int i;

#include "h.h"
}
```

the following stabs are generated:

```
.stabs "./h.h",N_BINCL,0,0,0
.stabs "./h.h",N_SOL,0,5,0
.stabn N_SLINE,0,1,4
.stabn N_EINCL,0,0,0
.stabs "c.c",N_SOL,0,0,0
```

The first stab is an `N_BINCL` (start include file) that starts the include file. This is followed by an `N_SOL` stab that indicates that the following executable code appeared in `h.h`. The third stab is an `N_SLINE` (source line) for the assignment in the include file. The fourth stab is an `N_EINCL` (end include file) stab. The last stab indicates that subsequent source is from `c.c`, the original source file.

`N_STSYM` — Initialized Static Symbol (`0x26`)

```
.stabs "Name : SymDesc Type", N_STSYM, OpenMP, Size, Offset
```

An `N_STSYM` stab describes an initialized static variable.

SymDesc is a symbol descriptor. It may be one of the following:

`G`	Global weak (C++)
`S`	File variable
`V`	Local static variable
`b`	FORTRAN-based variables

OpenMP is equal to `N_SYM_OMP_TLS` (see `stab.h`) when an OpenMP variable has been declared `THREAD_PRIVATE`. It is zero otherwise.

Size is the size of the symbol in bytes.

In an ELF file, *Offset* is the number of bytes into the object file's data area. This is identified by a by a linker symbol table entry for the local symbol `Ddata.data`, which has the address of the first byte of the data area used by the object file. This symbol can be defined by the following assembly statements:

```
    .section ".data"
Ddata.data:
```

This creates a symbol table entry with the type `STT_NOTYPE`.

If the `N_STSYM` stab appears outside of a function, only file variables (*SymDesc* is `S`) are permitted. The address of the symbol is found from the linker stab with the same name. *Offset* is ignored.

In an ELF file, if the `N_STSYM` stab appears within a function, *Offset* contains the offset in bytes into the *data* area for the object file. The start address of the *data* area allocated by the object file is defined in the `STT_NOTYPE` entry for `Ddata.data`, which appears in the symbol table following the `STT_FILE` for this object file.

If the *Name* is globalized, as can happen when the object file has been built for the fix and continue feature of `dbx`, the *Name* will be found in the ELF symbol table and the *Offset* will be zero.

The only exception is for Fortran 95 pointer-based variables, where *Name* is the simple user name of the variable and *Offset* is the number of bytes from the start of the `.data` section.

C source file:

```
static int y = 1;
foo()
{
  static int x = 6;
}
```

generates the following stabs:

```
.stabs "y:S(0,3)",N_STSYM,0,4,y
.stabs "foo:F(0,3)",N_FUN,0,0,foo
.stabs "x:V(0,3)",N_STSYM,0,4,.L16-Ddata.data
```

The first `N_STSYM` (initialized static) stab indicates that `y` is a static global variable. It appears before the `N_FUN` (function) stab for `foo`. The last stab is the `N_STSYM` for the local static variable `x`.

N_TCOMM — Begin Task Common Block (0xe3)

```
.stabs "BlockName", N_TCOMM, ...unspecified
```

Note – This is currently an unsupported stab. It is only partially documented here. The primary reason for its inclusion is to reserve it for future use.

A N_TCOMM stab introduces a task common block and precedes the listing of symbols contained in the task common block. The task common block is named *BlockName*. Subsequent stabs up to an N_ECOMM stab specify the variables in the task common block.

A N_ECOMM stab terminates the listing of symbols within the named task common block.

Only N_GSYM stabs with V symbol type may appear between the N_TCOMM and N_ECOMM.

N_TCOMM and N_ECOMM may not be nested.

The following common declaration:

```
task common /blk/ a, b, c
```

would generate the following stabs:

```
.stabs  "blk_",N_TCOMM, ...
.stabs  "a:V6",N_GSYM,0,0,0
.stabs  "b:V6",N_GSYM,0,0,4
.stabs  "c:V6",N_GSYM,0,0,8
.stabs  "blk_",N_ECOMM,0,0,0
```

The first stab is the N_TCOMM, which starts the task common block. The next three stabs are N_GSYM (global symbol) stabs, which describe the three variables defined in the task common block. The last stab is the N_ECOMM, which ends the task common block.

N_TFLSYM — Thread Local Storage (TLS) Fragmented Data Symbol (0x2f)

```
.stabs " Name : SymDesc Type ", N_TFLSYM, 0, Size, 0
```

The N_TFLSYM stabs defines a global symbol with the name *Name*. The symbol represents a TLS static local variable that has become global to the linker because it has been placed in a separate ELF section by the compiler. The ELF symbol contains an offset used to compute the address of the symbol at runtime.

SymDesc is a symbol descriptor and may be one of the following:

G	Global weak (C++)
S	File
V	Local

If the N_TFLSYM stab appears outside of a function, only *SymDesc* S is permitted.

Size is the size of the variable in bytes.

The following C function, compiled with -xF=lcldata:

```
foo ()
{
__thread static int x;
}
```

generates the following stab for x:

```
 .stabs "$XBY9kkBSHBZ_CZV.foo.x:V(0,3)",N_TFLSYM,0x0,0x4,0x0
```

where $XBY9kkBSHBZ_CZV.foo.x is the globalized name for x.

N_TLCSYM — Thread Local Storage (TLS) Uninitialized Static Symbol (0x29)

```
.stabs " Name : SymDesc Type ", N_TLCSYM, 0, Size, Offset
```

An N_TLCSYM stab describes an uninitialized TLS static variable.

SymDesc is a symbol descriptor and may be one of the following:

G	Global weak (C++)
S	File
V	Local

If the N_TLCSYM stab appears outside of a function, only *SymDesc* S is permitted.

Size is the size of the variable in bytes.

In an ELF file, *Offset* is the number of bytes into the object file's uninitialized TLS data (tbss) area. This is identified by a linker symbol table entry for the local TLS symbol Ttbss.bss that has the address of the first byte of the tbss area used by the object file. This symbol can be defined by the following assembly statements:

```
.section ".tbss"
Ttbss.bss:
```

This creates a symbol table entry for Ttbss.bss with the type STT_TLS .

If the *Name* is globalized, as can happen when the object file has been built for the fix and continue feature of dbx, the *Name* can be found in the ELF symbol table and the *Offset* is zero.

The following C function:

```
foo ()
{
__thread static int x;
}
```

generates the following stab for x:

```
.stabs "$XBY9kkBGKBZ_yZV.foo.x:V(0,3)",N_TLCSYM,0x0,0x4,.
L15-Ttbss.bss
```

where $XBY9kkBGKBZ_yZV.foo.x is the globalized name for x.

N_TSTSYM — Thread Local Storage (TLS) Initialized Static Symbol (0x27)

```
.stabs " Name : SymDesc Type ", N_TSTSYM, 0, Size, Offset
```

An N_TSTSYM stab describes an initialized TLS static variable.

SymDesc is a symbol descriptor. It may be one of the following:

G	Global weak (C++)
S	File variable
V	Local static variable
b	Fortran-based variables

If the N_TSTSYM stab appears outside of a function, only file variables (*SymDesc* is S) are permitted. The address of the symbol is found from the linker stab with the same name. *Offset* is ignored.

Size is the size of the variable in bytes.

In an ELF file, *Offset* is the number of bytes into the object file's data area. This is identified by a linker symbol table entry for the local symbol Ddata.data, which has the address of the first byte of the data area used by the object file. This symbol can be defined by the following assembly statements:

```
.section ".tdata"
Ttdata.data:
```

This creates a symbol table entry with the type STT_TLS .

In an ELF file, if the N_TSTSYM stab appears within a function, *Offset*

contains the offset in bytes into the data area for the object file. The start address of the data area allocated by the object file is defined in the STT_TLS entry for Ttdata.data, which appears in the symbol table following the STT_FILE for this object file.

If the *Name* is globalized, as can happen when the object file has been built for the fix and continue feature of dbx, the *Name* is found in the ELF symbol table and the *Offset* is zero.

The C source file:

```
 foo()
{
__thread static int x = 6;
}
```

generates the following stab:

```
.stabs "$XBY9kkBXNBZ_ycV.foo.x:V(0,3)",N_TSTSYM,0x0,0x4,.L16-
Ttdata.data
```

where $XBY9kkBXNBZ_ycV.foo.x is the globalized name for x.

N_UNDF — Undefined (0x00)

N_UNDF is a linker stab that indicates that the symbol has undefined type. It is used to contain the name of the object file and occasional other purposes.

```
.stabs "Filename", N_UNDF, 0, NumStabs, BytesOfStringTable
```

Filename is a source file name, when one exists. *Filename* can be an object file name when the object file is compiler generated, such as in the C++ template repository. *Filename* should match the name supplied to the Elf LOCL FILE symbol or the assembly .file directive.

This stab is used in the Solaris Operating Environment to indicate that *BytesOfStringTable* size will be needed in the string for the following *NumStabs* stabs.

N_USING — C++ USING statement (0xc4)

The C++ USING statement has two different forms (called USING declarations and USING directives), depending on whether the argument is a single name or a namespace.

USING statements may be either position dependent or position independent. The C++ compiler will issue local (function local, or block local) USING stabs in a position dependent manner. In these cases, dbx needs to know the scope of the USING statement. Position independent USING stabs are used for those USING statements occurring inside other namespaces, class or global scope. Stab types are used to distinguish these two forms. For position independent USING statements, scope information is encoded in the stab itself.

USING Declaration

The USING declaration selects a particular name from a namespace and makes it known in the current scope. For example:

```
using N::sname;
```

where N is a previously defined namespace that contains a member named *sname*.

Local USING Declaration, Position Dependent

These USING declarations are found in function or local block scopes:

```
.stabs "P:<mangled_{N::sname}>",N_USING,0,0,0
```

They must be produced at the site of declaration to allow dbx to know the proper scope.

In this form of N_USING, the simple name being made known is spelled out (mangled). If the name refers to an overloaded function, one stab is issued per name.

Global, Namespace, or Class Scope `USING` Declaration, Position Independent:

```
.stabs "N:<mangled_{N::sname}>:<EnclTypeId>",N_USING,0,0,0
```

In this form of `N_USING`, the simple name being made known is spelled out (mangled). If the name refers to an overloaded function, one stab is issued per name.

EnclTypeId is the typeid of the enclosing scope (for `USING` nested in a namespace or class), if any. The field is left blank if the `USING` occurs in global scope (for which there is no *EnclTypeId*).

These stabs are produced by the compiler wherever it finds a need. Position is not an issue, and should not be taken as an indication of scope.

`USING` Directive

```
using namespace NAMESPACE;
```

The `USING` directive opens a previously declared namespace.

Again, stabs here are either position independent (global, namespace, or class scope), or position dependent (function, or local block scope).

Local `USING` Directive, Position Dependent

These `USING` directives are found in functions. The stabs for this form of `USING` look like this:

```
.stabs "Q:<NamespaceTypeId>", N_USING , 0, 0, 0
```

Global, Namespace, or Class Scope `USING` Directives, Position Independent

If a `USING` directive or declaration occurs inside a namespace, class, the `N_USING` stab will also contains the typeid of that namespace, or class. If it occurs in global scope, the field will be left blank.

```
.stabs "O:<NamespaceTypeId>:<EnclTypeId>",N_USING,0,0,0
```

Summary of `USING` statement stabs

Prefixes

	func local position-dependent	class/namespace/global position-independent
directives	Q	O
declarations	P	N

Other Fields

Position independent stabs for namespace or class scope all have an additional field for the typeid of the enclosing scope.

`N_XLINE` — Extended Line Number (`0x45`)

For line numbers greater than 65535, the `N_XLINE` stab is used to set a state variable in `dbx` that left-shifted 16 bits and bitwise `ORed` with all subseqeuent `N_SLINE` line numbers:

```
.stabn N_XLINE, 0, Hi16bitsLineMask, 0
```

CHAPTER 5

Symbol Descriptors

In the stab string that describes a symbol, the name of a variable or type is followed by a colon, a symbol descriptor, and a type specification. Some redundancy and interaction exist between the stab types and the symbol descriptors, so not all symbol descriptors can be used with each stab type.

The symbol descriptors describe what the symbol represents and may be one of the following.

empty	Local variable, no symbol descriptor has defaul
`A`	Automatic variable (Fortran 95)
`b`	Based variable
`c`	Constant symbol (Fortran 95)
`E`	External data
`F`	Global function or procedure
`f`	Local function or procedure
`G`	Global variable
`I`	Interface block
`J`	Internal procedure (Fortran 95)
`LT`	Lines in Template
`l`	Literal
`M`	Module (Fortran 95)
`P`	Prototype
`p`	Value parameter
`r`	Register variable
`S`	Static file variable

`T`	Enumeration, structure or union
`t`	Type name
`U`	Class declaration
`V`	Common or static local variable
`v`	Fortran variable parameter by reference
`X`	Function result variable
`x`	Value parameter by reference (aka, Array value parameter)
`Y`	C++ specification (C++ 4.0 and later)

These symbol descriptors are described in alphabetical order, specifying which stab types may contain them.

Local Variable (empty)

The absence of a symbol descriptor is used to describe a variable that is local to a function or procedure. It can appear only in an `N_LSYM` (local symbol) stab. The `desc` field of the stab specifies the length of the variable and the *value* field specifies its offset from the frame pointer.

For example, the local declaration within a C function:

```
int a;
```

generates the following stab:

```
.stabs "a:(0,3)",N_LSYM,0,4,-4
```

The type number for the symbol `a` is `(0,3)`, which was assigned to the type `int` in a preceding stab. The variable is four bytes long and is located four bytes before the function's frame pointer.

Automatic Variable (A)

The A symbol descriptor is used to describe a Fortran automatic variable. It appears in N_LSYM (function local) stabs. The value field of the stab contains the address of the variable (in an a.out file) or the offset within the statics generated for this compilation (in an ELF file). For further description of the value field, see “N_LSYM — Local Symbol (0x80)” on page 31.

The following Fortran program:

```
subroutine s(n)
   real a(n)
   print *,a
end
```

generates (in part) the following stab:

```
.stabs  "a:A(0,18)=ar(0,3);1;T-8;(0,5)”,N_LSYM,0x0,0x4,0xfffffff4
```

The stab describes that a is an automatic variable whose address is stored at the address specified in the value field.

Based Variable (b)

The b symbol descriptor is used to describe a Fortran based variable. It can appear in N_STSYM (initialized static), N_LCSYM (uninitialized static), N_PSYM (parameter), N_ROSYM (read-only initialized static), or N_LSYM (Fortran function local) stabs. The value field of the stab contains the address of the variable (in an a.out file) or the offset within the statics generated for this compilation (in an ELF file). For further description of the value field, see the relevant stab description in Chapter 1.

The following Fortran program

```
pointer (p,i)
p = loc(j)
j = 100
print *,i,j
end
```

generates (in part) the following stabs:

```
.stabs  "i:b3",N_LCSYM,0,0,VAR_SEG1+4
.stabs  "j:V3",N_LCSYM,0,0,VAR_SEG1+0
.stabs  "p:V3",N_LCSYM,0,0,VAR_SEG1+4
```

The first stab describes that `i` is a based variable whose address is stored at the address specified in the value field. The second stab describes a simple variable `j`. The third describes the pointer variable `p`, which has the same address as is specified for `i`.

Constant (c)

The `c` symbol descriptor is used to describe a constant symbol (for example, a Fortran `parameter`). It may appear only in an `N_LSYM` stab. The *Desc* and the *Value* fields of the stab are unused.

For example:

```
parameter (intwo=31)
```

could generate either of the following stabs:

```
.stabs "intwo:c3;1f",N_LSYM,0,0,0
.stabs "intwo:c40=3;1f;",N_LSYM,0,0,0
```

This describes `intwo` as a constant of type `int` (type number 3), with the value of the constant being hexidecimal `1f` (ie, 31). The value of the constant is separated from the type number by a ';' in the string. The value is a hexidecimal representation of the binary value.

External Data (E)

The `E` symbol descriptor is used to describe global variables referenced but not defined by an ELF file. It can appear only in an `N_GSYM` stab. There is no corresponding index stab for this type of `N_GSYM` stab. The purpose is to provide `dbx` with type information for symbols that may be defined in system libraries or other object files that were stripped or were not compiled with the `-g` option.

```
extern int var;
int example()
{
    return var;
}
```

The following stab is generated for `var`:

```
.stabs "var:E(0,3)",N_GSYM,0x0,0x0,0x0
```

Global Function or Procedure (F)

The `F` symbol descriptor is used to describe a global function or subroutine. In an ELF file, the address of the function is found in the symbol table.

The function:

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

generates the following stab for the function `main`:

```
.stabs "main:F(0,3)",N_FUN,0,0,_main
```

This describes `main` as a global function that returns an `int (type (0,3))`.

Local Function or Procedure (f)

The f symbol descriptor is used to describe a local function or subroutine. It can appear only in an N_FUN stab. In an ELF file, the address of the function is found in the symbol table.

The function:

```
static int foo ()
{
int a;
}
```

generates the following stab for the function foo:

```
.stabs "foo:f(0,3)",N_FUN,0,0,_foo
```

This describes foo as a local function that returns an int (type (0,3)).

Global Variable (G)

The G symbol descriptor is used to describe a global variable. This descriptor can appear only in an N_GSYM stab. The address of the variable is found in the symbol table. The *Desc* field of the stab contains the length of the variable.

The global declaration

```
int x;
```

generates the following stab:

```
.stabs "x:G(0,3)",N_GSYM,0,4,0
```

This describes a variable x, which is a global int (type (0,3)).

Interface Block (I)

The I symbol descriptor is used to describe a Fortran 95 interface block. It was formerly implemented as Generic Name (W). It can be used with the N_LSYM stab.

The *Value* and *Desc* fields of the stab are unused. For example, when a generic name cube_root is used for the specific names d_cube_root and s_cube_root, the following stab is generated:

```
.stabs "cube_root:I14;d_cube_root_ s_cube_root_ ;;;",
       N_LSYM,0x0,0x0,0x0
```

This describes cube_root as being a generic name for either d_cube_root_ or s_cube_root_. The string ";;;" must be provided at the end (it's reserved for future use). To determine which specific is intended, a comparison to the type/prototype information for each must be made.

Internal Procedure (J)

The J symbol descriptor is used to describe a Fortran 95 internal procedure. It can appear only in an N_FUN stab. In an a.out file, the value field of the stab is the entry point address of the internal procedure. In an ELF file, the address of the internal procedure is found in the symbol table.

For example:

```
subroutine s
integer x,y
call t
contains
subroutine t
integer x
end subroutine t
end
```

generates the following stab for subroutine `t`:

```
.stabs "s.t:J14",N_FUN,0,0,0
```

This describes `t` as a procedure that returns `void` which is internal to (tht is, nested in) the procedure `s`.

An additional `W` stab (*nested subprogram name*) should be generated in conjuction with this stab.

Lines in Template (LT)

Note – This stab is not used in Forte Developer 7, but is documented to support versions of Sun WorkShop in which it was used

In addition to the type information (see "Templates (YT, YI)" on page 99), each template definition causes an `LT` stab to be output. The `LT` descriptor may appear in a `N_LSYM` stab. This descriptor allows `dbx` to tell if you try to stop at a line in a function template, even if it's never been instantiated in this program. There are two kinds of `LT` stabs—`LTf` for function templates and `LTm` for member function templates.

The line-stabs for function templates look like:

```
Template_name:LTfStarting_line;;Ending_line", N_LSYM
```

`LTf` is replaced by `LTm` for member function templates.The empty field between the starting and ending line numbers is reserved for future use.

The template name is mangled.

Literal (l)

The `l` symbol descriptor is used to describe literals, such as true and false of `bool` type. Although it can be used in `N_LSYM` and `N_GSYM` stabs, so far only the use in `N_LSYM` stabs is identified and supported.

For example, in:

```
.stabs “true:l(0,3);1”,N_LSYM,0,0,0
```

`(0,3)` is the type number of `bool`.

Module (M)

The `M` symbol descriptor is used to describe a Fortran 95 `module`. It can appear only in an `N_FUN` stab. The *Value* field of the stab has no meaning with this descriptor. The return `type` of the module should always be equivalent to `void`.

In Sun WorkShop 6, the use of `N_MOD` replaces this symbol descriptor.

The module:

```
module m1
real p,q
end module m1
```

generates the following stab for module `m1`:

```
.stabs "m1:M14",N_FUN,0,0,0
```

This describes `m1` as a module (with return type of `void`).

Each the time a module is used, all relevant stabs for that module should be emitted.

Value Parameter (p)

The `p` descriptor specifies that the symbol is a parameter that is passed to a subroutine or function by value. It can appear in an `N_PSYM` (parameter) or `N_RSYM` (register symbol) stab. The *Desc* field contains the length of the variable and the `value` field contains its offset from the frame pointer.

The C function:

```
int func(int i) { return i; }
```

generates (in part) the following stabs:

```
 9:  .stabs "int:t(0,3)=bs4;0;32",N_LSYM,0x0,0x0,0x0
27:  .stabs "func:F(0,3);(0,3)",N_FUN,0x0,0x0,0x0
28:  .stabs "i:p(0,3)",N_PSYM,0x0,0x4,0x44
```

The first stab is an `N_FUN` (function) stab that defines `func` to return `type int (0,3)`, and has one parameter of type `int`. The second stab defines parameter `i` to be an `int` passed by value..

Prototype (P)

The `P` symbol descriptor specifies that the name is a function or procedure that appears elsewhere in the program. It can appear only in an `N_FUN` (function) stab.

There may or may not be an `N_FUN` stab where the function is actually defined.

The following C program:

```
#include <math.h>

int main ()
{
   float f;

   f = sin(.345);
}
```

generates the following stab for `sin`:

```
.stabs "sin:P(0,17);(0,17)",N_FUN,0,0,0
```

This `N_FUN` (function) stab indicates that *sin* is a function that takes a `double` (type `(0,17)`) as an argument and returns a `double` as a result.

Register Variable (r)

The r symbol descriptor specifies that the name is a register variable. It can appear in an N_RSYM stab. The symbol may either be a local variable or a parameter. If a parameter, the N_RSYM stab is immediately preceded by a stab that contains a p symbol type. The *Value* field of the stab specifies which register contains the variable.

The following function:

```
int foo (register float x)
{
register int y;
y = x;
return y;
}
```

generates (in part) the following stabs:

```
.stabs "foo:F(0,3);(0,16)",N_FUN,0,0,_foo
.stabs "x:p(0,16)",N_RSYM,0,4,63
.stabs "x:r(0,16)",N_RSYM,0,4,63
.stabs "y:r(0,3)",N_RSYM,0,4,29
```

The first N_FUN stab describes the function *foo*, which takes a float as an argument and returns an integer (types (0,16) and (0,3) respectively). The second and third N_RSYM stabs specify that x is a parameter and that it is assigned to register 63. The last stab N_RSYM indicates that y is an integer register variable that is assigned to register 29.

Static File Variable (S)

The S symbol descriptor specifies that the name is a static file variable. It can appear only in an N_LCSYM (uninitialized static), N_STSYM (initialized static), or N_ROSYM (read-only initialized static) stabs. The *Desc* field has the length of the variable. In an a.out file, the *Value* field has the address of the variable. In an ELF file, the *Value* field contains the offset from a file local symbol (see the description for N_LCSYM, N_STSYM, and N_ROSYM stabs).

The following program:

```
static int x;
static int y = 5;
static const int z = 10;

int main ()
{
}
```

generates (in part) the following stabs when generating an `a.out` file:

```
.stabs "y:S(0,3)",N_STSYM,0,4,_y
.stabs "z:S(0,3)",N_STSYM,0,4,_z
.stabs "x:S(0,3)",N_LCSYM,0,4,_x
```

The first two stabs describe *y* and *z* as initialized static variables (since they are described in `N_STSYM` stabs) with type integer (type `(0,3)`). The third `N_LCSYM` stab describes `x` as an uninitialized static variable. The *Value* field in each of these stabs contains the address of the variable.

In an ELF file, the stabs generated are as follows:

```
.stabs "y:S(0,3)",N_STSYM,0,4,Ddata.data-_y
.stabs "z:S(0,3)",N_ROSYM,0,4,Drodata.rodata-_z
.stabs "x:S(0,3)",N_LCSYM,0,4,Bbss.bss-_x
```

The second stab is an `N_ROSYM` stab, indicating that the variable is a read-only symbol. The *Value* fields in each contain the offset from a file local symbol that has the starting address of data allocated for this file in the appropriate section.

Enumeration, Structure or Union (T)

The `T` symbol descriptor specifies that the symbol is either an enumeration, structure, or union tag name. It can appear only in an `N_LSYM` (file or function local) stab. If there is no tag name (an anonymous enumeration, structure or union) then the name is omitted.

The following code:

```
enum color {red, blue, green} farbe;
struct S { int a, b; } SS;
union U { int a, b; } UU;

enum {small, avg, big} X;
struct { int j, k; } XS;
union { int j, k; } XU;
```

generates (in part) the following stabs:

```
.stabs "color:T(0,20)=ered:0,blue:1,green:2,;",N_LSYM,0,4,1
.stabs "S:T(0,21)=s8a:(0,3),0,32;b:(0,3),32,32;",N_LSYM,0,8,1
.stabs "U:T(0,22)=u4a:(0,3),0,32;b:(0,3),0,32;",N_LSYM,0,4,1
.stabs ":T(0,23)=esmall:0,avg:1,big:2,;",N_LSYM,0,4,1
.stabs ":T(0,24)=s8j:(0,3),0,32;k:(0,3),32,32;",N_LSYM,0,8,1
.stabs ":T(0,25)=u4j:(0,3),0,32;k:(0,3),0,32;",N_LSYM,0,4,1
```

Each of these stabs is an `N_LSYM` stab, which indicates that the symbol is either file local (if the stab appears outside of a function) or function local (if it appears within a function). The first three stabs give tag names for the symbol and are followed by type descriptions. The last three stab have the names omitted (the types are anonymous) and have different type numbers, although their descriptions have similar structure.

Type Name (`t`)

The `t` symbol descriptor specifies that the symbol is a new type name. It can be used in either `N_LSYM` or `N_GSYM` stabs.

Most compilers will generate a “canned” list of standard types. For example, the Sun WorkShop C compiler generates the following:

```
.stabs "char:t(0,1)=bsc1;0;8;",N_LSYM,0,0,0
.stabs "short:t(0,2)=bs2;0;16;",N_LSYM,0,0,0
.stabs "int:t(0,3)=bs4;0;32;",N_LSYM,0,0,0
.stabs "long:t(0,4)=bs4;0;32",N_LSYM,0,0,0
.stabs "long long:t(0,5)=bs8;0;64;",N_LSYM,0,0,0
.stabs "signed char:t(0,6)=bsc1;0;8;",N_LSYM,0,0,0
.stabs "signed short:t(0,7)=bs2;0;16;",N_LSYM,0,0,0
.stabs "signed int:t(0,8)=bs4;0;32;",N_LSYM,0,0,0
.stabs "signed long:t(0,9)=bs4;0;32;",N_LSYM,0,0,0
.stabs "signed long long:t(0,10)=bs8;0;64;",N_LSYM,0,0,0
.stabs "unsigned char:t(0,11)=buc1;0;8;",N_LSYM,0,0,0
.stabs "unsigned short:t(0,12)=bu2;0;16;",N_LSYM,0,0,0
.stabs "unsigned int:t(0,13)=bu4;0;32;",N_LSYM,0,0,0
.stabs "unsigned long:t(0,14)=bu4;0;32;",N_LSYM,0,0,0
.stabs "unsigned long long:t(0,15)=bu8;0;64;",N_LSYM,0,0,0
.stabs "float:t(0,16)=R1;4;",N_LSYM,0,0,0
.stabs "double:t(0,17)=R2;8;",N_LSYM,0,0,0
.stabs "long double:t(0,18)=R6;16;",N_LSYM,0,0,0
.stabs "void:t(0,19)=bs0;0;0",N_LSYM,0,0,0
```

This symbol descriptor is also used to describe type equivalences, such as the C `typedef`. The C statements:

```
enum color { red, green, blue };
typedef enum color colour;
```

Generates the following `N_LSYM` (local symbol) stab, which describes `colour` as a new type that is equivalent to the previous `enum color` type:

```
.stabs "colour:t(0,26)=(0,20)",N_LSYM,0,4,16
```

Class Declaration (U)

Currently types defined in a class are not entered into the right scope. In order to achieve this a stab declaring the class needs to be put out before the type stabs. This declaration stab can be applied to struct, class or union. The declaration stab is similiar to the definition stab but its stab string only contains the class name & type

id. Also when appropriate this declaration stab can be generated instead of generating forward reference stab. These declaration stabs will not be generated for all struct/class/union, they are generated only when necessary, such as when types are defined in a class.

Declaration Syntax

```
" Name : U (filenum, typenum)"
```

The `U` symbol descriptor is used for declaration and the `T` symbol descriptor is used for definition.

Example

```
class x {
  typedef int myint;

public:
  myint a;
};

main()
{
x xv;


}
```

Stabs

```
.stabs "__1nBx_:U(0,19)",N_ESYM,0x0,0x0,0x0
.stabs "nFmyint(0,19):t(0,20)=(0,3)",N_ISYM,0x0,0x0,0x0
.stabs
"__1nBx_:T(0,19)=Yc4x;;CcBa:(0,20),0,32;;;;;;;;;000;",N_ESYM,0x0
,0
```

Common or Static Local Variable (V)

The V symbol descriptor describes a static local variable. It can appear in N_STSYM (initialized static), N_LCSYM (uninitialized static), N_ALIAS (symbol alias), or N_ROSYM (read-only initialized static) stabs. The *Desc* field contains the length of the symbol. The *Value* field contains either the address of the variable (in an a.out file) or the offset from the start of the statics for this compilation (in an ELF file). When used to represent Fortran common blocks, this descriptor may be used in a N_GSYM (global symbol) stab enclosed by N_BCOMM and N_ECOMM stabs. In this usage, the *Desc* field is usually zero, and the *Value* field is the offset within the common block. See the respective stab descriptions for further details.

In 64-bit programs, it is possible for the offset within the common block to be larger than the *Value* field can represent. When this happens, two `N_GSYM` stabs, identical except for the *Value* fields, are generated; the first contains the upper 32 bits of the offset on the *Value* field, the second contains the lower 32 bits of the offset.

The following program:

```
int main ()
{
   static int x;
   static int y = 5;
}
```

generates (in part) the following stabs:

```
.stabs "x:V(0,3)",N_LCSYM,0,4,L15
.stabs "y:V(0,3)",N_STSYM,0,4,L16
```

The first stab is an `N_LCSYM` (uninitialized static) stab. The second is an `N_STSYM` (initialized static) stab. Since these examples were taken from an `a.out` file, the value field points to the actual location of the variable.

Variable Parameter by Reference (v)

The `v` symbol descriptor is used to describe a function or subroutine parameter that is passed by reference. It can be used only in an `N_PSYM` (parameter) stab. The *Value* field is the offset of the address of the parameter from the frame pointer.

The following Fortran program

```
function ifun (j)
return j * 2
end
```

generates the following stab for the parameter `j`:

```
.stabs "j:v3",N_PSYM,0,0,68
```

This N_PSYM stab describes the parameter j, which is passed by reference. The address of j is stored at offset 68 from the frame pointer.

Function Result Variable (X)

The X symbol descriptor describes the function result variable used by Fortran It can appear only in an N_PSYM (parameter) stab. The *Value* field of the stab contains the offset from the frame pointer where the return value is stored.

The following function:

```
function ifun (j)
ifun = j * j
return
end
```

generates the following stab for the result of ifun:

```
.stabs "ifun:X3",N_PSYM,0,0,0xfffffff0
```

This N_PSYM stab describes ifun as a result variable that is stored at -16 from the frame pointer.

C++ Specification (Y)

The Y symbol descriptors are the C++-specific symbol descriptors. In Sun C++ 5.0 (ANSI C++), there is a new ABI (Application Binary Interface) which has some effect on stabs. C++ 5.0 also has a compatibility mode in which it generates code (and stabs) similar to that of the previous release, C++ 4.0. Where stabs differ because of ABI, descriptions in this document will mention ABI1 (C++ 4.0) or ABI2 (C++ 5.0). Releases prior to Sun C++ 4.0, which was released with SPARCworks 3.0.x used a different encoding for C++ stabs, with a *Z* symbol descriptor.

The C++ system encodes (*mangles*) some type information about various externally visible names into those names. For example, it encodes global functions so that their parameter types are a part of the name as it appears in stabs. On systems that

use the ELF object format, this *mangled name* is the same name that is seen by the linker (the *linkername*). The mangled names you see in examples in this document are ABI1. The mangling for ABI2 is different.

In most cases, the user does not need to know about mangled names. For information on mangled names, see the `c++filt`(1) and `dem`(1) man pages. For information on differential mangling, a technique used to conserve string space, see "Differential Mangling" on page 165.

C++ reference types are similar to pointer types — their type numbers are indicated in a similar way, but using `&` instead of `*`. See "Reference (&)" on page 109.

C++ also generates some "hidden" functions, intended to be called only by the debugger, for cases where some information is trivially available at compile-time, but is much harder (for `dbx`) to find at runtime.

Functions with Default Arguments

`dbx` needs to have some indication when any of the parameters for a function have default values. For each function with default arguments, Sun Studio C++ creates a set of helper functions, one for each default parameter. If the default value is a simple integer literal, C++ uses the literal value instead of creating a helper function.

When the user tells `dbx` to call a function that has default arguments, and does not provide all of the args, `dbx` calls each of the helper functions or substitutes the known literal value, until it has values for all of the arguments that the user omitted. Then dbx calls the user's function itself.

The `N_FUN` stab for the function prototype (a ":P" stab) or for the function definition (a ":F" stab) (or both) indicates that this function has default arguments by following the type indicator with the name of the helper function or the signed decimal literal value for that parameter. For example:

```
int fo ( int a = 9, double b = 4.7 );
```

might yield a stab like this:

```
82:  .stabs
"__1cCfo6Fid_i_:F(0,3);(0,3)9;(0,15)_dflt_argA",N_FUN,0x0,0x0,0x
0
```

Inline Functions

When Sun WorkShop C++ is generating `dbx` information (compiling with the `-g` option), it chooses automatically to ignore the “inlineness” of all functions—it will not expand any function calls inline. Instead, it will compile into each translation unit a static copy of each inline function used in that translation unit. These static copies of inline functions are given `N_FUN` stabs marking them as static.

To avoid this extra code generation, the user can supply the `-g0` flag. When given `-g0`, the compiler does its normal inline expansion of function calls. In this case, it is possible that the compiler will not generate any stabs for the inline function.

Stabs for anonymous unions (`Ya`)

In general, an anonymous union behaves as if it were a collection of independent variables (the members of the union) which happen to have the same address. We give each member its own `Ya` stab. The C++ language requires that file-level anonymous unions must be static, so their stabs mark them as `N_LCSYMs`:

```
“MangledName:Ya Username(f,t)”, N_LCSYM, ...
```

For example:

```
static union {
       int x;
       float y;
};
```

yields three `N_LCSYM` stabs with the strings:

```
.stabs "___SA:Yax(0,3)",N_LCSYM,0x0,0x4,0x0
.stabs "___SA:Yay(0,14)",N_LCSYM,0x0,0x4,0x0
.stabs "___SA:Ya__BASE_TABLE__(0,19)=ar(0,3);0;-1;(0,21)",
N_LCSYM,0x0,0x0,0x0
```

Within functions, anonymous unions still get the letters `Ya` and have the same form as above. Their location within a function tells `dbx` that they are local to that function. An anonymous union that is declared static yields a `N_LCYSM` stab; a non-static anonymous union gets a `N_LSYM` stab.

Member anonymous unions

Within classes, anonymous unions are a little more detailed: since `dbx` must be able to print out ("`whatis -t`") the members of a class, there must be a way to indicate of *which* anonymous union each union-member is a member. A single-digit code is used (see *DataMembers* in "The Components of the Class Stab" on page 91).

Stabs for classes, structs, and non-anonymous unions

```
Y Type Size ClassName;
        Bases;
        DataMembers;
        MemberFunctions;
        StaticDataMembers;
        Friends;
        VirtualFunctionInfo;
        NestedClassList;
        AccessAdjustments;
        VirtualBaseClassOffsets;
        PassMethod;
        ;
```

Local classes (defined inside functions) simply show up in the relevant scope, and are `N_LSYM` stabs instead of `N_GSYM` stabs. They are otherwise identical to file-level classes.

The stabs for nested classes come out with the most deeply-nested ones first; the `Type` will be upper-case for nested classes.

The Components of the Class Stab

Each list of members and each list of base classes is a space-separated list; each member of each such list begins with a "*ppp code*" letter indicating what access this member or base has, whether it is virtual, and whether it is static. The encoding for these is ASCII '`@`' (`0x40`) plus 1,2,3 for private/protected/public, 4 for static, and 8 for virtual. (An added 16 is also used, to indicate class members which are anonymous unions.) Thus, a virtual public member has a code of `K`, and a non-virtual non-static private member would have an `A`.

Type is a one-letter field indicating what kind of structure this type describes. The possibilities are `c,s,u,a`, and `o` meaning class, struct, union, anonymous union, and ObjectiveC interface/class, respectively. If this class is nested inside another one, the type letter will be in upper case (`C,S,U,A,` or `O`). Note that `o` and `O` use the same stab structure as `c` and `C`, so any description in the following that applies to `c` and `C` can also be read as `o` and `O`.

Size is the total size in bytes of a "normal" instance of this class—a "leaf" or "most-derived" instance.

ClassName is mangled (in case of nested and local classes). For file-level (i.e., global) classes, the mangled-name is the same as the user-visible name.

Bases is the list of immediate base classes. Each entry begins with a ppp-code, telling whether it's virtual, and whether it's public, protected, or private. All the possibilities (base classes are never static):

`A`	Private non-static non-virtual
`B`	Protected non-static non-virtual
`C`	Public non-static non-virtual
`I`	Private non-static virtual
`J`	Protected non-static virtual
`K`	Public non-static virtual

Next, the entry includes the location (offset) of the embedded instance (if any) of this base class, in decimal bytes. If the inheritance is virtual, the entry contains the offset to the *pointer* which implements the virtual inheritance. This is followed by the base class type number (pair). There are no delimiters separating entries in the list—the closing parenthesis of the type number pair is sufficient. For example, a list like this:

```
;A10(1,5)K20(2,9)I30(3,4);
```

says that the base class with type number `(1,5)` is a non-virtual private base class at offset 10 within the current class; base class `(2,9)` is a virtual public base class whose pointer is at offset 20; and `(3,4)` is a virtual private base class with a pointer at offset 30.

DataMembers is the semicolon-separated list of non-static data members of the class. As with bases, each name is preceded by a ppp-code; these are never virtual nor static, so only A B or C occur. Following the member name is its type, offset, and size information, in the same format as C-language struct stabs (See "Structure or Record (s) and Union (u)" on page 106).

The mutable specifier on a class data member is indicated in the ppp-code of that member by adding a 0xc0. This changes the three possible codes of A B or C to M N or O.

Note – Each data member entry in this list ends with a semicolon, and the list itself also ends with a semicolon. So the list usually (when non-null) appears to have *two* terminating semicolons.

Members which are anonymous unions have their sub-members "elevated" to the level of the class containing the anonymous union. In addition, they get an extra (16) bit set in their ppp codes, and they get an extra digit, between the ppp-code and the member name. The digit indicates which anonymous union the member belongs to. (They can be re-used within a class; dbx needs them only to avoid merging adjacent anonymous unions.)

MemberFunctions lists the mangled names of member functions. These have ppp-codes, and can be virtual, static, or neither. So the first-character code will be `A`, `B`, `C`, `I`, `J`, or `K` (as above), or

`E`	Private static non-virtual
`F`	Protected static non-virtual
`G`	Public static non-virtual

For virtual functions, the ppp-code will be followed by an optional minus sign (indicating a pure virtual function), followed by a positive integer, the "virtual function index".

For explicit constructors add 0x08 to the ppp-code. Since a constructor has to be static and non-virtual, the possible ppp-codes for a constructor are `E`, `F`, or `G`, so explicit constructors become `M`, `N`, or `O`.

StaticDataMembers The mangled names of all of the static data members of this class are listed here. Their "ppp" codes can be `E`,`F`, or `G`.

Friends is a space-separated list of classes and functions to which this class grants access—the classes and functions declared to be the friends of the class being defined in this stab. In this list, each friend class name is preceded by an `@` character. The friend functions are listed without an `@`.

VirtualFunctionInfo is two numbers. The first is the number of the virtual function algorithm to be used. The second is the offset (in bytes) of the virtual table pointer in the class layout. The field is empty (i.e., just the terminating semicolon) if the class has no virtual functions.

Note – The algorithm number provides dbx with a small amount of insulation from changes in the compiler's virtual function calling algorithm. The compiler group supplies a library to the dbx group, and when dbx wants to call a virtual function, it calls into that library, supplying the virtual function algorithm number.

NestedClassList is a list of the type numbers for the classes nested within this class, each one preceded by the relevant ppp- code.

AccessAdjustments is a space-separated list of the access adjustment declarations. These consist of a ppp-code followed by the mangled name of the member whose access is being adjusted. The type and size information (if any) are deduced from the base class.

VirtualBaseClassOffsets lists where (by byte offset) each of this class' virtual base classes reside, when this is the "most derived" class. (This is in contrast to the *Bases* field, described above, which lists the offset to the virtual base's *pointer*.) This field allows dbx to be able to "downcast" a virtual base pointer or reference back to certain derived types. The form of this field is a list each of whose entries has the decimal offset followed by the type number for the virtual base. No separator is necessary within the list.

Example

This code example assumes some previously-declared classes; it declares a complicated but silly class called `green`:

```
class green : public blue, private black, public virtual bay {
    int x;
    virtual void purevirt() = 0;
    // three anonymous unions:
    union { float uf;  double ud; };
    union { char *ucp; void *uvp; };
    union { short stack; long  odds; };
protected:
    blue::moon;                    // an access adjustment
    static int z_static;
    public:
    blue::sky;
    int mf( );
    // a few friends:
    friend class blue;
    friend void frfn( );
    friend class  bay;
    friend int  sq( int );
};
```

Given type numbers of `(0,21)` for class `blue`, `(0,22)` for `black`, and `(0,23)` for `bay`, the above class `green` might generate a stab directive like this (broken apart for readability and commentary):

```
40:.stabs "__1nFgreen_:T(0,20)=Yc32green; // size 32, name green
  C4(0,21)A12(0,22)K2(0,23);// 3 base classes
  AcBx:(0,3),96,32;// a private int data member x
  Q1cCuf:(0,14),128,32;// uf in lst anon union
  Q1cCud:(0,15),128,64;// ud in lst anon union
  Q2cDucp:(0,26),192,32;// ucp in 2nd anon union
  Q2cDuvp:(0,27),192,32;// uvp in 2nd anon union
  Q3cFstack:(0,2),224,16;// stack in 3rd anon union
  Q3cEodds:(0,4),224,32;// odds in 3rd anon union
  ;        // second ';' ends data
  I-2cIpurevirt6M_v CcCmf6M_i;// 2 member functions
  FcIz_static;// 1 static data member
  __1cCsq6Fi_i_ @__1nDbay_ __1cEfrfn6F_v_  // a few friends
  @__1nEblue_;
  2 0;// virtual function algorithm #2 ("vtable"), and
    // virtual function table pointer offset 0 bytes.
  A(0,24)A(0,25)A(0,28);// 3 nested classes (the anon. unions)
  AcDsky(0,21) AcEmoon(0,21);// 2 access adjustments
  32(0,23);010;",// virt. base offset
  N_ESYM,0x0,0x20,0x0// the rest of the stab
```

Namespaces (Yn)

A `Yn` `N_LSYM` stab is generated for each namespace declaration. If the namespace has a mangled name, that should be used.

```
.stabs "NamespaceName:T(0,18)=Yn0username",N_LSYM,0x0,0x0,0x0
```

For these namespaces:

```
namespace N1 {
    int i1;
    void f1(char);
};
namespace N2 {
    int i2;
    void f2(char);
    namespaces N3 {
        int i3;
    };
};
```

The following stabs are generated:

```
.stabs "__1nCN1_:T(0,19)=Yn0N1;",N_ISYM,0x0,0x0,0x0
.stabs "__1nCN2_:T(0,20)=Yn0N2;",N_ISYM,0x0,0x0,0x0
.stabs "nCN3(0,20):T(0,21)=Yn0N3;",N_ISYM,0x0,0x0,0x0
```

This `N_ISYM` stab tells `dbx` the existence of the namespaces whose names are listed after `Yn`. When a name like `N1::i1` is encountered dbx can tell that `N1` is a namespace rather than a class. Each namespace's `N_ISYM` stab must appear before stabs of any of its members. In the example above, the `N_ISYM` stab for namespace `N1` must appear before the stab of i1 or f1. The stab for the name of a nested namespace must appear after the stab for the name of the namespace that contains it. The username in the `N_ISYM` stab is the unqualified username.

Pointers to class members (`YM, YD`)

Stabs for pointers to class data members and to member functions need not only the type information of what the member is pointing to, but also the type information of the class to which the member belongs. The YM and YD symbol types provide sufficient information for C++ 5.0 pointers.

New stab for pointer to class member function type:

`YM [K][B]` *ClassType ReturnType* `[`*ArgumentType*`]#`

`K` is used when the function is const. `B` is used when the function is volatile.

New stab for pointer to class data member type:

```
YD ClassType DatamemberType
```

Use the following program as an example:

```
int foo(int x) {return x;}
class A {
public:
     A(int arg) : d1(arg) {}
     int d1;
     int bar(int x) {return x+d1;};
};
int main()
{
A a(1);
A b(2);
int (A:: *x)(int) = &A::bar;
int (*y)(int) = &foo;
int A:: *z = &A::d1;
A *p = &a;
}
```

Old stabs for x, y and z :

```
.stabs “x:(0,25)=*(0,26)=f(0,3)”,N_LSYM,0x0,0x8,0xffffffec
.stabs “y:(0,27)=*(0,28)=f(0,3)”,N_LSYM,0x0,0x4,0xfffffe8
.stabs “z:(0,29)=*(0,22)”,N_LSYM,0x0,0x4,0xfffffe4
```

Stabs for x and y convey same thing—pointer to function that returns integer. The stab for z indicates pointer to void type, but has insufficient information for dbx.

New stabs for x, y, and z:

```
.stabs "__1fEmain1ABx_:(0,20)=YM(0,19)(0,3)(0,21)=*(0,19)(0,3)#",
N_LSYM,0x0,0x8,0xffffffec
.stabs "__1fEmain1ABy_:(0,22)=*(0,23)=g(0,3)(0,3)#",N_LSYM,0x0,
0x4,0xfffffe8
.stabs "__1fEmain1ABz_:(0,24)=YD(0,19)(0,3)",N_LSYM,0x0,0x4,
0xfffffe4
```

(0,19) is the type ID of class A.

Templates (YT, YI)

There are several kinds of template stabs[1]. The template source line stabs (LT stabs) are detailed under "Lines in Template (LT)" on page 78. The stabs describing the templates and their instantiations are described here. These are the YT stabs (T for Template) and YI stabs (I for Instantiation). There are the following kinds:

YTc or YTs	class or struct template
YTC or YTS	nested class or struct template
YTf	function template
YIc or YIs	template class or struct instance
YIC or YIS	nested template class or struct instance
YIf	template function (instantiation)
YIm	template member function
@< and @>	markers for "fake" index stabs

Since some templates are instantiated into the template repository (usually in the ./Templates.DB directory for C++ 4.0, and ./SunWS_cache directory for C++ 5.0), some of the template stabs will be generated in both the main object file and the template instantiation object file.

In ABI2, template definition names are mangled and template formal parameters are part of the mangled names. The resulting mangled names uniquely identify a template definition. In ABI1, template definition names were not mangled.

Most template stabs include a *TemplateParamList*. This is a list of the parameters in one of the following forms:

```
name:tYC type_number ;   // "normal" parameters
```

or

```
name:type_number ;       // non-type parameters
```

That is the parameter name is followed by a colon, an optional tYC, a type number, and a semicolon. dbx will define a new type number for each "tYC" type; the *type_number* given in these stabs should not be defined elsewhere in the file. For non-type parameters, the *type_number* must be defined previously.

1. Because of continuing changes in the C++ language definition, the organization of the template stabs is subject to possible change in some future release.

For example, given a template definition like:

```
template<class A, int x> ... // template definition...
```

the *TemplateParamList* looks like:

```
A:tYC(0,19);x:p(0,3)
```

where `(0,3)` means "int", and `(0,19)` is not defined elsewhere in the file. The `tYC` string indicates to dbx that this type is being defined here, so that it can be used within the template and within the stabs for its member functions. This `tYC` is also used for the main type defined in the `YIc` (instantiation) stabs.The *p* prevents dbx from interpreting `x` as the name of the type `(0,3)`.

Class templates (YTc, YTC, YTs, and YTS)

```
TemplateName:YT cCsS TypeNumber TemplateParamList;@;
    Size ClassName ;;;;;;;;;
```

The stabs for class templates are largely made up of placeholders.

The fields in the stab for a template class were designed to make its structure similar to those of the corresponding instantiation, and those of a non-template class. However, at the time these stabs are created, the compiler knows very little about the template—it has not even fully parsed its contents. Because of this, most of the entries have no meaning in the stab for the template itself.

TemplateName is the name of the template; it is not a mangled name.

`cCsS` indicates whether this is a class template (`c`), nested class template (`C`), struct template (`s`), or nested struct template (`S`).

TypeNumber is the (new) type number pair which this stab is defining to refer to this template class.

TemplateParamList is as described above.

Size is always zero (it is not yet known).

ClassName is the same as the `TemplateName`; it is not mangled

The several trailing semicolons correspond to the fields of the non-template class stab.

Example

The following code:

```
template<class T> class vector {
    T* v;
    int sz;
public:
    vector(int a) {sz = a;};
    T& operator[]( int );
    T& elem(int i) { return v[i]; }
    int size() { return sz; }
    int size(int) { return sz; }
    void dump( const char * );
};
```

yields:

```
.stabs "__1nGvector3CTA__:YTc(0,19);T:tYC(0,20);@;
0vector;;;;;;;;;",N_GSYM,0x0,0x0,0x0
```

Template Member Function Instantiations (`YIm`)

Each member function of a template gets a fake instantiation. This is actually a series of stabs generated to look somewhat like the set of stabs generated by a "real" function. That is, there is a `N_FUN` stab (with a `YIm` descriptor), `N_PSYM` stabs for each of the parameters for that member, and a pair of level one brackets (`N_LBRAC` and `N_RBRAC`).

The `YIm` stab string consists of:

```
MangledMemberFunctionName:YIm TemplateParamList;@;
          m;
          TemplateClassName:F ReturnType;ParamTypeList
```

For example, given the `vector` class example of the previous section, each of its member functions would get `YIm` and related stabs similar to this one (which is for the `vector<T>::elem` member):

```
42:  .stabs "__1cGvector4Ci_Eelem6Mi_ri_:YImT:(0,3);@;m;
     cEelem6Mi_r0(0,19):F(0,23);(0,24);(0,3)",N_FUN,0x0,0x0,0x0
43:  .stabs "this:p(0,24)",N_PSYM,0x0,0x4,0x44
44:  .stabs "i:p(0,3)",N_PSYM,0x0,0x4,0x48
```

Function templates (`YTf`)

A function template has a single `YTf` stab, perhaps preceded by a source-file (`N_SOL`) stab, and has an `LTf` stab, but no index stab. The `LTf` stabs are described in "Lines in Template (`LT`)" on page 78. The `YTf` stab is a `N_GSYM` stab, with the (unmangled) name of the template; it has three sections terminated by `@;` and then a pair of line numbers. The `YTf` stab's string is of this form:

TemplateName`:YTf` *TemplateParamList*`;@;`
`;`*TemplateName*`:T` *ReturnType*`;`*ParamTypesList*`;@;`
ParamNamesList`;@;`
StartingLineNum`;EndingLineNum`

TemplateName is the user's (unmangled) name of the template.

The *TemplateParamList* is described above.

The list following the `:T` is just like the list of *return-type-plus-arguments* that are in a normal `N_FUN` stab, although the parameters in this list might refer to the "dummy" types defined within the template's *TemplateParamList*.

The *ParamNamesList* is a list of entries like this:

ParameterName`:p` *type_number*`;`

The `:p` parts are similar to the strings that are contained in the `N_PSYM` stabs for normal functions.

Finally, the line numbers are in decimal—the beginning and the ending of the template source. These refer to lines in the source file most recently set by a `N_SOL` stab.

Stabs generation for function templates is "lazy", in the sense that the stabs for a given template are not output until unless that template is actually used.

Example

```
template<class A, class B> int tfex( B* x, A y ) { ... }
```

yields a function-template stab like this (split and annotated):

```
.stabs "__1cEtfex3CTACTB_6Fp10_i_:YTf
    A:tYC(0,19);B:tYC(0,20);  // define template params A and B
    @;;__1cEtfex3CTACTB_6Fp10_i_
    :T(0,3);(0,21)=*(0,20);(0,19); // returns int, gets (B*, A)
    @;
    x:p(0,21);y:p(0,19);// name function params x and y
    @;1;1;",// line number of template source.
    N_GSYM,0x0,0x0,0x0// rest of stab
```

Instantiations

Instantiation of a template for a given type can be overridden (“*specialized*”) by the user. For classes, the compiler can easily tell that this is happening, because the class name has angle brackets in it. For functions, there is no way for the compiler to tell that this is happening, because the corresponding function template might not be `#included` in the source file that contains the overriding function definition.

With respect to stabs, neither user-specialized functions nor user-specialized classes are treated as templates. For functions which are specializations of function templates, the compiler just creates a normal, non-template `N_FUN` stab. For classes, it creates a normal class stab.

Stabs for (Instantiated) template classes

Template classes that the compiler did instantiate from a template have `YIc` stabs of this form:

ClassName`:YI cCsS` *TypeNumber*`;@;`
ActualArguments`;@;g;`
Size TemplateName`;`
Bases`;`*DataMembers*`;`*MemberFunctions*`;`
StaticDataMembers`;`*Friends*`;`
VirtualFunctionInfo`;`
NestedClassList;AccessAdjustments`;`
VirtualBaseClassOffsets`;`
Passmethod`;`

ClassName is the mangled name of the instantiated class (for example, "stack<int>"). In C++ 4.0 (Stabs 3.1), this was a specially constructed mangled name of one of the member functions of the class. In C++ 5.0, however, there may not be any member functions in the instantiation.

`cCsS` tells whether the template was declared using `class` (`c`) or `struct` (`s`), and whether it was a nested class (`C`) or a nested struct (`S`).

TypeNumber is the (new) type number pair which this stab is defining to refer to this template class.

ActualArguments is a list of the actual arguments that were supplied in this instantiation. Normal `<class T>` parameters have the parameter name followed by a colon and then a type number not previously defined in this object file. Non-type parameters are represented either by a literal decimal number (for integral parameters), or as the name of a global variable (for parameters which are addresses). The types in the list are separated by semicolons.

Note – Non-type parameters are not currently implemented. They have the same form as normal template parameters.

Size is the actual size in bytes of an object of this class.

TemplateName names the template of which this class is an instantiation (or the template for which this class is an overriding specialization).

The rest of the stab for a template class looks a lot like the stab for a non-template class; since this is an instantiation (or a user-supplied override), we now know the sizes and offsets of everything.

Example

If we start with the `vector` template used in an above example, and instantiate it with a declaration like

```
  vector<double> vw(12);
```

then the compiler will produce these stabs:

```
.stabs "__1nGvector4Cd__:YIc(0,21);
@;T:(0,15);@;g;8__1nGvector3CTA__;;
AcBv:(0,22)=*(0,15),0,32;
AcCsz:(0,3),32,32;
;
Cc2t6Mi_v Cc2F6Mi_rd
CcEelem6Mi_rd CcEsize6M_i
CcEsize6Mi_i CcEdump6Mpkc_v;
;;;;;;010;",N_GSYM,0x0,0x0,0x0
```

Stabs for (instantiated) template functions

These are similar to part of the stab that is generated for the function template; the main difference is that the `:p` section is split off into separate `N_PSYM` stabs, as it is for normal `N_FUN` stabs. So the sequence of stabs that will be generated for template functions would look like this:

```
a “template index (@>) stab” for this generated function
        the “YIf” stab (type is N_FUN)
        some “:p” stabs (types are N_PSYM)
        <the guts of a normal function:  LBRACs,RBRACs
            SLINEs, local variable declarations, etc.>
```

The template index stab for a template function is an `N_FUN` stab. Its name starts with a special `@>` indicator, which is followed by the mangled name of the template function.

On systems using the `a.out` object format, these `@>` index stabs are put out as normal (non-index) stabs, and the mangled name has a colon at the end of it.

The `YIf` stab is an `N_FUN` stab, with a stab string like:

```
MangledFunctionName:YIf TemplateActualList;@;;
MangledTemplateName
:F ReturnType;FunctionArgList
```

The *TemplateActualList* is a list of entries each of which consists of a parameter name, a colon, and the type used in this instantiation.

Note – A function which is a specialization of a function template yields a normal, non-template `N_FUN` stab.

The *ReturnType* list is like the one for the template, except that the correct instantiation types have been substituted into it.

For example, given the template function:

```
template<class A, class B> int tfex( B* x, A y ) { ... }
```

And given an instantiation due to a call like this one:

```
int i = tfex( “str”, 9 );
```

The corresponding sequence of stabs for the instantiation might be (assuming that `char` is `(0,1)` and `int` is `(0,3)`):

```
28:  .stabs "__1cEtfex4CiCc_6FpTBTA_i_:YIfA:(0,3);B:(0,1);@;g;
__1cEtfex3CTACTB_6Fp10_i_:F(0,3);(0,22)=*(0,1);(0,3)",N_FUN,0x0,
0x0,0x0
  29:  .stabs "x:p(0,22)",N_PSYM,0x0,0x4,0x44
  30:  .stabs "y:p(0,3)",N_PSYM,0x0,0x4,0x48
  31:  .stabn N_LBRAC,0x0,0x1,0xc
  32:  .stabn N_SLINE,0x0,0x1,0xc
  33:  .stabn N_SLINE,0x0,0x1,0x18
  34:  .stabn N_RBRAC,0x0,0x1,0x18
```

Run Time Type Information (RTTI) (YR)

The only stabs used for exception handling are the RTTI (YR) stabs. One of these is generated for each type used in a `throw` expression. Its form is:

```
.stabs "RTTI_Symbol:YR TypeNumber",N_LSYM,0,0,0
```

The *RTTI_Symbol* names a compiler-generated variable which contains a `__RTTI` structure. That structure contains the type information for the thrown type. The *TypeNumber* refers to the thrown type itself. For example, given the following program:

```
struct type2throw { char *msg; } ;
main ( ) {
    int x = 12;
    type2throw var2throw;
    var2throw.msg = "test exception to throw";
    if( x < 20 ) throw x;
    else throw var2throw;
}
```

the compiler will generate the following stabs (among others):

```
.stabs "__1nKtype2throw_:T(0,19)=Ys4type2throw;;
     CcDmsg:(0,20)=*(0,1),0,32;;;;;;;;;000;",
     N_ESYM,0x0,0x4,0x0
.stabs "__RTTI__1Ci:YR(0,3)",N_LSYM,0x0,0x0,0x0
.stabs "__RTTI__1nKtype2throw_:YR(0,19)",N_LSYM,0x0,0x0,0x0
```

The mangled name `__RTTI__1Ci` indicates a builtin type (`int`), and the longer name `__RTTI__1nKtype2throw_` indicates the user-defined type in the second throw.

Note – No RTTI structure is valid until its initialization function has been called. This is normally done during the C++ static initialization phase.

Miscellaneous Stabs

Variadic (that is, *varargs* or *stdarg*) functions appear as if the ellipsis ("...") had been replaced with a parameter named `__builtin_va_alist`. The type for that argument will be a new built-in type name "...". For example, the final two entries in each object file's list of built-in types are:

```
.stabs "void:t(0,13)=bs0;0;0",N_ISYM,0x0,0x0,0x0
.stabs "...:t(0,17)=buv4;0;32",N_ISYM,0x0,0x0,0x0
```

The definition of a function declared as `void etc(const char * fmt, ...)` will cause stabs like these to be generated:

```
.stabs "__1cDetc6FpkcE_v_:F(0,13);// return type
    (0,19)=*(0,20)=k(0,1);// ptr to const char *
    (0,17)",// ellipsis "type"
    N_FUN,0x0,0x0,0x0
.stabs "fmt:p(0,19)",N_PSYM,0x0,0x4,0x44
.stabs "__builtin_va_alist:p(0,17)",N_PSYM,0x0,0x48,0x4
```

followed by the normal `LBRAC`/`RBRAC` stabs for the function definition.

CHAPTER 6

Type Specification

One major purpose of stabs is to describe the types of variables, parameters, or function return values. Stabs provide a very flexible method of describing types and defining variables and functions with these types. There are no predefined types; every type used in a program must be described by the compiler. Types are defined independently for each object file. There is no assumed relationship between the types defined in one object file with those defined in another. The stabs description also permits nameless or anonymous types.

A type described by stabs is a directed graph that ends in one of a few basic types: integer, floating point, or enumerated type. These are described in terms of their basic attributes (size, format, and so forth). All other types are described in terms of constructs built on these basic types or other defined types. A graphic representation of this directed graph is shown in FIGURE 6-1.

Each type, whether named or not, is given a unique type number. This type number is usually a pair of numbers within parentheses; the first number represents the sequence number of an include file and the second number represents the type defined within that file. The file sequence number corresponds to the number of `N_BINCL` or `N_EXCL` stabs in the object file. The original source file is given file number zero.

Languages that do not support a nested input file organization (such as Fortran) may use only a single number as the type number, or choose not to use the file numbers. This number represents the type within the source file. The single type number usage may appear in parentheses; For example, `(n)` is accepted as type `n`.

In the stab string, the name of a variable or type is followed by a colon, a symbol descriptor, and a type specification. This type specification may be a reference to a previously defined type number, a new definition of a type number, or a type description. This means that there are three variations in the format for the stab, which can be seen from the following C program fragment:

```
typedef int Integer;
    Integer var;
    Integer *pvar;
```

This generates (in part) the following stabs:

```
.stabs "int:t(0,3)=bs4;0;32",N_LSYM,0x0,0x0,0x0
.stabs "Integer:t(0,21)=(0,3)",N_LSYM,0x0,0x4,0x0
.stabs "pvar:G(0,22)=*(0,21)",N_GSYM,0x0,0x4,0x0
.stabs "var:G(0,21)",N_GSYM,0x0,0x4,0x0
```

The first stab is a description of the predefined type `int`. Symbol descriptor `t` indicates that `int` is a type that is given type number `(0,3)` and defined (as indicated by the equal sign followed by a type specification) as a basic integer.

The second stab describes `Integer`, also a type, which is given type number `(0,21)` and defined to be the same as `int (0,3)`.

The third stab describes `pvar`, a global variable (as indicated by the symbol descriptor `G`). Its type, `(0,22)`, is defined as a pointer (`*`) to type `Integer (0,21)`.

The fourth stab describes `var`, also a global variable, of type `Integer (0,21)`, which was defined in the second stab.

The symbol descriptors are described in Chapter 5.

The first character of the type description describes the type and is one of the following:

empty	Type reference
`a`	Array
`B`	Volatile
`b`	Basic integer
`D`	Dope vector (assumed shape array)
`d`	Dope vector (allocated array)
`e`	Enumeration

`F`	Function parameter
`f`	Function
`g`	Function with prototype info
`K`	Restricted
`k`	Const
`P`	Procedure parameter
`R`	Floating point
`r`	Range
`s`	Structure or record
`u`	Union
`x`	Forward reference
`Y`	C++ specification (C++ 4.0 and later, see page 61)
`z`	C99 variable length array
`*`	Pointer
`&`	Reference

These types are described below in alphabetical order.

A type is defined in terms of other types, which may in turn be defined in terms of more primitive types. Each type number must be unique within an object module, although the same numbers may appear in different object modules with completely different definitions. A new type may be defined wherever a type is referenced by following the new type number with an equal sign and the type definition.

There may be more than one type definition in a stab. This can be done by giving a type that is itself a type definition. For example, the global definition:

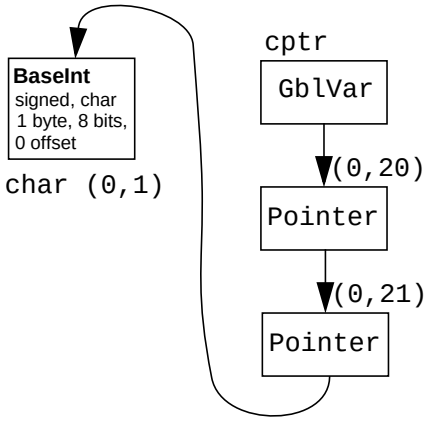
```
char **cptr;
```

generates the following stabs:

```
.stabs "char:t(0,1)=bsc1;0;8;",N_LSYM,0,0,0
.stabs "cptr:G(0,20)=*(0,21)=*(0,1)",N_GSYM,0,4,0
```

These stabs define `char` to be type number `(0,1)` and specify that it is a signed integer occupying one byte. `cptr` is described to be a global variable with type number `(0,20)`, which is unnamed. This is defined to be a pointer to type number `(0,21)`, which has no name. This, in turn, is defined to be a pointer to type `(0,1)`, which is `char`. This type tree is illustrated in FIGURE 6-1.

FIGURE 6-1 Example Type Tree.



Array (a)

```
a IndexType ; Type
```

The `a` type describes an array type by giving the type of the index(es) and elements. Multidimensional arrays are treated as if they were one-dimensional arrays of arrays.

IndexType is the type of the index value. It is usually a range type. *Type* is the type of the elements of the array.

The C statements:

```
char msg[5][10];
float array[15];
```

generate the following stabs:

```
.stabs "char:t(0,1)=bsc1;0;8;",N_LSYM,0,0,0
.stabs "int:t(0,3)=bs4;0;32;",N_LSYM,0,0,0
.stabs "float:t(0,16)=R1;4;",N_LSYM,0,0,0
.stabs "msg:G(0,21)=ar(0,3);0;4;(0,22)=ar(0,3);0;9;(0,1)",
       N_GSYM,0,50,0
.stabs "array:G(0,23)=ar(0,3);0;14;(0,16)",N_GSYM,0,60,0
```

In the fourth stab, `msg` is defined as a global variable (with a new type number `(0,21)` defined for it), which is an array. The first index is a range of 0 to 4 based on type `(0,3)`, which is defined as `int` in the second stab. The elements of this array are a new type definition (given number `(0,22)`), which is itself an array with an integer index with the range of 0 to 9. The elements of this array are each of type `(0,1)`, which are defined as `char` in the first stab.

The fifth stab declares `ar`, which is an array of fifteen elements (range zero to fourteen) each of which is a float defined as type `(0,16)`.

Normally the range supplied for an array type is respecified for each array.

Volatile (B)

`B` *Type*

Types which are `volatile` (or `const`, or `const volatile`) generate stabs which are similar to pointer stabs (see “Pointer (*)” on page 126). Use the type descriptor `B` for `volatile`. For example:

```
“volatile int x;  const volatile int y;”
```

might look like:

```
44:  .stabs “x:(0,29)=B(0,3)”,...
45:  .stabs “y:(0,30)=k(0,31)=B(0,3)”,...
```

Basic Integer (b)

```
b Sign [ Display ] Width ; Offset ; Nbits
```

The b type describes a binary integer. The value is an unsigned or twos-complement integer that is *Nbits* wide and stored in *Width* bytes. *Sign* specifies u for an unsigned value and s for a signed value. *Display* optionally specifies c, b, or v. This indicates the default mode in which the value is to be displayed: character, Boolean (true or false), or varargs ("...” or "__builtin_va_alist"). *Offset* specifies the starting bit of the value from the left-most bit in the storage allocated to the value.

A stab defining an `unsigned long` integer in C would look like the following:

```
.stabs  "unsigned long:t(0,14)=bu4;0;32;",N_LSYM,0,0,0
```

In most cases, *Nbits* will be eight times *Width* and *Offset* will be zero. In the case of a 16-bit value stored in the low-order half of a 32-bit word, the stab describing this is:

```
.stab "packed_val:t(0,21)=bs4;16;16",N_LSYM,0,0,0
```

A binary integer of zero length is the normal representation for `void` in C and comparable types in other languages:

```
.stab "void:t(0,19)=bs0;0;0",N_LSYM,0,0,0
```

Dope Vector (D)

```
D Type
```

The D type describes a Fortran 95 assumed shape array type, that is, an array that must be accessed via a dope vector, a pointer, or an anonymous reference to an array valued object. This specifier can be applied only to a *Type* that is an array or a pointer to an array. The value field of the stab represents the offset into the global dope vector table named dv_hdr .

For example, the statements:

```
subroutine sub(a)
integer a(:)

subroutine sub(a)
do i = 1, 100
   a(i) = i
enddo
end subroutine sub
```

generate the following stabs:

```
.stabs "VOID:t1=bs0;0;0",N_LSYM,0x0,0x0,0x0
.stabs "INTEGER*4:t4=bs4;0;32",N_LSYM,0x0,0x0,0x0
.stabs "sub_:F1;25=*26=D27=ar4;1;J0;4",N_FUN,0x0,0x2,0x0
.stabs "a:v26",N_PSYM,0x0,0x14,0xfffffffc
```

where the first two stabs define types 1 (`VOID`) and 4 (`INTEGER*4`). The third stab defines the subroutine `sub_` with return type 1, and a parameter of type 25 that is a pointer to type 26. Type 26 is an assumed shape dope vector (`D`) of type 27 that is an array whose range has bounds of type 4, a lower bound of 1, an unknown variable upper bound (`J0`), and whose elements are of type 4. The last stab defines `a` as a parameter passed by reference (*Symdesc* `v`). It has type 26, a size in bytes of `0x14`, and an offset from the stack frame pointer of `0xfffffffc`.

Dope Vector (d)

```
d Type
```

The `d` type describes a Fortran 95 allocatable array type, that is., an array which must be accessed via a *dope vector*, a pointer, or an anonymous reference to an array valued object. This specifier can be applied only to a *Type* which is an array or a pointer to an array. The `value` field of the stab represents the offset into the global dope vector table named `dv_hdr`.

For example, the statement:

```
real, allocatable :: x(:,:)
target y(10,10)
pointer corners (:,:)
corners => y(1:10:9,1:10:9)
```

generates the following stabs:

```
.stabs "x:42=dar3;1;1;ar3;1;1;6",N_LSYM,0,0,0,0x0
.stabs "corners:39=d*40=ar3;1;1;ar3;1;1;6",N_LSYM,0,0,
       0xfffffe40
```

The first stab describes `x` as an allocatable array (the `d` specifier) of type `real*4` (type number 6). It has two dimensions (the two `ar` specifiers), each having indices of type `integer` (type number 3) with range of 1 to 1. The second stab is similar except it defines a dope vector for a pointer to an array (the `d*` specifiers).

Note – For an allocatable array, the array dimensions specified in the stab represents the dimensions of the array declaration. The actual dimensions at any point during execution can be found in the dope vector for the array.

For a description of dope vectors, see Appendix E.

Enumeration (e)

```
e [ Type ] { Name : Number , }* ;
```

The `e` type describes an enumeration type by giving zero or more pairs of *Name* (the defined name) and *Number* (its value) separated by a colon. Each pair is followed by a comma. The last pair is followed by a semicolon.

Type is the base type that the enumeration is based on. If it is omitted, the enumeration is assumed to be based on 32-bit integers.

The statements:

```
enum { small, avg, big } size;
enum color { red, blue = 5, green };
```

generate the following stabs:

```
.stabs "size:G(0,13)=esmall:0,avg:1,big:2,;",N_GSYM,0,4,0
.stabs "color:T(0,14)=ered:0,blue:5,green:6,;",N_LSYM,0,4,0
```

The first stab declares `size` to be a global variable with an unnamed enumeration type, with the values specified. The *Desc* field of the stab gives the size of the variable in bytes. The second stab declares `color` to be an enumeration type and gives its values.

Function Parameter (F)

`F` *Type*

The `F` type describes a function passed as a parameter to a Fortran procedure or function. *Type* specifies the return value of the function.

The following Fortran subroutine:

```
subroutine func(pfunc)
        external pfunc
        return
        end
```

generates (in part) the following stabs:

```
.stabs "REAL*4:t6=R1;4",N_LSYM,0x0,0x0,0x0
.stabs "func_:F1;22=*21=f6",N_FUN,0x0,0x0,0x0
.stabs "pfunc:pF24=*23=f6",N_PSYM,0x0,0x4,0xfffffffc
```

The first stab describes a floating point type (type number `6`).

The second stab describes subroutine `func_`, which takes a single argument of type `22`, defined to be a pointer to type `21`, which is defined to be a function (`f`) returning type `6`.

The third stab desribes `pfunc`, a value parameter (`p`), which is a function (`F`) returning type `24`, which is defined to be a pointer to type `6`.

Function (`f`)

```
f Type
```

The `f` type describes a function value. *Type* is the return type of the function.

The C declaration:

```
int (*f) ();
```

generates the following stab:

```
.stabs "f:G(0,20)=*(0,21)=f(0,3)",N_GSYM,0,4,0
```

This stab describes global variable `f` (symbol descriptor `G`), which has type `(0,20)`. This is defined to be a pointer (type `*`) to type `(0,21)`, which in turn is defined to be a function (type `f`) returning an integer (type `(0,3)`).

Function With Prototype Info (`g`)

```
g Type [ArgumentType]#
```

The `g` type describes a function value with parameter prototype information. *Type* is the return type of the function. For each function parameter, there is an *ArgumentType* representing its type. The `g` type should be used in place of the `f` type when prototype information is available. The `f` type is still supported and may even be embedded in a `g` type stab (example below).

Simple example:

```
int (*fptr)(int,float);
```

produces stab:

```
.stabs “fptr:G(0,21)=*(0,22)=g(0,3)(0,3)(0,17)#”,N_GSYM,0,0,0
```

The second `(0,3)` is for first argument `int`, and `(0,17)` is for second argument `float`.

Another example:

```
typedef int (*func_type)(int (*) (int, float),float);
struct a {
int x;
my_type* (*fptr1)(int (*) (int, float),float);
char y;
int (*xptr)( int (*)(), int (*)(void), int);
// Note int(*)() is function with unknown parameter types
```

produces the following stabs:

```
.stabs “func_type:t(0,22)=*(0,23)=g(0,3)(0,50)=*(0,51)
       =g(0,3)(0,3)(0,17)#(0,17)#”,N_LSYM,0x0,0x4,0x40
.stabs “a:T(0,24)=s12x:(0,3),0,32;fptr1:(0,25)=*(0,26)=g(0,27)
       =*(0,3)(0,40)=*(0,41)=g(0,3)(0,3)(0,17)#(0,17)#,32,32;
       y:(0,1),64,8;;”,N_LSYM,0x0,0xc,0x1
.stabs “xptr:G(0,28)=*(0,29)=g(0,3)(0,60)=*(0,61)=f(0,3)(0,62)=
       *(0,63)=g(0,3)(0,20)#(0,3)#”,N_GSYM,0x0,0x4,0x0
```

Restricted (K)

```
K Type
```

The K descriptor describes a C “restricted pointer”. This may only appear as a modifier to a pointer *Type,* and directly supports the `_Restrict` keyword in C. For example:

```
typedef int *int_p;
_Restrict int_paa_ptr;
int * _Restrictbb_ptr;
```

would produce:

```
.stabs “int_p:t(0,21)=*(0,3)”,N_LSYM,0x0,0x4,0x40
.stabs “bbptr:G(0,22)=K(0,23)=*(0,3)”,N_GSYM,0x0,0x4,0x0
.stabs “aaptr:G(0,24)=K(0,21)”,N_GSYM,0x0,0x4,0x0
```

The second stab defines `bbptr` to be a global variable of type `(0,22)` which is unnamed. Type `(0,22)` is defined to be a type which is defined as a “restricted” type `(0,23)` which is also unnamed. Finally, type `(0,23)` is defined as pointer to a type `(0,3)` which is type named `int`. So you get a “restricted pointer to an integer”.

The third stab defines `aaptr` to be a global variable of type `(0,24)` which is unnamed. Type `(0,24)` is defined as a “restricted” type `(0,21)` which was defined by the first stab to be a pointer to an integer which was named `int_p`.

Const (k)

```
k Type
```

Types which are `const` (or `volatile`, or `const volatile`) generate stabs which are similar to pointer stabs (see “Pointer (*)” on page 126). Use the type descriptor `k` for `const`. For example:

```
const int x;  const volatile int y;
```

might look like:

```
.stabs “x:(0,29)=k(0,3)”,...
.stabs “y:(0,30)=k(0,31)=B(0,3)”,...
```

Floating Point (R)

```
R Format ; Nbytes
```

The `R` type describes a floating-point value. *Format* specifies the encoding used and may have the following values:

`NF_SINGLE`	`1`	IEEE 32-bit float value
`NF_DOUBLE`	`2`	IEEE 64-bit float value
`NF_COMPLEX`	`3`	Fortran complex (two 32-bit floats)
`NF_COMPLEX16`	`4`	Fortran double complex (two 64-bit doubles)
`NF_COMPLEX32`	`5`	Fortran quad complex (two 128-bit long doubles)
`NF_LDOUBLE`	`6`	Long double (one 128-bit float)
`NF_INTERARITH`	`7`	Interval (two 32-bit floats)
`NF_DINTERARITH`	`8`	Interval (two 64-bit doubles)

NF_QINTERARITH	9	Interval (two 128-bit long doubles)
NF_IMAGINARY	10	Imaginary (one 32-bit float)
NF_DIMAGINARY	11	Imaginary (one 64-bit double)
NF_QIMAGINARY	12	Imaginary (one 128-bit long double)

Nbytes is the number of bytes of storage the value occupies.

The standard C floating point type may be described by the following stabs:

```
.stabs "float:t(0,16)=R1;4;",N_LSYM,0,0,0
.stabs "double:t(0,17)=R2;8;",N_LSYM,0,0,0
.stabs "long double:t(0,18)=R6;16;",N_LSYM,0,0,0
```

Range (r)

```
r Type ; MinValue ; MaxValue
```

The r type describes a range of values based on *Type*.

MinValue and *MaxValue* are the smallest and largest values in the range. These values may be preceded by a code character. If the code character is an A or T, the number that follows is the run-time offset of the actual range value, either as an argument (A) or within the frame (T). If the code character is an S, the number that follows is the run-time offset of the size of the range (*MaxValue* - *MinValue* +1). A J indicates that the value is adjustable at run time and may not be determinable (as for Fortran array arguments). A J must be followed by a number (for example. J1). If no code character is specified, the range bounds are constant.

One example of a range can be found in "Array (a)" on page 112.

In the following C code:

```
int array[14];
```

generates the following stabs, where the range stab is `r(0,4);0;13`:

```
.stabs "int:t(0,3)=bs4;0;32",N_LSYM,0x0,0x0,0x0
.stabs "long:t(0,4)=bs4;0;32",N_LSYM,0x0,0x0,0x0
.stabs "array:G(0,21)=ar(0,4);0;13;(0,3)",N_GSYM,0x0,0x38,0x0
```

The first two stabs describe types `int (0,3)` and `long (0,4)`.

The third stab describes `array`, a global variable of type `(0,21)`, defined to be an array (`a`) whose range (`r`) has bounds of type `long (0,4)`, a lower bound (`MinValue`) of 0, and an upper bound (`MaxValue`) of 13. Each element of the array is of type `int (0,3)`.

Set (S)

```
S Type
```

The `S` type describes a set by giving its base type.

The Pascal statement:

```
var s: set of char;
```

generates the following stab:

```
.stabs  "s:GS(0,3)",N_GSYM,0x0,0x1,0
```

This defines `s` to be a global variable of type set of `char` (type `(0,3)`).

Structure or Record (s) and Union (u)

```
{ s | u } Size FieldList
```

This type describes a record or structure type, or a union type. A structure is identified with the type `s`, a union with the type `u`. *Size* is the number of bytes the object occupies in storage. *FieldList* is a list of one or more fields defined in the record or union. It has the following format:

```
FieldName : Type, BitOffset , BitSize ; }+
```

Each field description includes its name, *FieldName*, a *Type* description, the bit offset (*BitOffset*) within the record, and the size of the field in bits (*BitSize*).

The C declaration for a structure:

```
struct x {
  int a;
  float b;
};
```

generates the following stab:

```
.stabs "x:T(0,20)=s8a:(0,3),0,32;b:(0,16),32,32;;",N_LSYM,0,8,1
```

This defines `x` to be a type (number `(0,20)`), which is defined as a structure of eight bytes consisting of fields `a` (an `int` (type `(0,3)`) starting at bit 0, which is 32 bits long, and `b` (a `float` type `(0,16)`) starting at bit 32, which is 32 bits long.

Forward Reference (x)

```
x [ e | s | u | Type ] name
```

This type specifies that a type is declared but will be fully defined later in the stabs. It is an error to have a forward reference without an actual definition.

Name is the name of the enumeration, class, structure, or union, or other type that will be defined later in the stabs. If a *Type* is given, it must be a type pair.

The characters `e`, `s` and `u` represent enumeration, class/structure and union, respectively.

The following C code, which gives a forward reference for struct S:

```
struct S;

struct T {
   struct S *s;
};

struct S {
   struct T t;
};
```

generates the following stabs:

```
.stabs "T:T(0,20)=s4s:(0,21)=*(0,22)=xsS:,0,32;;",N_LSYM,0,4,1
.stabs "S:T(0,22)=s4t:(0,20),0,32;;",N_LSYM,0,4,0
```

The first stab describes `struct T`, which contains a pointer to a `struct S`. Because `S` has not been described, this pointer is given type `(0,22)`, which is described as a forward reference to a structure named `S`.

The second stab is the actual description of `struct S`, and gives the same type number as appeared in the forward reference.

C++ Types (Y)

The Y-stabs are used to represent various C++ types and symbols. For further description, see “C++ Specification (Y)” on page 88.

C99 Variable Length Array (z)

The C99 Variable Length Array type specifier (z) is used very much like an Array type specifier (a) only in situations where the array is a VLA.

For example, the declarations:

```
void foo()
{
int n=10;
char the_table[n];
the_table[2] = ’a’;
}
```

generate the following stab for the_table:

```
stabs  "the_table:(0,26)=zr(0,4);0;S-16;(0,1)",128,0,0,-20
```

The stab defines `the_table` to be of type z (a VLA), with an integer range whose *MinValue* is 0 and whose size (S) is given by the value stored at offset -16 from the current frame pointer. See “Range (r)” on page 122 for more information.

Pointer (*)

```
* Type
```

This type describes a pointer type by giving its base type.

The C statement:

```
char *p;
```

generates the following stab:

```
.stabs  "p:G(0,13)=*(0,2)",N_GSYM,0,1,0
```

This defines `p` to be a global variable (type `(0,13)`), which is defined to be a pointer to char (type `(0,2)`).

Reference (&)

```
& Type
```

In the stabs, reference types are "type makers" analogous to pointers. But where a pointer type gets `*(typeref)`, the corresponding reference type will have `&(typeref)`. A C++ program declaring some global variables

```
int *ptrv;   // a pointer
int &refv = x // (assume an int x exists)
```

might yield stabs like these:

```
30:  .stabs "ptrv:G(0,24)=*(0,3)",N_GSYM,0x0,0x4,0x0
32:  .stabs "refv:G(0,25)=&(0,3)",N_GSYM,0x0,0x4,0x0
```

CHAPTER 7

Auto-load Stab Processing

Introduction

Although it is possible to store all of the debugging stabs in the executable file this has several undesirable results:

- Large executable files
- Slow debugger start-up
- Longer link time

`dbx` was designed to load the executable file into memory and then read and process the debugging stabs to build its internal tables. This meant that the stabs for all of a program had to be processed, even if only a small part of the program would be looked at using the debugger.

One natural compensation is to compile only selected files with the `-g` option. This prevented the user from stepping through the entire program with the debugger. The user could also discover that the error was in a file that was compiled without the `-g` option, resulting in a time-consuming recompilation and relinking of the program, followed by restarting the debugger.

Auto-load stabs processing addresses these problems in several ways:

- `dbx` processes only the stabs necessary to display information at the time the information is needed.
- Debugging stabs are stored in the ELF object files, with only an index to the object files in the executable file or shared library.
- Debugging stabs may be stored in the executable file if no object file is saved or by specifying a flag to the compiler.

A somewhat different implementation of delayed stabs processing is used with `a.out` files to improve debugger performance. This is described in the section "Delayed Processing of `a.out` Files" on page 133.

Stabs Index

Each ELF object file created by a compiler must create a `.stab.index` section (and corresponding `.stab.indexstr` section for string values) that contains stab entries to support auto-load stab processing. This section must be created whether or not the compiler generates debugging stabs. (System libraries are shipped with these sections stripped away.)

The SPARCompiler assembler provides the `.xstabs` pseudo-operation, which directs a stab entry to a special section, such as `.stab.index:`

```
.xstabs "Section", "String", StabType, 0, Desc, Value
```

Section is the name of the section where the stab entry is to be stored. The other arguments are exactly the same as the `.stabs` directive. *String* will be stored in a section formed from the section name with `str` appended.

The assembler precedes the stabs explicitly generated by the compiler with a `N_UNDF` stab, which contains the number of characters in the corresponding string section in the `n_value` field.

The minimal `.stab.index` section for an object file that is compiled without debugging information looks like the following:

```
.stabs "heap.c",N_UNDF,0x0,0xc,0xee
.stabs "Xt ; V=3.0 ; R=3.0",N_OPT,0x0,0x0,0x29f30999
.stabs “/ex; /opt/bin/cc -o heap.o -c heap.c”,N_CMDLINE,0,0,0
```

The two entries the compiler must supply are `N_OPT` and `N_CMDLINE` stabs.

If debugging stabs are generated (if `-g` is specified) or if the file was compiled to permit function re-ordering (if `-xF` was specified) there are additional stab entries in the `.stab.index` section. The first two stab entries in the `.stab.index` section will then be `N_SO` stabs, the same as are generated for the debugging stabs. The next two stabs are then `N_OBJ` stabs with null strings in the string field. The linker will place the name of the link directory and object file name into these strings. There is an `N_FUN` stab for each global function and an `N_GSYM` for each global symbol defined in the file. The *Desc* and *Value* fields of these stabs are ignored and should be

set to zero. The N_OPT stab must also specify the -g option to indicate that debugging stabs were generated. The previous source file, compiled with debugging stabs, looks like this:

```
.stabs "heap.c",N_UNDF,0x0,0xc,0xee
.stabs "/ex",N_SO,0x0,0x0,0x0
.stabs "heap.c",N_SO,0x0,0x2,0x0
.stabs "",N_OBJ,0x0,0x0,0x0
.stabs "",N_OBJ,0x0,0x0,0x0
.stabs "Xt ; V=3.0 ; R=3.0; g",N_OPT,0x0,0x0,0x29f30999
.stabs “/ex; /opt/bin/cc -g -o heap.o -c heap.c”,N_CMDLINE,0,0,0
.stabs "alloc",N_FUN,0x0,0x0,0x0
.stabs "free",N_FUN,0x0,0x0,0x0
.stabs "alloc__6Heap_tFUi",N_FUN,0x0,0x0,0x0
.stabs "errcode",N_GSYM,0x0,0x0,0x0
```

When appropriate an N_MAIN stab will also appear in the .stab.index section.

Stabs in Object Files

The stabs entries generated by the .stabs directive are stored by the assembler in one of two sections in the object file. These are the .stab and .stab.excl sections. Strings are stored in the corresponding .stabstr and .stab.exclstr sections.

By default, the assembler stores stab entries in the .stab.excl section. When given the -s command option, it places stab entries in the .stab section. Every object file should have a .stab.index section and either a .stab or .stab.excl section, along with their corresponding string sections.

Compiler drivers should pass the -s command option to the assembler when it creates a temporary object file that will be deleted after the linker is invoked (for example, when the -c option is not specified), or when the compiler is given the -xs option, indicating that delayed stabs processing is to be suppressed for this compilation.

Stabs in Executable Files

The Solaris linker ignores the `.stab.excl` and `.stab.exclstr` sections and does not copy them to the executable or shared library. The `.stab` and `.stab.index` (and their corresponding `.stabstr` and `.stab.indexstr` sections) are processed like other sections: they are concatenated in the order in which the object files are processed.

When the linker encounters a pair of `N_OBJ` stabs in the `.stab.index` or `.stab` sections, it adds the directory and file name of the object file to the corresponding string section and update the `N_OBJ` stab entries to point to the strings. These are used by the debugger to locate the object file containing the stabs.

An executable or shared library file should contain a `.stab.index` section, which contains entries from all of the object files. It may also contain a `.stab` section, but it will never contain a `.stab.excl` section. If the stabs from an object file are present in the `.stab` section, the debugger will not look for any corresponding object file. If the `-g` option is not specified in the `N_OPT` stab, the debugger will not look for the object file, either.

An `N_MAIN` stab must be present in the `.stab.index` section for the main function in an executable file.

Debugger Operation

When the debugger reads an ELF executable file or shared library, it merges information from the linker symbol table and the stabs index. Next, all of the stabs that are contained in the executable or shared library are processed.

When the debugger requires information from the debugging stabs that have not been read, it will attempt to locate the object file containing that information and read the stabs from its `.stab.excl` section. The debugger uses the `use` path and the `N_OBJ` stabs to locate the object file.

Delayed Processing of `a.out` Files

Because the linker and debugger stabs for object files are merged, an `a.out` executable file contains sets of stabs. As a result, delayed stabs processing for an `a.out` file is somewhat different.

When the stabs are initially read by the debugger, only a small number are processed. The debugger identifies the place in the a.out file where each object file's stabs are located, and where each function or global variable is defined. When the debugger requires information that it has not read, it rereads the stabs from the a.out or shared library, and then processes only the required object files.

CHAPTER 8

Stabs Generation

Minimal Stabs Requirements

The compiler must generate a small number of debugging stabs even if the user has not specified the `-g` option.

The compiler must generate `N_SO`, `N_OBJ`, `N_MAIN`, and `N_OPT` in the `.stab.index` section.

The debugging stabs must include `N_SO` and `N_ENDM` stabs.

Stabs for Optimized Code

When the user requests both optimization and debugging, the compiler must generate many of the same debugging stabs. Stabs for functions and global variables are complete. An optimized function may often require more `N_SLINE` stabs because the optimizer moves instructions around. Stabs for local variables are not used, since the optmizer can often keep these values in registers for some or all of their lifetimes.

APPENDIX **A**

Stab Codes

TABLE A-1 Numerical Index

Stab Name	Stab Code	Described on Page	Description Model
N_UNDF	0x00	50	Start of object file .stabs *"Filename.o"*, N_UNDF, 0, *NumberOfStabs, BytesOfStringTable*
N_GSYM	0x20	26	Global symbol .stabs "*Name* : *SymDesc Type*", N_GSYM, 0, *Desc*, 0
N_OUTL	0x25	35	Outlined function .stabs "*Name*", N_OUTL, 0, 0, 0
N_FUN	0x24	24	Function or procedure .stabs "*Name* : *SymDesc RtnType* [; *ArgType*]*", N_FUN, 0, *RtnSize*, 0
N_STSYM	0x26	47	Initialized static symbol .stabs "*Name* : *SymDesc Type*", N_STSYM, 0, 0, *Offset*
N_TSTSYM	0x27	65	Initialized TLS static variable .stabs " *Name* : *SymDesc Type* ", N_TSTSYM, 0, *Size*, *Offset*
N_LCSYM	0x28	29	Unitialized static symbol .stabs "*Name* : *SymDesc Type*", N_LCSYM, 0, 0, *Offset*
N_TLCSYM	0x29	64	Uninitialized TLS static variable .stabs " *Name* : *SymDesc Type* ", N_TLCSYM, 0, *Size*, *Offset*
N_MAIN	0x2a	32	Name of main routine .stabs "*Name*", N_MAIN, 0, 0, 0

TABLE A-1 Numerical Index *(Continued)*

Stab Name	Stab Code	Described on Page	Description Model
N_ROSYM	0x2c	40	Read-only static symbol .stabs "Name : SymDesc Type", N_ROSYM, 0, 0, Offset
N_FLSYM	0x2e	35	Global symbol .stabs " *Name* : *SymDesc Type* ", N_FLSYM, OpenMP, *Size*, 0
N_TFLSYM	0x2f	63	Global symbol .stabs " *Name* : *SymDesc Type* ", N_TFLSYM, 0, *Size*, 0
N_CMDLINE	0x34	17	.stabs "cd *path*;cc..etc", 0, 0, 0
N_OBJ	0x38	33	Object file or path name .stabs "*ObjectDir*", N_OBJ, 0, 0, 0 .stabs "*ObjectFile*", N_OBJ, 0, 0, 0
N_OPT	0x3c	34	Compiler options .stabs "*Options*", N_OPT, 0, 0, *TimeStamp*
N_RSYM	0x40	41	Register symbol .stabs "*Name* : *SymDesc Type*", N_RSYM, 0, *Size*, *Number*
N_SLINE	0x44	42	Source line .stabn N_SLINE, 0, *Line*, *Offset*
N_XLINE	0x45	53	Source line .stabn N_XLINE, 0, *Hi16bitsLineMask*, 0
N_BROWS	0x48	17	Path to associated .cb file .stabs "*bdfile*", N_BROWS, 0, 0, 0 (Not used by dbx)
N_ILDPAD	0x4c	27	Pad string table (used by linker, not dbx) .stabs "*Objname*", N_ILDPAD,0,0, *Len* (Formerly N_FLINE)
N_ENDM	0x62	21	End module .stabn N_ENDM, 0, 0, 0
N_SO	0x64	45	Compilation source file or path name .stabs "*SourceDir*", N_SO, 0, 0, 0 .stabs "*SourceFile*", N_SO, 0, *LangCode*, 0
N_MOD	0x66	32	Fortran 95 module begin .stabs "*Name*: *MemberList*", N_MOD, 0, 0, 0

TABLE A-1 Numerical Index *(Continued)*

Stab Name	Stab Code	Described on Page	Description Model
N_EMOD	0x68	21	Fortran 95 module end .stabs "*Name*", N_EMOD, 0, 0, 0
N_READ_MOD	0x6a	39	Fortran 95 use statement .stabs "*Name* [:] [only;] [*NameList*]
N_ALIAS	0x6c	13	.stabs "*FNewname*:*Oldname*", N_ALIAS, 0, 0, 0 .stabs "*VNewname*:*Oldname*", N_ALIAS, 0, 0, 0 .stabs "*externalname*:*Rsourcename*", N_ALIAS, 0, 0, 0 .stabs "*newname*:S<*typeidof_newname*>=<*typeidof_oldname*>", N_ALIAS, 0, 0, 0
N_LSYM	0x80	31	Local symbol .stabs "*Name* : *SymDesc Type*", N_LSYM, 0, *Desc*, *Value*
N_BINCL	0x82	16	Begin include file .stabs "*FilePath*", N_BINCL, 0, 0, *HashValue*
N_SOL	0x84	46	Included or referenced source file .stab "*FilePath*", N_SOL, 0, 0, 0
N_PSYM	0xa0	38	Formal parameter .stabs "*Name* : *SymDesc Type*", N_PSYM, 0, 0, *Offset*
N_EINCL	0xa2	21	End of include file .stabn N_EINCL, 0, 0, 0
N_ENTRY	0xa4	22	Fortram alternate entry .stabs "*Name* : e *RtnType* ; *FunName* ; ;", N_ENTRY, 0, Line, 0
N_LBRAC	0xc0	28	Start of scope (left bracket) .stabn N_LBRAC, 0, *Level*, *Offset*
N_USING	0xc4	50	C++ USING statement .stabs "P:<*N*::*m*>", N_USING, 0, 0, 0 .stabs "N:<*N*::*m*>:<*EnclTypeId*>", N_USING, 0, 0 ,0 .stabs "Q:<*NamespaceTypeId*>", N_USING, 0, 0 0 .stabs "O:<*NamespaceTypeId*>:<*EnclTypeId*>", N_USING, 0, 0, 0

TABLE A-1 Numerical Index *(Continued)*

Stab Name	Stab Code	Described on Page	Description Model
N_ISYM	0xc6	27	Position independent type, internal .stabs "*Name*:*SymDesc Type*", N_ISYM, 0, *Desc*, *Value*
N_ESYM	0xc8	23	Position independent type, external .stabs "*Name*:*SymDesc Type*", N_ESYM, 0, *Desc*, *Value*
N_PATCH	0xd0	35	.stabn N_PATCH, 0, 0, *Addr*
N_CONSTRUCT	0xd2	19	Constructor description .stabs "*Var*:*State*",N_CONSTRUCT, 0,*End-Start*,*Start-Func*
N_DESTRUCT	0xd4	20	Destructor description .stabs "*FromState*:*ToState*",N_DESTRUCT, 0,*End-Start*,*Start-Func*
N_CODETAG	0xd8	18	Code generation detail .stabn N_CODETAG, *marker*, 0, *addr*
N_FUN_CHILD	0xd9	38	Created when a nesting relationship between functions needs to be communicated to the debugger .stabs "*Name*", N_FUN_CHILD, 0, 0, 0
N_RBRAC	0xe0	39	End of scope (right bracket) .stabn N_RBRAC, 0, *Level*, *Offset*
N_BCOMM	0xe2	15	Begin common block .stabs "*BlockName*", N_BCOMM, 0, 0, *HashValue*
N_TCOMM	0xe3	49	Begin task common block .stabs "*BlockName*", N_TCOMM, 0, 0, *Offset*
N_ECOMM	0xe4	20	End common block .stabs "*BlockName*", N_ECOMM, 0,0,0

TABLE A-2 Alphabetical Index

Stab Name	Stab Code	Described on Page	Description Model
N_ALIAS	0x6c	13	.stabs "*FNewname*:*Oldname*", N_ALIAS, 0, 0, 0 .stabs "*VNewname*:*Oldname*", N_ALIAS, 0, 0, 0 .stabs "*externalname*: *Rsourcename*", N_ALIAS, 0, 0, 0 .stabs "*newname*:S<*typeidof_newname*>= <*typeidof_oldname*>", N_ALIAS, 0, 0, 0
N_BCOMM	0xe2	15	Begin common block .stabs "*BlockName*", N_BCOMM, 0, 0, *HashValue*
N_BINCL	0x82	16	Begin include file .stabs "*FilePath*", N_BINCL, 0, 0, *HashValue*
N_BROWS	0x48	17	Path to associated .cb file .stabs "*bdfile*", N_BROWS, 0, 0, 0
N_CMDLINE	0x34	17	.stabs "cd *path*;*cc*..etc", 0, 0, 0
N_CODETAG	0xd8	18	Code generation detail .stabn N_CODETAG, *marker*, 0, *addr*
N_CONSTRUCT	0xd2	19	Constructor description .stabs "*Var*:*State*",N_CONSTRUCT, 0,*End-Start*,*Start-Func*
N_CPROF	0xf0	19	Cache profile feedback (reserved for future use)
N_DESTRUCT	0xd4	20	Destructor description .stabs "*FromState*:*ToState*",N_DESTRUCT, 0,*End-Start*,*Start-Func*
N_ECOMM	0xe4	20	End common block .stabs "*BlockName*", N_ECOMM, 0,0,0
N_EINCL	0xa2	21	End of include file .stabn N_EINCL, 0, 0, 0
N_EMOD	0x68	21	Fortran 95 module end .stabs "*Name*", N_EMOD, 0, 0, 0
N_ENDM	0x62	21	End module .stabn N_ENDM, 0, 0, 0
N_ENTRY	0xa4	21	Fortran alternate entry .stabs "*Name* : e *RtnType* ; *FunName* ; ;", N_ENTRY, 0, *Line*, 0

TABLE A-2 Alphabetical Index *(Continued)*

Stab Name	Stab Code	Described on Page	Description Model
N_ESYM	0xc8	23	Position independent type, external .stabs "*Name*:*SymDesc Type*", N_ESYM, 0, *Desc*, *Value*
N_FLSYM	0x2e	35	Global symbol .stabs " *Name* : *SymDesc Type* ", N_FLSYM, OpenMP, *Size*, 0
N_FUN	0x24	24	Function or procedure .stabs "*Name* : *SymDesc RtnType* [; *ArgType*]*", N_FUN, 0, *RtnSize*, 0
N_FUN_CHILD	0xd9	38	Created when a nesting relationship between functions needs to be communicated to the debugger .stabs "*Name*", N_FUN_CHILD, 0, 0, 0
N_GSYM	0x20	26	Global symbol .stabs "*Name* : *SymDesc Type*", N_GSYM, 0, *Desc*, *Value*
N_ILDPAD	0x4c	27	Pad string table (used by linker, not dbx) .stabs "*Objname*", N_ILDPAD,0,0, *Len* (Formerly N_FLINE)
N_ISYM	0xc6	27	Position independent type, internal .stabs "*Name*:*SymDesc Type*", N_ISYM, 0, *Desc*, *Value*
N_LBRAC	0xc0	28	Start of scope (left bracket) .stabn N_LBRAC, 0, *Level*, *Offset*
N_LCSYM	0x28	29	Unitialized static symbol .stabs "*Name* : *SymDesc Type*", N_LCSYM, 0, 0, *Offset*
N_LSYM	0x80	31	Local symbol .stabs "*Name* : *SymDesc Type*", N_LSYM, 0, *Desc*, *Value*
N_MAIN	0x2a	32	Name of main routine .stabs "*Name*", N_MAIN, 0, 0, 0
N_MOD	0x66	32	Fortran 95 module begin .stabs "*Name*: *MemberList*", N_MOD, 0, 0, 0
N_OBJ	0x38	33	Object file or path name .stabs "*ObjectDir*", N_OBJ, 0, 0, 0 .stabs "*ObjectFile*", N_OBJ, 0, 0, 0

TABLE A-2 Alphabetical Index *(Continued)*

Stab Name	Stab Code	Described on Page	Description Model
N_OPT	0x3c	34	Compiler options .stabs "*Options*", N_OPT, 0, 0, *TimeStamp*
N_OUTL	0x25	35	Outlined function .stabs "*Name*", N_OUTL, 0, 0, 0
N_PATCH	0xd0	35	.stabn N_PATCH, 0, 0, *Addr*
N_PSYM	0xa0	38	Formal parameter .stabs "*Name* : *SymDesc Type*", N_PSYM, 0, 0, *Offset*
N_RBRAC	0xe0	39	End of scope (right bracket) .stabn N_RBRAC, 0, *Level*, *Offset*
N_READ_MOD	0x6a	39	Fortran 95 use statement .stabs "*Name* [:] [only;] [*NameList*]
N_ROSYM	0x2c	40	Read-only static symbol .stabs "*Name* : *SymDesc Type*", N_ROSYM, 0, 0, *Offset*
N_RSYM	0x40	41	Register symbol .stabs "*Name* : *SymDesc Type*", N_RSYM, 0, *Size*, *Number*
N_SLINE	0x44	42	Source line .stabn N_SLINE, 0, *Line*, *Offset*
N_SO	0x64	45	Compilation source file or path name .stabs "*SourceDir*", N_SO, 0, 0, 0 .stabs "*SourceFile*", N_SO, 0, *LangCode*, 0
N_SOL	0x84	46	Included or referenced source file .stab "*FilePath*", N_SOL, 0, 0, 0
N_STSYM	0x26	47	Initialized static symbol .stabs "*Name* : *SymDesc Type*", N_STSYM, 0, 0, *Offset*
N_TCOMM	0xe3	49	Begin task common block .stabs "*BlockName*", N_TCOMM, 0, 0, *Offset*
N_TFLSYM	0x2f	63	Global symbol .stabs " *Name* : *SymDesc Type* ", N_TFLSYM, 0, *Size*, 0

TABLE A-2 Alphabetical Index *(Continued)*

Stab Name	Stab Code	Described on Page	Description Model
N_TLCSYM	0x29	64	Uninitialized TLS static variable .stabs " *Name* : *SymDesc Type* ", N_TLCSYM, 0, *Size*, *Offset*
N_TSTSYM	0x27	65	Initialized TLS static variable .stabs " *Name* : *SymDesc Type* ", N_TSTSYM, 0, *Size*, *Offset*
N_UNDF	0x00	50	Undefined symbol
N_USING	0xc4	50	C++ USING statement .stabs "P:<*N*::*m*>", N_USING, 0, 0, 0 .stabs "N:<*N*::*m*>:<*EnclTypeId*>", N_USING, 0, 0 ,0 .stabs "Q:<*NamespaceTypeId*>", N_USING, 0, 0 0 .stabs "O:<*NamespaceTypeId*>: <*EnclTypeId*>", N_USING, 0, 0, 0
N_XLINE	0x45	64	Source line .stabn N_XLINE, 0, *Hi16bitsLineMask*, 0

APPENDIX **B**

Symbol Descriptors

TABLE B-1 Symbol descriptors — Alphabetical by character code

Code	Description
empty	Local variable, page 56
`A`	Automatic variable (Fortran 90), page 57
`b`	Based variable, page 57
`c`	Constant symbol (Fortran), page 58
`E`	External data, page 59
`F`	Global function or procedure, page 59
`f`	Local function or procedure, page 60
`G`	Global variable, page 60
`I`	Interface block, page 61
`J`	Internal procedure (Fortran 90), page 61
`LT`	C++ 4.0 Lines in template, page 62
`l`	Literal, page 62
`M`	Module (Fortran 90), page 63
`P`	Prototype, page 64
`p`	Value parameter, page 63
`r`	Register variable, page 65
`S`	Static file variable, page 65
`T`	Enumeration, structure or union, page 66
`t`	Type name, page 67

TABLE B-1 Symbol descriptors — Alphabetical by character code *(Continued)*

Code	Description
U	Class declaration, page 68
V	Common or static local variable, page 68
v	Variable parameter by reference, page 71
X	Function result variable, page 72
Y	C++ 4.0 specification (see below), page 72

TABLE B-2 C++ 5.0 Specification

Code characters	Description
YA	Anonymous union, nested, page 74
Ya	Anonymous union, page 74
YC	Class nested, page 75
Yc	Class, page 75
YD	Pointer to class data member, page 81
YI	Template Instantiation, page 83
YM	Pointer to class member function, page 81
YR	Run Time Type Information (RTTI), page 91
YS	Structure nested, page 75
Ys	Structure, page 75
YT	Template definition, page 83
YU	Union, nested, page 75
Yu	Union, page 74

APPENDIX **C**

Type Codes

TABLE C-1 Type codes — Alphabetical by character code

Code character	Description
empty	Type reference, page 96
a	Array, page 96
B	Volatile, page 97
b	Basic integer, page 98
D	Dope vector, page 114
d	Dope vector, page 98
e	Enumeration, page 99
F	Function parameter, page 100
f	Function, page 101
g	Function with prototype info, page 101
K	Restricted, page 103
k	Const, page 104
R	Floating point, page 104
r	Range, page 105
s	Structure or record, page 106
u	Union, page 106
x	Forward reference, page 107

TABLE C-1 Type codes — Alphabetical by character code *(Continued)*

Code character	Description
Y	C++ specification, page 108
*	Pointer, page 109
&	Reference, page 109

APPENDIX **D**

Index Stabs

Index stabs are used to support auto-load stab processing, where stabs are loaded on demand from the object files (not an executable file or library), which must be kept available for the debugger to use. The debugger knows which object file to open to find the stabs for any global symbol by using the index stabs that are always copied to the executable file.

The index stabs are similar to the regular stabs in form and in the ordering that imparts file scoping. For every regular stab that describes a file or a global symbol, there is a corresponding index stab. The index stab is often simplified, with less information than the regular stab. Not all regular stabs have corresponding index stabs.

The stabs that can be index stabs are:

```
N_SO

N_OBJ

N_OPT

N_CMDLINE

N_MAIN

N_FUN

N_MOD

N_GSYM

N_ESYM

N_ILDPAD

N_PATCH

N_UNDF
```

N_SOL

N_CODETAG

In general, every such regular stab has a corresponding index stab. One exception is the extern data Symdesc E, which is an N_GSYM stab but has no corresponding index stabs. Another is the N_CMDLINE stab , which is only an index stab; it has no corresponding regular stab. Some regular stabs like N_LCSYM, N_STSYM, and N_FLSYM can have globalized names in their strings, and this globablization results in a corresponding N_GSYM index stab. There is no correlation at all between N_SOL index stabs and regular N_SOL stabs. N_SOL index stabs are used solely to identify header files with executable code in them, while regular N_SOL stabs also indicate the definitions of types and variables within headers files.

Simplified index stabs usually contain nothing more than the name of the symbol. This applies to the stabs N_FUN, N_MOD, N_GSYM, N_ESYM, and N_PATCH. All other index stabs (N_SO, N_OBJ, N_OPT, N_MAIN, N_ILDPAD, and N_UNDF) are identical to their corresponding regular stabs. One exception is COMDAT index stabs, which always have n_other == 1. N_OBJ index stab strings are filled in at link time, while the regular N_OBJ stabs have null strings. All other index stabs (N_SO, N_OPT, N_MAIN, N_ILDPAD, and N_UNDF) are identical to their corresponding regular stabs.

Here is an example of the index stabs and regular stabs for a hello world program:

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    printf("Hello world\n");
}
```

```
% cc -c -g hello.c
% cc -g hello.o
% dumpstabs -s a.out
```

```
Debugging Stab table -- 60 entries

   0:  .stabs "hello.c",N_UNDF,0x0,0x3b,0x567
   1:  .stabs "/home/ohair",N_SO,0x0,0x0,0x0
   2:  .stabs "hello.c",N_SO,0x0,0x3,0x0
   3:  .stabs "",N_OBJ,0x0,0x0,0x0
   4:  .stabs "",N_OBJ,0x0,0x0,0x0
   5:  .stabs "V=8.0;DBG_GEN=4.0.83;Xa;g;R=Forte Developer 7 C 5.4
EA1
2001/10/21;G=$XA28kkBin_H8CsY.",N_OPT,0x0,0x0,0x3c1fe9e2
   6:  .stabs "char:t(0,1)=bsc1;0;8",N_LSYM,0x0,0x0,0x0
   7:  .stabs "short:t(0,2)=bs2;0;16",N_LSYM,0x0,0x0,0x0
   8:  .stabs "int:t(0,3)=bs4;0;32",N_LSYM,0x0,0x0,0x0
   9:  .stabs "long:t(0,4)=bs4;0;32",N_LSYM,0x0,0x0,0x0
  10:  .stabs "long long:t(0,5)=bs8;0;64",N_LSYM,0x0,0x0,0x0
  11:  .stabs "signed char:t(0,6)=bsc1;0;8",N_LSYM,0x0,0x0,0x0
  12:  .stabs "signed short:t(0,7)=bs2;0;16",N_LSYM,0x0,0x0,0x0
  13:  .stabs "signed int:t(0,8)=bs4;0;32",N_LSYM,0x0,0x0,0x0
  14:  .stabs "signed long:t(0,9)=bs4;0;32",N_LSYM,0x0,0x0,0x0
  15:  .stabs "signed long
long:t(0,10)=bs8;0;64",N_LSYM,0x0,0x0,0x0
  16:  .stabs "unsigned char:t(0,11)=buc1;0;8",N_LSYM,0x0,0x0,0x0
  17:  .stabs "unsigned short:t(0,12)=bu2;0;16",N_LSYM,0x0,0x0,0x0
  18:  .stabs "unsigned:t(0,13)=bu4;0;32",N_LSYM,0x0,0x0,0x0
  19:  .stabs "unsigned int:t(0,14)=bu4;0;32",N_LSYM,0x0,0x0,0x0
  20:  .stabs "unsigned long:t(0,15)=bu4;0;32",N_LSYM,0x0,0x0,0x0
  21:  .stabs "unsigned long
long:t(0,16)=bu8;0;64",N_LSYM,0x0,0x0,0x0
  22:  .stabs "float:t(0,17)=R1;4",N_LSYM,0x0,0x0,0x0
  23:  .stabs "double:t(0,18)=R2;8",N_LSYM,0x0,0x0,0x0
  24:  .stabs "long double:t(0,19)=R6;16",N_LSYM,0x0,0x0,0x0
  25:  .stabs "void:t(0,20)=bs0;0;0",N_LSYM,0x0,0x0,0x0
  26:  .stabs "/usr/include/stdio.h",N_BINCL,0x0,0x0,0x0
  27:  .stabs "/usr/include/sys/
feature_tests.h",N_BINCL,0x0,0x0,0x0
  28:  .stabs "/usr/include/sys/isa_defs.h",N_BINCL,0x0,0x0,0x0
  29:  .stabn N_EINCL,0x0,0x0,0x0
  30:  .stabn N_EINCL,0x0,0x0,0x0
  31:  .stabs "/usr/include/sys/va_list.h",N_BINCL,0x0,0x0,0x0
  32:  .stabs "__va_list:t(4,1)=*(0,20)",N_LSYM,0x0,0x0,0x0
  33:  .stabn N_EINCL,0x0,0x0,0x0
  34:  .stabs "/usr/include/stdio_tag.h",N_BINCL,0x0,0x0,0x0
  35:  .stabs "__FILE:t(5,1)=xs__FILE:",N_LSYM,0x0,0x0,0x0
  36:  .stabn N_EINCL,0x0,0x0,0x0
  37:  .stabs "/usr/include/stdio_impl.h",N_BINCL,0x0,0x0,0x0
  38:  .stabs "ssize_t:t(6,1)=(0,3)",N_LSYM,0x0,0x4,0x0
```

```
39:  .stabs
"__FILE:T(6,2)=s16_cnt:(6,1),0,32;_ptr:(6,3)=*(0,11),32,32;_base
:(6,3),64,32;_fl
ag:(0,11),96,8;_file:(0,11),104,8;__orientation:(0,14),112,2;__i
onolock:(0,14),1
14,1;__filler:(0,14),115,5;",N_LSYM,0x0,0x10,0x0
  40:  .stabn N_EINCL,0x0,0x0,0x0
  41:  .stabs "FILE:t(1,1)=(6,2)",N_LSYM,0x0,0x10,0x0
  42:  .stabs "size_t:t(1,2)=(0,14)",N_LSYM,0x0,0x4,0x0
  43:  .stabs "__longlong_t:t(1,3)=(0,5)",N_LSYM,0x0,0x8,0x0
  44:  .stabs "off_t:t(1,4)=(0,4)",N_LSYM,0x0,0x4,0x0
  45:  .stabs "off64_t:t(1,5)=(0,5)",N_LSYM,0x0,0x8,0x0
  46:  .stabs "fpos_t:t(1,6)=(0,4)",N_LSYM,0x0,0x4,0x0
  47:  .stabs "fpos64_t:t(1,7)=(0,5)",N_LSYM,0x0,0x8,0x0
  48:  .stabn N_EINCL,0x0,0x0,0x0
  49:  .stabs
"main:F(0,3);(0,3);(0,22)=*(0,21)=*(0,1)",N_FUN,0x0,0x0,0x0
  50:  .stabs "main",N_MAIN,0x0,0x0,0x0
  51:  .stabs "argc:p(0,3)",N_PSYM,0x0,0x4,0x44
  52:  .stabs "argv:p(0,22)",N_PSYM,0x0,0x4,0x48
  53:  .stabn N_LBRAC,0x0,0x1,0xc
  54:  .stabs
"$XB28kkBin_H8CsY.main.__func__:V(0,24)=ar(0,4);0;4;(0,23)=k(0,1
)",N_ROSYM,0x0,0
x5,0x0
  55:  .stabn N_SLINE,0x0,0x4,0xc
  56:  .stabn N_SLINE,0x0,0x5,0x24
  57:  .stabn N_RBRAC,0x0,0x1,0x24
  58:  .stabs
"printf:P(0,3);(0,26)=*(0,25)=k(0,1);0",N_FUN,0x0,0x0,0x0
  59:  .stabn N_ENDM,0x0,0x0,0x0
```

```
dumpstabs -s hello.c
```

```
Index Stab table -- 9 entries

   0:  .stabs "hello.c",N_UNDF,0x0,0x8,0xf7
   1:  .stabs "/home/ohair",N_SO,0x0,0x0,0x0
   2:  .stabs "hello.c",N_SO,0x0,0x3,0x0
   3:  .stabs "/home/ohair",N_OBJ,0x0,0x0,0x0
   4:  .stabs "hello.o",N_OBJ,0x0,0x0,0x0
  5:  .stabs "V=8.0;DBG_GEN=4.0.83;Xa;g;R=Forte Developer 7 C 5.4
EA1
2001/10/21;G=$XA28kkBin_H8CsY.",N_OPT,0x0,0x0,0x3c1fe9e2
  6:  .stabs "/home/ohair; /set/dist/sparc-S2/current/bin/../YNH/
bin/cc -g -c
hello.c -W0,-xp\$XA28kkBin_H8CsY.",N_CMDLINE,0x0,0x0,0x0
   7:  .stabs "main",N_MAIN,0x0,0x0,0x0
   8:  .stabs "main",N_FUN,0x0,0x0,0x0
```

APPENDIX **E**

Fortran 95 Pointers and Array Descriptors

This appendix describes the implementation of Fortran 95 pointers and array pointers for the Solaris operating environment.

Terminology

If a Fortran 95 object possesses the `DIMENSION` attribute, it is an *array*; otherwise, it is a *scalar*. If an object possesses the `POINTER` attribute, it is a *pointer*. An object that possesses both the `DIMENSION` attribute and the `POINTER` attribute is an *array pointer*. An object that possesses the `POINTER` attribute but not the `DIMENSION` attribute is a *scalar pointer*.

In Fortran 95, a pointer that references a data object is *associated* with that object. A pointer that does not reference a data object is *disassociated*. A disassociated pointer is the Fortran 95 equivalent of a null pointer. The data object referenced by an associated pointer is the *target* of the pointer.

Run-time Representations

Some of the properties of a pointer object are known at compile time. The declarations for a Fortran 95 data object determine its type. The declarations for a character object also determine its length. The length of a character object need not be constant; it can be an asterisk or an integer expression of a special form called a *specification expression*. The declarations of an array object must include the

DIMENSION attribute, which determines the rank of the array object. The declaration of an array pointer does not determine the bounds of the arrays the pointer can reference.

Fortran 95 represents a scalar pointer value as a single address. For the SPARC V8 architecture, a scalar pointer is 32 bits wide. For the SPARC V9 architecture, a scalar pointer is 64 bits wide. An associated scalar pointer contains the address of the start of the target object. A disassociated scalar pointer contains the address zero.

Fortran 95 represents a array pointer value as a structured value. If the rank of an array pointer value is n, its representation consists of the following sequence of fields:

Field Name	Type
actual-origin	Address
extent	array[1:n] of unsigned integers
stride	array[1:n] of unsigned integers
virtual-origin	address
lower-bound	array[1:n]] of signed integers

The field named actual-origin contains the address of the array with which the pointer value is associated. If the pointer value is not associated with a target object, actual-origin contains the address zero and the rest of the fields are undefined. For an associated pointer value, the value of actual-origin is the address of the start of the first element referenced by the pointer value. The target array object might be contained within a larger array object. In that case, the actual-origin need not be the address of the first element of that larger object.

The field named extent contains the shape of the associated array object. The shape is an n-element vector. The i-th element of extent is the extent of the i-th dimension of the target array. The extent of a dimension is the number of elements in that dimension.

The field named stride is used in address calculations. The i-th element of stride contains the difference between the starting addresses of consecutive elements of the i-th dimension of the array.

The field named virtual-origin contains the address of the start of the possibly hypothetical element of the target array whose subscripts are all zeros. The field is used to simplify address calculations. The value of virtual-origin can be computed from the other fields of a pointer value.

The field named `lower-bound` contains the lower bounds of the target array. The i-th element of `virtual-origin` is the lower bound of the i-th dimension. The upper bounds can be computed from the lower bounds and the extents. The upper bound of the i-th dimension of the target array is

```
lower-bound[i] + extent[i] - 1.
```

For SPARC V8, the addresses and integers in an array pointer are all 32 bits wide. For SPARC V9, the addresses and integers in an array pointer are all 64 bits wide. Fortran 95 allows integers wider than the default integer type to appear in bounds expressions, but it allows implementations to restrict the range of acceptable values for array bounds. Furthermore, the routines for querying the values of array bounds return values of the default integer type.

The lengths of character pointers are not part of the pointer values. The lengths of static character objects and of a character objects in structures are required to be compile-time constants. The length of a character object that is local to a program unit is evaluated on entry to that program unit and remains fixed until the program unit is exited. Assignments and allocations do not affect the lengths of character objects.

Example

Let `A` be an array pointer declared as

```
CHARACTER, POINTER, DIMENSION(:,:) :: A*11
```

The front-end allocates space for `A` either on the stack or in a static area. After successful execution of the statement

```
ALLOCATE( A(-2:12, 0:9) )
```

the space allocated for `A` contains

`actual-origin`	The address of a 1650-byte block of memory
`extent[1]`	15
`extent[2]`	10
`stride[1]`	11
`stride[2]`	165
`virtual-origin`	`actual-origin` + 22
`lower-bound[1]`	-2
`lower-bound[2]`	0

Subscripting

Given a rank n array pointer that is associated with a target array, the address of an element of the target array is

$$\text{virtual-origin} + \sum_{i=1}^{n} \text{stride}[i]*\text{subscript}[i]$$

where `subscript[i]`, for `1 <= i <= n`, is the index of the particular element. The field `actual-origin`, the array `extent`, and the array `lower-bound` are not involved in subscript calculations.

The same address could be calculated as

$$\text{actual-origin} + \sum_{i=1}^{n} \text{stride}[i]*(\text{subscript}[i] - \text{lower-bound}[i]).$$

The two expressions produce the same result because the virtual origin is set to

$$\text{actual-origin} - \sum_{i=1}^{n} \text{stride}[i]*\text{lower-bound}[i].$$

Since the virtual origin is determined by other elements of the dope vector, it could be eliminated without loss of functionality. The reason for retaining it is to avoid some loads and subtracts in subscript calculations.

Whole Array Operations

Whole array operations operate over all of the elements of an array. No particular order of application is implied. A possible translation of a whole array operation applied to a rank n array pointer is

```
address[n] = actual-origin;
for (i[n] = extent[n]; i[n] > 0; --i[n])
{
    address[n-1] = address[n];
    for (i[n-1] = extent[n-1]; i[n-1] > 0; --i[n-1])
    {
        ...
                address[1] = address[2];
                for (i[1] = extent[1]; i[1] > 0; --i[1])
                {
```

```
                    Operation(address[1]);
                    address[1] += stride[1];
                }
            ...
            address[n-1] += stride[n-1];
        }
        address[n] += stride[n];
    }
```

The loops are decremental rather than incremental to simplify the test for the end-condition. The order of the loops was chosen to make the order of the addresses presented to the operation monotonic increasing.

Memory Management

Fortran 95 dynamically allocates storage for automatic arrays, allocatable arrays, pointer targets, and return values for array-valued functions. Sun Fortran 95 ultimately uses the routine `malloc` to allocate dynamic storage and the routine `free` to free it. `malloc` and `free` are used because of the need for compatibility with routines written in C. Both user codes and routines in the system libraries use the set of routines based on malloc and free.

Fortran 95 normally uses the system library versions of `malloc` and `free`. Users are allowed to substitute other versions of `malloc` and `free` if they choose to do so.

APPENDIX **F**

Globalization

For the functions recompiled by fix and continue to be able to access the current values in the executable, all file static variables in the executable must converted into global values. The exception to this rule is nested static variables within a function, which must instead generate a warning message. This *globalization* is performed by the compiler.

The first time a file is compiled, the compiler creates a unique prefix and prepends it to each file static variable. The unique symbol consists of the Internet address (or 0), time of day, and process ID (16-bits only.) The globalization prefix is encoded into the `N_OPT` stab (see "N_OPT — Options (0x3c)" on page 34) with the form:

```
G=prefix
```

The globalization prefix is of the form:

```
.Internet_address  time_of_day  process_ID
```

The 80-bits are encoded as 14 characters, where each character contains 6-bits of information. The characters `a-z`, `A-Z`, `0-9`, `_`, and `$` are used to store the 6-bits.

The following code shows how various static variables are globalized:

```
int a;  /* a */
static int b;/* .<prefix>.b */
int
main(void)
{
    static int c;/* ..main.c */
    {
    static int d;/* Warning message */
    }
    return 0;
}
static void
foo(void)
{
    static int e;/* .<prefix>.foo.e */
    {
    static int f;/* Warning message */
    }
}
```

This code generates the following stabs:

```
Index Stab table -- 11 entries

   0:  .stabs "ex4.c",N_UNDF,0x0,0xa,0xf2
   1:  .stabs "/home/ohair",N_SO,0x0,0x0,0x0
   2:  .stabs "ex4.c",N_SO,0x0,0x3,0x0
   3:  .stabs "",N_OBJ,0x0,0x0,0x0
   4:  .stabs "",N_OBJ,0x0,0x0,0x0
  5:  .stabs "V=8.0;DBG_GEN=4.0.83;Xa;g;R=Forte Developer 7 C 5.4
EA1
2001/10/21;G=$XA28kkBzr_H8StY.",N_OPT,0x0,0x0,0x3c1feaf3
  6:  .stabs "/home/ohair; /set/dist/sparc-S2/current/bin/../YNH/
bin/cc -g -c
ex4.c -W0,-xp\$XA28kkBzr_H8StY.",N_CMDLINE,0x0,0x0,0x0
   7:  .stabs "main",N_MAIN,0x0,0x0,0x0
   8:  .stabs "main",N_FUN,0x0,0x0,0x0
   9:  .stabs "$XA28kkBzr_H8StY.b",N_GSYM,0x0,0x0,0x0
  10:  .stabs "a",N_GSYM,0x0,0x0,0x0
```

```
Excluded Stab table -- 49 entries

   0:  .stabs "ex4.c",N_UNDF,0x0,0x30,0x392
   1:  .stabs "/home/ohair",N_SO,0x0,0x0,0x0
   2:  .stabs "ex4.c",N_SO,0x0,0x3,0x0
   3:  .stabs "",N_OBJ,0x0,0x0,0x0
   4:  .stabs "",N_OBJ,0x0,0x0,0x0
   5:  .stabs "V=8.0;DBG_GEN=4.0.83;Xa;g;R=Forte Developer 7 C 5.4
EA1
2001/10/21;G=$XA28kkBzr_H8StY.",N_OPT,0x0,0x0,0x3c1feaf3
   6:  .stabs "char:t(0,1)=bsc1;0;8",N_LSYM,0x0,0x0,0x0
   7:  .stabs "short:t(0,2)=bs2;0;16",N_LSYM,0x0,0x0,0x0
   8:  .stabs "int:t(0,3)=bs4;0;32",N_LSYM,0x0,0x0,0x0
   9:  .stabs "long:t(0,4)=bs4;0;32",N_LSYM,0x0,0x0,0x0
  10:  .stabs "long long:t(0,5)=bs8;0;64",N_LSYM,0x0,0x0,0x0
  11:  .stabs "signed char:t(0,6)=bsc1;0;8",N_LSYM,0x0,0x0,0x0
  12:  .stabs "signed short:t(0,7)=bs2;0;16",N_LSYM,0x0,0x0,0x0
  13:  .stabs "signed int:t(0,8)=bs4;0;32",N_LSYM,0x0,0x0,0x0
  14:  .stabs "signed long:t(0,9)=bs4;0;32",N_LSYM,0x0,0x0,0x0
  15:  .stabs "signed long
long:t(0,10)=bs8;0;64",N_LSYM,0x0,0x0,0x0
  16:  .stabs "unsigned char:t(0,11)=buc1;0;8",N_LSYM,0x0,0x0,0x0
  17:  .stabs "unsigned short:t(0,12)=bu2;0;16",N_LSYM,0x0,0x0,0x0
  18:  .stabs "unsigned:t(0,13)=bu4;0;32",N_LSYM,0x0,0x0,0x0
  19:  .stabs "unsigned int:t(0,14)=bu4;0;32",N_LSYM,0x0,0x0,0x0
  20:  .stabs "unsigned long:t(0,15)=bu4;0;32",N_LSYM,0x0,0x0,0x0
  21:  .stabs "unsigned long
long:t(0,16)=bu8;0;64",N_LSYM,0x0,0x0,0x0
  22:  .stabs "float:t(0,17)=R1;4",N_LSYM,0x0,0x0,0x0
  23:  .stabs "double:t(0,18)=R2;8",N_LSYM,0x0,0x0,0x0
  24:  .stabs "long double:t(0,19)=R6;16",N_LSYM,0x0,0x0,0x0
  25:  .stabs "void:t(0,20)=bs0;0;0",N_LSYM,0x0,0x0,0x0
  26:  .stabs "main:F(0,3)",N_FUN,0x0,0x0,0x0
  27:  .stabs "main",N_MAIN,0x0,0x0,0x0
  28:  .stabn N_LBRAC,0x0,0x1,0x4
  29:  .stabs "$XB28kkBzr_H8StY.main.c:V(0,3)",N_LCSYM,0x0,0x4,0x0
  30:  .stabs
"$XB28kkBzr_H8StY.main.__func__:V(0,22)=ar(0,4);0;4;(0,21)=k(0,1
)",N_ROSYM,0x0,0
x5,0x0
  31:  .stabn N_LBRAC,0x0,0x2,0x4
  32:  .stabs "d:V(0,3)",N_LCSYM,0x0,0x4,0x4
  33:  .stabn N_RBRAC,0x0,0x2,0x4
  34:  .stabn N_SLINE,0x0,0xa,0x4
```

```
  35:  .stabn N_SLINE,0x0,0xb,0x14
  36:  .stabn N_RBRAC,0x0,0x1,0x14
  37:  .stabs "foo:f(0,20)",N_FUN,0x0,0x0,0x0
  38:  .stabn N_LBRAC,0x0,0x1,0x4
  39:  .stabs "$XB28kkBzr_H8StY.foo.e:V(0,3)",N_LCSYM,0x0,0x4,0x0
  40:  .stabs
"$XB28kkBzr_H8StY.foo.__func__:V(0,24)=ar(0,4);0;3;(0,23)=k(0,1)
",N_ROSYM,0x0,0x
4,0x0
  41:  .stabn N_LBRAC,0x0,0x2,0x4
  42:  .stabs "f:V(0,3)",N_LCSYM,0x0,0x4,0xc
  43:  .stabn N_RBRAC,0x0,0x2,0x4
  44:  .stabn N_SLINE,0x0,0x13,0xc
  45:  .stabn N_RBRAC,0x0,0x1,0xc
  46:  .stabs "$XA28kkBzr_H8StY.b:S(0,3)",N_LCSYM,0x0,0x4,0x0
  47:  .stabs "a:G(0,3)",N_GSYM,0x0,0x4,0x0
  48:  .stabn N_ENDM,0x0,0x0,0x0
```

The prefix format is a dot '.' or a dollar sign '$' followed by uppercase 'X', one of the letters ABC, a unique pattern of characters, an optional trailing function name, and an optional variable name:

```
{.$}X{ABC}uniquepattern[.function_name][EQUIVn][.variable_name]
```

For example:

```
$XA28kkBzr_H8StY.b
$XB28kkBzr_H8StY.main.c
$XC28kkBt1_H8CwY.funcEQUIV1_0.i
```

where:

A — Is for file static variables

B — Is for function static variables

C — Is for Fortran equivalence blocks (the value field of the stab indicates the offset into the block) and the *.variable_name* should be removed from the name to find the Elf symbol that represents this block of static data.

APPENDIX **G**

Differential Mangling

The C++ compiler generates "mangled" names to obtain unique symbols for linking programs and for other reasons.

In stabs, some of these names are mangled further in order to conserve string space—a technique called differential mangling. Here is an example:

```
namespace foo {
    class bar {
        void buz();
    };
};
foo::bar obj;
```

The applicable stabs generated for a -compat=5 (ABI-2) compile are:

```
Index Stab table -- 10 entries

   7:  .stabs "__1nDfooDbar_:U",N_ESYM,0x0,0x0,0x0

Excluded Stab table -- 33 entries

  25:  .stabs "__1nDfoo_:T(0,19)=Yn0foo;",N_ISYM,0x0,0x0,0x0
  26:  .stabs "nDbar(0,19):U(0,20)",N_ESYM,0x0,0x0,0x0
  27:  .stabs
"nDbar(0,19):T(0,20)=Yc1bar;;;AcDbuz6M_v;;;;;;;000;",N_ESYM,\
              0x0,0x1,0x0
  28:  .stabs
"cDbuz6M_v(0,20):P(0,13);(0,21)=*(0,20)",N_FUN,0x0,0x0,0x0
  31:  .stabs "obj:G(0,20)",N_GSYM,0x0,0x1,0x0
```

Note that names that appear in index stabs are not differentially mangled, while names that appear in excluded stabs are (compare index stab 7 with stab 26). The compiler constructs a differential name based on the member name and the type

number of the containing class or namespace (for example, `cDbuz6M_v(0,20)` in stab 28). In class (struct) stabs themselves, the list of members is differentially mangled without type number (for example, `cDbuz6M_v` in stab 27).

The analysis of mangled names for `buz` is:

	ABI-2	ABI-1
prefix	`__1`	`__0`
type	`c`	`f`
outer	`Dfoo`	`Dfoo`
separator	`5`	
inner	`Dbar`	`Dbar`
member	`Dbuz6M_v`	`Dbuzv`
suffix	`_`	

The differential algorithm for ABI-1 is:

container	member	differential
`__0`	`__0`	
`T6`	`f`	`f`
`Dfoo`	`Dfoo`	
	`5`	`E`
`Dbar`	`Dbar`	
	`Dbuzv`	`Dbuzv`

(The `E` is the number of characters (4) in outer portion of the mangled name. If the length is ‘A’ (0), there is no separator.)

The differential algorithm for ABI-2 is:

container	member	differential
`__1`	`__1`	
`n`	`c`	`c`

container	member	differential
DfooDba	DfooDbar	
	Dbuz6M_v	Dbuz6M_v
_	_	

Sun Microsystems, Inc has a patent pending for the differential mangling algorithm.

Glossary

ELF The Executable and Linking Format stores executable files, shared libraries, and object files on the Solaris operating environment. The format is fully described in the *System V Application Binary Interface* and the *SPARC Processor Supplement*, both of which are produced by AT&T and published by Prentice-Hall.

stab Symbol table entry, used throughout this document to refer to those generated by the debugger.

`.stabs` Assembler pseudo-directive that generates a symbol table entry.

`.stabn` Assembler pseudo-directive that generates a symbol table entry similar to .stabs, except with a null string.

symbol descriptor A character indicating the kind of symbol being described. It follows the name of the symbol and is separated from it by a colon.

type A description of the ways in which a symbol can be used. Basic types are integer and float. All other types are constructed from these basic types by collecting them into arrays, structures, and so forth, or by adding modifiers, such as pointer-to or constant attributes.

`.xstabs` Assembler pseudo-directive that generates a symbol table entry and permits the user to specify which section in an ELF file the stab is to be stored.

`24405`Assembler pseudo-directive that generates a symbol table entry for a COMDAT stab and permits the user to specify which section in an ELF file the stab is to be stored.

type description A substring of characters within a stab that describes a type. The first character of the substring determines the syntax of the characters that follow. Type descriptions can be nested.

type number A compiler-generated unique identifier for a type used in stabs. It can take one of the following forms: a number, (file_id, number), or (number).

type specification A reference to a previously defined type number, a new definition of a type number, or a type description.