Stabs Interface Manual

Version 4.0

THE NETWORK IS THE COMPUTER™

Sun Microsystems, Inc.
901 San Antonio Road Palo Alto, CA 94303-4900 USA
650 960-1300
fax 650 969-9131

Please
Recycle

Adobe PostScript

Contents

Introduction 1

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`** and this document describes how these stabs are created, stored, modified and interpreted.

The debugger supports two formats for object files: a.out (as generated under the SunOS operating environment) and ELF (generated under the Solaris operating environment). Although these two formats contain similar information, they have significantly different structure. Both object file formats will be described. The stabs generated to support debugging have similar format and meaning in either format.

The name *stabs* comes from symbol table entries, and, in an a.out file, the stabs are merged with the symbol table generated by the compiler and linker. In an ELF file, the stabs are stored in separate sections from the symbol table generated by the compiler and linker.

The symbol table entries stored in the a.out object file by the compiler so that the files can be linked together are also called stabs. Within this document, when the word *stabs* is used, we mean the debugging stabs, and not the stab entries generated by a compiler to support linking or loading. If we need to refer to the symbol table entries generated by the compiler and linker, we will refer to them as linker stabs to avoid confusion.

Although **dbx** is the most common interpreter of stabs, other programs do use the information in stabs in one form or another (for example, Source Browser and SunPerf).

This document describes 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, and how **dbx** interprets the stabs. A glossary is included to define the various terms used.

Overview 2

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++, Pascal, Fortran77, Fortran90 and C, reduce the size of executable files, support additional operating system features such as dynamically loaded shared libraries, and 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 format:

`.stabs "`*string*`", `*type*`, 0, `*desc*`, `*value*

and

`.stabn `*type*`, 0, `*desc*`, `*value*

The *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. The *type* field specifies the type of the stab entry, the *desc* field is used in some stab types to further describe the symbol, and the *value* field contains an address, offset, or other value. The *.stabn* directive is identical to the *.stabs* directive, except there is no string.

In an a.out file, debugging stabs are intermingled with the linker stabs in the symbol table. In an ELF file, debugging stabs are stored in the *.stab* section with the text of the strings in *.stabstr*. The *.xstabs* directive can be used to direct the stab into a different section in an ELF file.

The string in the *.stabs* directive may be of any length, up to the string size limit imposed by the assembler. 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* fields. 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* or *.stabn* directive contains a stab type which 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 which 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*.

ELF Object File Format 3

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]. In this section we will briefly describe the format, focusing on the aspects which 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 */
};
```

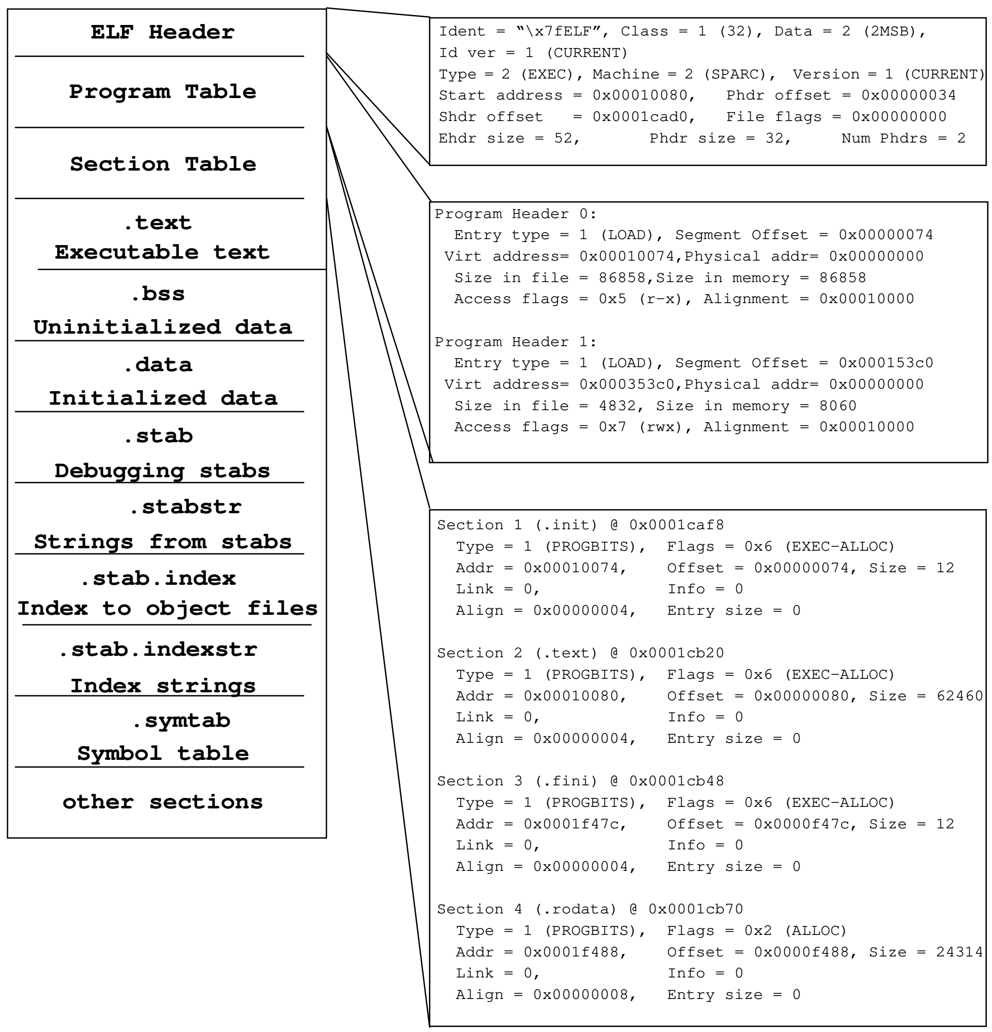
The `n_strx` field is the offset of the string in the symbol string table. All strings are terminated by a null byte. Previous stabs versions defined struct nlist in `nlist.h`; this has been changed to avoid conflict with the system header file and struct of that name. The fields struct stab have also changed to accomodate use with 64-bit (Solaris 7) programs.

1.

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

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 then a number of named sections. The section table is an index to each of the sections, describing the 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 which have contents which are specified by the vendor, such as the `.stab` section. The following figure shows the layout of an ELF executable file.

ELF Executable Layout



The symbol table is in the section named `.symtab` and the strings are stored in the `.stabstr` section. Each entry in the symbol table consists of a name, a value (which is usually an address), a size, type and binding flags, and 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. This is just the file name, 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.

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. Note that 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. There is nothing in the symbol table which 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 `.stabs` 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 which are used to support auto-load stab processing. These stabs specify the names of global functions and data which 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. A more complete description will be presented later.

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 stab contains the offset in strings for the object file, rather than the offset from the beginning of the string table as is the case in an a.out file. The linker does not relocate or modify the offset to the strings.

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

ELF Stabs Sections

```
Debugging Stab table (.stab) -- 33 entries

   0:  .stabs "busy.c",N_UNDF,0x0,0x20,0x268
   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 "Xt ; g ; V=2.0",N_OPT,0x0,0x0,0x2a009ce6
   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
                              .
                              .
                              .
  21:  .stabs "float:t(0,16)=R1;4;",N_LSYM,0x0,0x0,0x0
  22:  .stabs "double:t(0,17)=R2;8;",N_LSYM,0x0,0x0,0x0
  23:  .stabs "long double:t(0,18)=R6;16;",N_LSYM,0x0,0x0,0x0
  24:  .stabs "void:t(0,19)=bs0;0;0",N_LSYM,0x0,0x0,0x0
  25:  .stabs "main:F(0,3)",N_FUN,0x0,0x0,0x105e8
  26:  .stabs "main",N_MAIN,0x0,0x0,0x0
  27:  .stabn N_LBRAC,0x0,0x1,0xc
  28:  .stabn N_SLINE,0x0,0x5,0xc
  29:  .stabn N_SLINE,0x0,0x6,0x14
  30:  .stabn N_RBRAC,0x0,0x1,0x14
  31:  .stabs "lazy:P(0,3)",N_FUN,0x0,0x0,0x0
  32:  .stabs "",N_ENDM,0x0,0x0,0x0


Index Stab table (.stab.index) -- 22 entries

   0:  .stabs "values-Xt.c",N_UNDF,0x0,0x6,0x49
   1:  .stabs "/build/src/lib/libc/",N_SO,0x0,0x0,0x0
   2:  .stabs "port/gen/values-Xt.c",N_SO,0x0,0x3,0x0
   3:  .stabs "/usr/src/play",N_OBJ,0x0,0x0,0x0
   4:  .stabs "/usr/ccs/lib/values-Xt.o",N_OBJ,0x0,0x0,0x0
   5:  .stabs "Xt",N_OPT,0x0,0x0,0x0
   6:  .stabs "_lib_version",N_GSYM,0x0,0x0,0x0
   7:  .stabs "busy.c",N_UNDF,0x0,0x7,0x4c
   8:  .stabs "/usr/src/play/",N_SO,0x0,0x0,0x0
   9:  .stabs "busy.c",N_SO,0x0,0x3,0x0
  10:  .stabs "/usr/src/play",N_OBJ,0x0,0x0,0x0
  11:  .stabs "busy.o",N_OBJ,0x0,0x0,0x0
  12:  .stabs "Xt ; g ; V=2.0",N_OPT,0x0,0x0,0x2a009ce6
  13:  .stabs "main",N_MAIN,0x0,0x0,0x0
  14:  .stabs "main",N_FUN,0x0,0x0,0x0
  15:  .stabs "lazy.c",N_UNDF,0x0,0x6,0x8f
  16:  .stabs "/usr/src/play/",N_SO,0x0,0x0,0x0
  17:  .stabs "lazy.c",N_SO,0x0,0x3,0x0
  18:  .stabs "/usr/src/play",N_OBJ,0x0,0x0,0x0
  19:  .stabs "lazy.o",N_OBJ,0x0,0x0,0x0
  20:  .stabs "Xt ; g ; V=2.0",N_OPT,0x0,0x0,0x2a009cec
  21:  .stabs "lazy",N_FUN,0x0,0x0,0x0
```

Debugger Stabs 4

The *.stab* 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.

We describe each stab in alphabetic order by type name. For convenience, the a.out linker stab codes are also listed.

N_ABS — Absolute Symbol (a.out only) (0x02)

This is an a.out linker stab, which indicates that the specified symbol is an absolute symbol. The linker will not relocate any value with this type. See the description of `a.out` files in Chapter 3, “The a.out Object File Format” for further details.

N_ALIAS - Symbol Alias (0x6c)

This 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***. In the following:

```
.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 due to 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, and ***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 that the ***externalname*** ends up in.

This stab is a no-op when: 1) the ***externalname*** does not exist, or 2) 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 new prefix **S**. The stabs can be position dependent or position independent, depending on the scopes the aliases occur in.

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
```

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 up to 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. On the SunOS operating environment, refer to Chapter 3, "The a.out Object File Format".

A **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** 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.

N_BINCL must follow an **N_SO** for a source file and precede the matching **N_ENDM**. **N_BINCL**s and **N_EINCL**s 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
```

This stab specifies the path to the `.bd` file created to support SourceBrowser. On the Solaris operating environment, this stab is entered into the `.stab.sbfocus` section. On the SunOS operating environment, this stab is entered in the `.stab` section.

N_BSS — BSS Segment Symbol (a.out only) (0x08)

This is an a.out linker symbol, which indicates that the symbol is in the `bss` segment. The linker relocates the value to be the address within this segment. See the description of a.out files in Chapter 3, "The a.out Object File Format", for further details.

N_CMDLINE - Compilation Command Line (0x34)

To support "Fix and Continue", this 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, see the example below. Note, care must be taken to ensure that multiple "Fix and Continue" runs do not keep duplicating prefix.

Together, ***driver options prefix***, make up a complete compile line. This compile line is passed to `/bin/sh`, so all special characters must be quoted with a back-slash (e.g., '$').

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_COMM — Common Symbol (a.out only) (0x12)

This is an a.out linker symbol, which is used internally by the loader. It is also used by the runtime checking (RTC) facility in dbx.

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, the ***Var*** field 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). ***End*** is the location of the instruction immediately following the object's construction (regardless of whether it's a call or is done inline), and ***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, you are in state ***State***.

N_DATA — Data Segment Symbol (a.out only) (0x06)

This is an a.out linker symbol, which indicates that the symbol is in the data segment. The linker relocates the value to be the address within this segment. See the description of a.out files in Chapter 3, "The a.out Object File Format" for further details.

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.

The ***State*** field 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 15.

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 15.

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** 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 that this procedure was entered through (useful in determining a correct traceback) . This special symbol is only present 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`. Note that 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_EXCL — Excluded Include File (0xc2)

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

The **N_EXCL** stab indicates that the linker has seen a sequence of stabs, beginning with an **N_BINCL** stab and ending with an **N_EINCL** stab, that are identical to a sequence that was previously seen in the same executable or shared library. Instead of copying all of the stabs, the linker generates the **N_EXCL** stab with the same ***FilePath*** and ***HashValue*** as the **N_BINCL** stab. A compiler should never generates this stab.

It is an error if there is no previous **N_BINCL** stab with the same ***FilePath*** and ***HashValue***.

If `h.h` includes `i.h`, and both are included in the compiled source, the following stabs are generated:

```
.stabs "./h.h",N_BINCL,0x0,0x0,0x268
.stabs "s:T(1,1)=s8x:(0,3),0,32;y:(0,3),32,32;;",
        N_LSYM,0x0,0x8,0x1
.stabs "./i.h",N_BINCL,0x0,0x0,0x388
.stabn N_EINCL,0x0,0x0,0x0
.stabn N_EINCL,0x0,0x0,0x0
.stabs "./i.h",N_EXCL,0x0,0x0,0x388
```

The first and third stab are **N_BINCL** stabs, which identify the names of include files that are included in the compilation. The second stab is a local structure definition in h.h. The fourth and fifth stabs are the **N_EINCL** stabs, which indicate the ends of the stabs for the include file. The last stab is the **N_EXCL** stab indicates that the stabs that appeared in the previous occurrence of `i.h` have been removed by the linker.

N_EXT — External Symbol Flag (a.out only) (0x01)

This flag may be combined with any of the a.out linker stabs (**N_UNDF**, **N_ABS**, **N_TEXT**, **N_DATA**, **N_BSS**, of **N_COMM**) to indicate that the symbol is a global (or external) symbol. See the description of a.out files in Chapter 3, "The a.out Object File Format" for further details.

N_FN — File Name Symbol (a.out only) (0x1F)

This is an a.out linker symbol and is ignored by `dbx`. See the a.out man page.

N_FUN — Function or Procedure Definition (0x24)

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

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

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 (Fortran90)
J	Internal procedure (Fortran90)
P	Prototype for function or procedure
Y	C++ specification (see, “C++ Specification (Y)” on page 61)

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*** will show that an `int` is passed to the function while the **N_PSYM** stab for the formal argument will show the formal argument’s declared type.

RtnSize is the size of the returned value. Procedures that do not return a value have ***RtnType***, which specifies the type number of `void` or equivalent. In an a.out file, the ***Address*** is the entry point address of the function. In an ELF file, the address of the function is found in the symbol table. The value of ***SymDesc*** can change this meaning. See the individual symbol descriptor descriptions for specifics.

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_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 (formerly N_FLINE) - Incremental Link Padding (0x4c)

This 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** 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.

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, 0, 0, *Offset*

An **N_LCSYM** stab describes an uninitialized static variable.

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

S	Global
V	Local
b	Fortran “based” variable

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 & Continue feature of dbx, the ***Name*** will be found in the ELF symbol table and the ***Offset*** will be zero.

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_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.

These stabs are only fully specified in an ELF file; in an `a.out` file the **N_OPT** stabs have null strings.

N_OPT — Options (0x3c)

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

The **N_OPT** stabs 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)
g	Debugging stabs were generated for this object file
F	Fragmented
G=<g>	Global prefix is <g>

l	Pascal lower case
nu	No underscores added to Fortran symbols
O	Optimized code
p	Position-independent code (`-p`)
P	Position-independent code (`-P`)
R=xx.xx<r>	Compiler release number is xx.xx ("r" is a version string)
s	Standard Pascal
U	Fortran mixed case variables
V=<v>	Stab version is <v> (must be a numeric string)
Xa	ANSI C promotions
xl	Pascal xl
Xt	ANSI C transition
Xs	ANSI C Sun compatibility

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_PATCH - Patch Run Time Checker (0xd0)

This 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,

and ***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 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:

This 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 "Run Time Checking" to not check for "Read Uninitialized Data" for that load.

When to emit ***P_BITFIELD N_PATCH*** *stabs?*
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 "Run Time Checking" to not perform any checking on these load or store instructions.

When to emit ***P_SPILL N_PATCH*** *stabs?*
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.

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 “Run Time Checking” to not perform any checking on these load instructions.

*When to emit **P_SCOPY N_PATCH** stabs?*
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_PC — Global Pascal Symbol (0x30)

```
.stabs "Ident", N_PC, 0, 0, 0
```

This stab provides identification information used by Pascal utilities. It is ignored by `dbx`.

N_PSYM — Formal Parameter (0xa0)

```
.stabs "Name : SymDesc Type", N_PSYM, 0, 0, Offset
```

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

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

b	Based variable
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

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 23.

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_RBRAC** 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 26.

N_ROSYM — Read-Only Static Symbol (0x2c)

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

S Global static variable
V Local static variable
b FORTRAN based variables

In an ELF file, ***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:
```

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

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

If the **N_ROSYM** stab appears outside of a function, only global static (***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 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.

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

Examples of use of the **N_SLINE** are in “N_ILDPAD (formerly N_FLINE) - Incremental Link Padding (0x4c)” on page 25.

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
N_SO_FORTRAN90	**7**	Fortran90 source

For example, if the current working directory is `/usr/example/test` and we compile the source file `../src/ex.f`, 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
.stabs "../src/ex.f",N_SO,0,0
```

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.

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 are 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, 0, 0
```

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

The **N_SOL** stab must be generated before any **N_SLINE** stabs 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`.

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,0,0,0
.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, 0, 0, Offset
```

An **N_STSYM** stab describes an initialized static variable.

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

S	Global static variable
V	Local static variable
b	FORTRAN-based variables

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**.

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

If the **N_STSYM** stab appears outside of a function, only global static (***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 a.out file, a linker **N_DATA** stab is present for the symbol that specifies the variables address.

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.

The 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_TEXT — Text Segment Symbol (a.out only) (0x04)

This is an a.out linker stab that indicates that the symbol is in the text segment. The linker will relocate its value to the correct address in the text segment. See the description of a.out files in Chapter 3, "The a.out Object File Format"," for further details.

N_UNDF — Undefined (0x00)

This 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 "*Objname*", N_UNDF, 0, *NumStabs*, *BytesOfStringTable*

This stab is used for various purposes under the SunOS operating environment. See Chapter 3, "The a.out Object File Format"," for further details. On the Solaris operating environment, the stab us used 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 occuring 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. These look like

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

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

Local USING declaration, position dependent.

These are found in function or local block scopes:

`.stabs "P:<mangled_N::sname>",N_USING,0,0,0`

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

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 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_WITH—Pascal WITH Statement (0xea)

```
.stabs "Name : Type", N_WITH , 0, 0, Offset
```

The **`N_WITH`** stab describes the Pascal `WITH` statement, which specifies the record to use when a partial reference to a field is used. If ***`Name`*** is specified, it is an alias for the record to be used. ***`Type`*** is the type of the record. ***`Offset`*** is the offset of the record (or a pointer to the record) in the current frame.

The **N_WITH** stab must be contained within an **N_LBRAC**/**N_RBRAC** pair, which defines the scope of the `WITH` specification. For example:

```
begin;
    for n := 1 to 100 do
        with ar[n] do begin
                i := 2 * n;
        end;
end.
```

generates the following stabs:

```
.stabn N_LBRAC,0,0x2,LL15
.stabs ":*(0,17)",N_WITH,0x0,0x0,-20
.stabn N_RBRAC,0,0x2,LL17
```

The first stab is the **N_LBRAC**, which begins the scope that the `WITH` statement controls. The second stab specifies that there is no alias (since the name is omitted) and that the value at -20 from the frame pointer is a pointer to type `(0,17)`, which is the type of the `ar` array. The last **N_RBRAC** stab ends the scope of the `WITH` statement.

If the following statement had been used:

```
with ar[n] : x do begin
```

the **N_WITH** stab would have been:

```
.stabs "x:*(0,17)",N_WITH,0x0,0x0,-20
```

N_XCOMM — Excluded Common Block (0xe6)

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

The **N_XCOMM** stab indicates that the stabs for common block *BlockName* has been omitted from the stabs by the Fortran compiler because it was identical to a previous common block. An **N_BCOMM** stab with the same *BlockName* and *HashValue* must precede this stab in the compilation unit.

The Fortran source file

```
    subroutine a
    common /com/ x, y, z
    print *, x
    return
    end

    subroutine b
```

```
common /com/ x, y, z
print *, y
return
end
```

generates the following stabs:

```
.stabs "a:F14",N_FUN,0,0,_a_
.stabs "_com_",N_BCOMM,0,0,0x4478
.stabs "x:V6",N_GSYM,0,0,0
.stabs "y:V6",N_GSYM,0,0,4
.stabs "z:V6",N_GSYM,0,0,8
.stabs "_com_",N_ECOMM,0,0,0
.stabs "b:F14",N_FUN,0,0,_b_
.stabs "_com_",N_XCOMM,0,0,0x4478
```

The first stab is the **N_FUN** (function) stab that starts the function `a`. This is followed by an **N_BCOMM** (begin common) stab for common `com`, three **N_GSYM** (global symbol) stabs for the elements in the common and an **N_ECOMM** (end common) stab. The seventh stab is the **N_FUN** for function `b`. The last stab is the **N_XCOMM** stab generated by the compiler to indicate that the previous occurrence of common block `com` is to be repeated where the **N_XCOMM** stab appears.

N_XLINE — Extended Line Number (0x45)

For line numbers greater than 65535, this additional 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`***

Symbol Descriptors 5

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. There is some redundancy and interaction 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	[NOTE: no symbol descriptor has default]
A	Automatic variable (Fortran 90)	
b	Based variable	
C	Conformant array bound	
c	Constant symbol (Fortran90, pascal)	
F	Global function or procedure	
f	Local function or procedure	
G	Global variable	
H	Hosted non-local variable (Fortran90)	
J	Internal procedure (Fortran90)	
LT	Lines in Template	
l	Literal	
M	Module (Fortran90)	
P	Prototype	
p	Value parameter	
r	Register variable	
S	Static global variable	
T	Enumeration, structure or union	

t	Type name
U	Class declaration
V	Common or static local variable
v	Fortran variable parameter by reference
W	Generic name (Fortran90)
X	Function result variable
x	Value parameter by reference (aka, Array value parameter)
Y	C++ specification (C++ 4.0 and later)
Z	Old-format C++ specification (see Appendix D)

We will describe each of these symbol descriptors in alphabetical order and specify which stab types may contain them.

Within the type specifications, C and C++ types are identified by a pair of numbers in parentheses, separated by a comma. For historical reasons, this pair of numbers is called the *type number.* See Chapter 6, "Type Specification for details.

Local Variable (empty)

The absence of a symbol descriptor is used to describe a variable that is local to a function or procedure. It may only appear 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)

This 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 the stab description.

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)

This symbol descriptor is used to describe a Fortran based variable. It may 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 stab description.

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`.

Conformant Array Bound (C)

This symbol descriptor is used to specify the array bounds for a Pascal conformant array. It may appear in an **N_PSYM** (function parameter) stab. The *value* field of the stab specifies the offset of the parameter from the frame pointer.

The following Pascal program defines a procedure `proc2` that accepts a conformant array named `a`. The upper and lower bounds of the array are passed as conformant array bounds `x` and `y`:

```
program test(input,output);

var ain : array[1..2] of integer;

procedure proc2(a : array[x..y : integer] of integer);
var j,k  : integer;

begin
    k := 0;
    for j := x to y
        do k := k + a[j];
end;

begin
    proc2(ain);
end.
```

This program generates (in part) the following stabs:

```
.stabs "integer:t(0,5)=bs4;0;32;",N_LSYM,0x0,0x0,0
.stabs "proc2:F(0,8);(0,17)=cC,,(0,5);(0,5);(0,5);(0,5)",
        N_FUN,0,0,_proc2
.stabs "x:C(0,5)",N_PSYM,0x0,0x0,72
.stabs "y:C(0,5)",N_PSYM,0x0,0x0,76
.stabs "a:xcCx,y,(0,5);(0,5)",N_PSYM,0x0,0x0,68
```

The first stab listed describes the `integer` type. The second describes the global function `proc2` that takes a conformant array of integers. The third and fourth **`N_PSYM`** stabs describe the conformant array bounds, with the specified offsets from the frame pointer. The last stab listed describes the conformant array `a` passed to the function.

Constant (c)

This symbol descriptor is used to describe a constant symbol (e.g., a Fortran `parameter` or Pascal `const`). 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.

Global Function or Procedure (F)

This symbol descriptor is used to describe a global function or subroutine. It may appear only in an **`N_FUN`** or **`N_ALIAS`** stab. In an `a.out` file, the `value` field of the stab is the entry point address of the function. 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)

This symbol descriptor is used to describe a local function or subroutine. It may 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 function. 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)

This symbol descriptor is used to describe a global variable. This descriptor may 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)).

Hosted, Non-Local Variable (H)

This symbol descriptor is used to describe a *hosted* symbol. That is, a symbol which is non-global and non-local but is in scope. It may appear only in an **N_LSYM** stab. The value field of the stab is the byte offset from the host frame pointer. On SPARC, the host frame pointer is passed in global register g2 before the call to the internal procedure is made.

For example:

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

generates the following stab for `y` inside the scope of `subroutine t`:

```
.stabs "y:H3",N_LSYM,0x0,0x0,0xfffffff8
```

While a normal **`N_LSYM`** is generated for `y` inside the scope of subroutine `s`. This describes `y` as a hosted variable of type `int` (type number 3), with byte offset of `0xfffffff8` from the frame pointer.

Internal Procedure (J)

This symbol descriptor is used to describe a Fortran 90 internal procedure. It may 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 "t_in_s:J14",N_FUN,0,0,0
```

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

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

Lines in Template (LT)

In addition to the type information (see the subsection“Templates (YT, YI)” on page 73), each template definition causes an **LT** stab (“Lines in Template”) to be output. The **LT** descriptor may appear in a **N_LSYM** stab. This descriptor allows dbx to tell if you try to “stop in” a function template, even if it’s never been instantiated in this program. There are three kinds of **LT** stabs — **LTc** for class templates, **LTf** for function templates, and **LTm** for member function templates.

The line-stabs for class templates looks like this:

```
Template_name:LTcStarting_line;;Ending_line”, N_LSYM
```

The **c** is for class templates; it is replaced with **f** for (normal) function templates, and **m** for member function templates (used only when a member function is defined outside of the class template). The empty field between the starting and ending line numbers is reserved for future use.

Since template names cannot be mangled and must be unique in a given program, the name is not mangled in **LTc** or **LTf** stabs. It *is* mangled in the **LTm** stabs.

Literal (l)

This new symbol descriptor ‘l’ is used to describe literals, such as true & false of bool type. Although symbol descriptor ‘l’ can be used in N_LSYM & N_GSYM, so far only the use in N_LSYM is identified. So for now only ‘l’ in N_LSYM is supported. Examples:

1) literal true

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

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

Module (M)

This symbol descriptor is used to describe a Fortran 90 module. It may 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.

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)

This descriptor specifies that the symbol is a parameter that is passed to a subroutine or function by value. It may 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 Pascal function

```
procedure x ( l:integer; var i:integer; procedure a(k:integer) );
begin
    i := l;
    a(i);
end;
```

generates (in part) the following stabs:

```
.stabs "x:F(0,8);(0,5);(0,5);(0,17)=P(0,8)",N_FUN,0,0,_x
.stabs "l:p(0,5)",N_PSYM,0x0,0x0,68
.stabs "i:v(0,5)",N_PSYM,0x0,0x0,72
.stabs "a:pP(0,8)",N_PSYM,0x0,0x0,76
```

The first stab is an **N_FUN** (function) stab that describes the procedure *x*. It takes three arguments, two are `integers` (type `(0,5)`) and one is a procedure. The next three **N_PSYM** (parameter) stabs describe these three parameters. The first indicates that `l` is a parameter passed by value; the second indicates that `i` is a var parameter passed by reference, and the last that “`a`” is a procedure parameter. Type `(0,8)` is *void*.

Prototype (P)

This symbol descriptor specifies that the name is a function or procedure that appears elsewhere in the program. It may only appear 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)

This symbol descriptor specifies that the name is a register variable. It may 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 Global Variable (S)

This symbol descriptor specifies that the name is a static global variable. It may only appear 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)

This symbol descriptor specifies that the symbol is either an enumeration, structure or union tag name. It may only appear 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)

This symbol descriptor specifies that the symbol is a new type name. It may be used in either **N_LSYM** or **N_GSYM** stabs.

Most compilers will generate a "canned" list of standard types. For example, the SunPro 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 ina class.

New stab syntax:

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

Note: 'U' is used for declaration whereas 'T' is used for definition.

Example:

```
class x {
     typedef int myint;
public:
     myint a;
};
```

Old stabs:

```
.stabs "__0dBxFmyint:t(0,24)=(0,3)",N_ISYM,0x0,0x0,0x0
.stabs "x:T(0,25)=Yc4x;;C__0dBxBa:(0,24),0,32;;;;;;;;;",
 N_ESYM,0x0,0x4,0x0
```

New stabs:

```
.stabs "x:U(0,25)",N_ESYM,0x0,0x0,0x0
.stabs "mangled_myint(0,25):t(0,24)=(0,3)",N_ISYM,0x0,0x0,0x0
.stabs "mangled_x:T(0,25)=Yc4x;;C__0dBxBa:(0,24),0,32;;;;;;;;;",
 N_ESYM,0x0,0x4,0x0
```

Common or Static Local Variable (V)

This symbol descriptor describes a static local variable. It may 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.

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)

This symbol descriptor is used to describe a function or subroutine parameter that is passed by reference. It may 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.

Generic Name (W)

This stab may be used with the **N_LSYM** stab for two purposes. First, it can map a Fortran generic name to its possible bindings. 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:W14;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.

The second use for this stab is to specify the linker name of a (Fortran) internal procedure. For example:

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

generates the following stab is additionally generated for `subroutine t`:

```
.stabs "t:W14;t_in_s_ ;;;",N_LSYM,0x0,0x0,0x0
```

Function Result Variable (X)

This symbol describes the function result variable used by Fortran or Pascal. It may only appear in an **N_PSYM** (parameter). 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 that `ifun` is a result variable that is stored at -16 from the frame pointer.

Value Parameter by Reference (x)

This symbol descriptor is use to describe the formal value parameters that are passed to a function or procedure by reference. It may appear in an **N_PSYM** (parameter) stab. The `value` field in the stab is the offset of the address of the parameter from the frame pointer.

The following Pascal procedure

```
type
    str = array [1..20] of char;

procedure q (k : str);
begin
    writeln (k);
end;
```

generate the following stabs:

```
.stabs "q:F(0,8);(0,19)=(0,17)”,N_FUN,0,0,_q
.stabs "k:x(0,17)”,N_PSYM,0x0,0x0,68
```

The first stab is an **N_FUN** (function) stab that describes the function *q*. The second stab is an **N_PSYM** (parameter) stab describes the parameter `k`, which is passed by reference. The address of the parameter is at 68 from the frame pointer.

C++ Specification (Y)

The *Y-stabs* are the C++-specific stabs. 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. Those are described in Appendix D.

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, but for the cases where it does become desirable, dbx has a *demangle* command.

C++ reference types are similar to pointer types — their type numbers are indicated in a similar way, but using `&` instead of `*`. See page 100.

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 C++ creates a set of helper functions, one for each default parameter.

When the user tells dbx to call a function that has default arguments, and does not provide all of the args, dbx will call each of the helper functions, until it has values for all of the arguments that the user omitted. Then dbx will call 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 for that parameter. For instance,

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

might yield a stab like this:

```
82:  .stabs "__0FCfoid:F(0,22);(0,3)__defarg__0___0FCfoid;
        (0,20)__defarg__1___0FCfoid",N_FUN,0x0,0x0,0x0
```

Inline Functions

When Sun C++ is generating dbx information (-g is on), it will choose 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` (gee-zero) 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_LCSYM**s:

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

Example:

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

yields two **N_LCSYM** stabs with the strings:

```
___SI:Yax(0,3)
___SI:Yay(0,14)
```

Note that on SunOS operating environment (pre-ELF object format) systems, the mangled names are not the same as the linker names — the linker names have an extra (fourth!) underscore prepended.

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 which 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 has to be able to print out ("`whatis -t`") the members of a class, there must be a way to indicate *which* anonymous union each union-member is a member of. We use a single-digit code (see below, under "data members" in the "classes" section below).

Stabs for classes, structs, and non-anonymous unions

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

Local classes (defined inside functions) simply show up in the relevant scope, and are **N_LSYM**s instead of **N_GSYM**s. They're 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's virtual, and whether it's 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/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 page 97).

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 `(0,24)` for class `blue`, `(0,25)` for `black`, and `(0,26)` for `bay`, the above class `green` might generate a stab directive like this (broken apart for readability and commentary):

```
42:  .stabs “green:T(0,32)=Yc64green;   // size 64, name green
  C0(0,24)A16(0,25)K20(0,26);           // 3 base classes
  A__0dFgreenBx:(0,3),192,32;           // x, a private int
  Q1__0dFgreenCuf:(0,19),256,32;        // uf in 1st anon union
  Q1__0dFgreenCud:(0,20),256,64;        // ud in 1st anon union
  Q2__0dFgreenDucp:(0,29),320,32;       // ucp in 2nd anon union
  Q2__0dFgreenDuvp:(0,30),320,32;       // uvp in 2nd anon union
  Q3__0dFgreenFstack:(0,2),352,16;      // stack, 3rd anon union
  Q3__0dFgreenEodds:(0,4),352,32;       // odds, 3rd anon union
  ;                                     // second ‘;’ ends data
  I-1__0fFgreenIpurevirtv C__0fFgreenCmfv;// two member functions
  F__0dFgreenIz_static;                 // one static data member
  __0FCsqi @bay __0FEfrfnv @blue;       // a few friends
  1 48;            // virtual function algorithm #1 (“vtable”), and
                  // virtual function table pointer offset 48 bytes.
 A(0,27)A(0,28)A(0,31);      // 3 nested classes (the anon. unions)
  C__0dEblueDsky B__0dEblueEmoon;       // two access adjustments
  56(0,26);”,                           // virt. base offset
        N_LSYM,0x0,0x40,0x0             // the rest of the stab
```

`TemplateMembers` contains the mangled template definition names of all the template members of the class. Only the template defintion names show up in the class stabs..

Example:

This code example shows how a template definition name appears in the class stab of a silly class called `red`:

```
class red {
public:
     int x;
     template < class T > class stack {
         T *v;
     }
    template < class T1, class T2 > int inrange(T1 a, T2 b, int val);
};
// Another inrange
template < class X > int inrange(x a) { ... };
```

```
.stabs "red:T(0,32)=Yc12red;;A__0dfredBx;;;;;;;;
         stack_mangled inrange_mangled;", N_LSYM, 0x0, 0x40, 0x0
```

`inrange_mangled` can point to the correct template `inrange`:

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);
        namespace N3 {
            int i3;
        };
    };
```

The following stabs are generated:

```
.stabs "N1:T(0,25)=Yn0N1;",N_LSYM,0x0,0x0,0x0
.stabs "N2:T(0,27)=Yn0N2;",N_LSYM,0x0,0x0,0x0
.stabs "MANGLED_N3:T(0,28)=Yn0N3;",N_LSYM,0x0,0x0,0x0
```

This `N_LSYM` 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_LSYM` stab must appear before stabs of any of its members. In the example above, the `N_LSYM` 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. Note that the username in the `N_LSYM` stab is the unqualified username.

Members of an unnamed namespace appear as members of the containing namespace. For example:

```
    namespace X1 {
        namespace {
            int noname1;
        };
    };
```

Calling the demangler library with the mangled name of `noname1`, the return would be "`X1::noname1`".

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,0x0,0x4,0xffffffe8
.stabs "z:(0,29)=*(0,22)",N_LSYM,0x0,0x4,0xffffffe4
```

Stabs for x and y convey same thing - pointer to funtion 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 "x:(0,25)=YM(0,24)(0,3)(0,3)#",N_LSYM,0x0,0x8,0xfffffffec

.stabs "y:(0,27)=YM(0,24)(0,3)(0,3)#",N_LSYM,0x0,0x4,0xffffffe8

.stabs "z:(0,29)=YD(0,24)(0,3)",N_LSYM,0x0,0x4,0xffffffe4
```

`(0,24)` is the type ID of class A.

Typedefs within classes (Yy)

If a typedef is defined within a class, it will get a "Yy" stab that precedes the class definition. E.g.,

```
    struct simple {
        typedef double real;
        real member;
    };
```

will yield stabs like these:

```
.stabs "__0dGsimpleEreal:Yy(0,24)=(0,20)",N_LSYM,0x0,0x0,0x0
.stabs "simple:T(0,25)=Ys8simple;;
        C__0dGsimpleGmember:(0,24),0,64;;;;;;;;;",
        N_LSYM,0x0,0x8,0x0
```

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 52. 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
`YTf`	function template
`YIc` or `YIs`	template class or template struct
`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
```

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

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.

For example, given a template definition like:

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

the **`template-parameter-list`** looks like:

```
A:tYC(0,19);x:(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.

Class templates (YTc and YTs)

```
TemplateName:YT cs 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.

`cs` indicates whether this is a class template (`c`) or a 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 (we can't know it yet)

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:

```
template<class T> class vector {
    T* v;
    int sz;
 public:
    vector(int);
    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 "vector:YTc(0,24);T:tYC(0,25);@;0vector;;;;;;;;;",
        N_GSYM,0x0,0x0,0x0
```

Stabs for member functions of class templates (@<)

The compiler generates an index stab for each member function of a class template. Because these entities are not yet instantiated and therefore have no linker symbol associated with them, they must be marked to tell dbx not to look for that linker symbol. That marking consists of the two-character code `@<` at the start of the name.

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.

Template Member Function Instantiations (YIm)

In addiition to the @< index stabs, 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):

```
41:  .stabs "__0fGvector7G_Eelemi:YImT:(0,25);@;m;
             vector:F(0,27);(0,26);(0,3)",N_FUN,0x0,0x0,0x0
42:  .stabs "this:p(0,26)",N_PSYM,0x0,0x0,0x0
43:  .stabs "i:p(0,3)",N_PSYM,0x0,0x0,0x0
44:  .stabn N_LBRAC,0x0,0x1,0x0
45:  .stabn N_RBRAC,0x0,0x1,0x0
```

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 52. 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 "tfex:YTf
        A:tYC(0,24);B:tYC(0,25);  // define template params A and B
        @;;tfex
        :T(0,3);(0,26)=*(0,25);(0,24);  // returns int, gets (B*, A)
        @;
        x:p(0,26);y:p(0,24);      // name function params x and y
        @;
        1;10;",                   // line numbers 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 cs TypeNumber;@;
          ActualArguments;@;g;
          Size TemplateName;
          Bases;DataMembers;MemberFunctions;
          StaticDataMembers;Friends;
          VirtualFunctionInfo;
          NestedClassList;AccessAdjustments;
          VirtualBaseClassOffsets;
          TemplateMembers;
          Passmethod;
```

`ClassName` is the unmangled name of the instantiated class (e.g. “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.

`cs` tells whether the template was declared using `class` or `struct`.

`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 & 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 "__0oGvector7d_ctv:YIc(0,30);
  @;T:(0,20);@;g;8vector;;
  A__0dGvector7d_Bv:(0,31)=*(0,20),0,32;
  A__0dGvector7d_Csz:(0,3),32,32;
  ;
  C__0oGvector7d_cti C__0oGvector7d_vci
   C__0fGvector7d_Eelemi C__0fGvector7d_Esizev
   C__0fGvector7d_Esizei C__0fGvector7d_EdumpPCc;
  ;;;;;;",N_GSYM,0x0,0x0,0x0A
```

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**`;@;;`
NonMangledTemplateName
`: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.

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)`):

```
.stabs "__0FEtfexPci:YIfA:(0,3);B:(0,1);@;g;
```

```
        tfex:F(0,3);(0,27)=*(0,1);(0,3)",N_FUN,0x0,0x0,0x10860
.stabs "x:p(0,27)",N_PSYM,0x0,0x4,0x44
.stabs "y:p(0,3)",N_PSYM,0x0,0x4,0x48
.stabn N_LBRAC,0x0,0x1,0xc
.stabn N_SLINE,0x0,0x3,0xc
.stabn N_LBRAC,0x0,0x2,0x2c
```

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 "type2throw:T(0,24)=Ys4type2throw;;
    C__0dKtype2throwDmsg:(0,25)=*(0,1),0,32;
    ;;;;;;;;",N_LSYM,0x0,0x4,0x0
.stabs "__rtbi:YR(0,3)",N_LSYM,0x0,0x0,0x0
.stabs "__rtu__6Ktype2throw:YR(0,24)",N_LSYM,0x0,0x0,0x0
.stabs "__0FI__rtbi_fv:P(0,26)=*(0,27)=xs__RTTI:",
      N_FUN,0x0,0x0,0x0
```

The mangled name `__rtbi` indicates a builtin type (int), and the longer name `__rtu__6Ktype2throw` indicates the user-defined type in the second `throw`. The final stab shown is the stab for the function prototype

```
struct __RTTI *__rtbi_f();
```

which is the initialization function for the `__rtbi` structure.

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

Ellipses: Variadic (i.e., *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(0,17)=bs0;0;0",N_LSYM,0,0,0
.stabs "...:t(0,18)=bs4;0;32",N_LSYM,0,0,0
```

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

```
.stabs "__0FDetcPCce:F(0,22);        // return type
            (0,24)=*(0,25)=k(0,1);   // ptr to const char *
            (0,23)",                 // ellipsis "type"
                N_FUN,0x0,0x0,0x0
.stabs "fmt:p(0,24)",N_PSYM,0x0,0x4,0x44
.stabs "__builtin_va_alist:p(0,23)",N_PSYM,0x0,0x4,0x48
```

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

Type Specification 6

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 1 on page 86.

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. This number represents the type within the source file.

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 Pascal program fragment:

```
type x : set of char;
var y : set of char;
var z : x;
```

This generates (in part) the following stabs:

```
.stabs "char:t(0,4)=buc1;0;8;",N_LSYM,0x0,0x0,0
.stabs "x:t(0,17)=S(0,4)",N_LSYM,0x0,0x0,0
.stabs "y:GS(0,4)",N_GSYM,0x0,0x0,0
.stabs "z:G(0,17)",N_GSYM,0x0,0x0,0
```

The first line is a description of the `char` type. Symbol descriptor `t` indicates that `char` is a type, which is given type number `(0,4)` and defined (as indicated by the equal sign followed by a type specification) as a basic integer.

The second line describes `x`, also a type, which is given type number (0,17). This is defined as a set of type (0,4), `char`. This is the same format as the first line.

The third line describes `y`, a global variable (as indicated by the symbol descriptor `G`). This is followed by a type description for a set of `char`. In this case, no new type number is defined for the type description.

The last line describes `z`, also a global variable, which is described as type `(0,17)`, which was defined in line two.

The symbol descriptors are described in Chapter 5, “Symbol Descriptors.”

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
C	Conformant array bounds
c	Conformant array
d	Dope vector

`e`	Enumeration
`F`	Function parameter
`f`	Function
`g`	Function with prototype info
`K`	Restricted
`k`	Const
`L`	Pascal file
`P`	Procedure parameter
`p`	Value parameter
`q`	Reference parameter
`R`	Floating point
`r`	Range
`S`	Set
`s`	Structure or record
`u`	Union
`v`	Variant record (Pascal)
`x`	Forward reference
`Y`	C++ specification (C++ 4.0 and later, see page 61)
`y`	Varying string (Pascal)
`*`	Pointer
`&`	Reference

Each of these types is 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

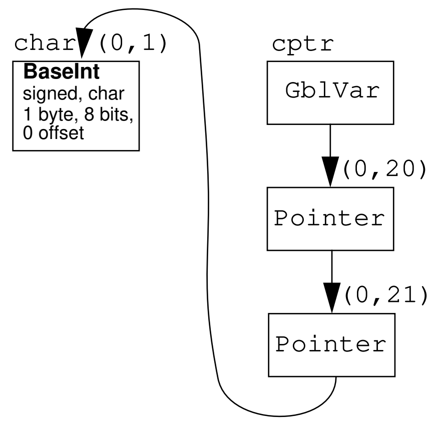
```
char **cptr;
```

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 1.

Figure 1 Example Type Tree.



Array (a)

a IndexType ; Type

This 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 ar[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 "ar: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)`.

Volatile (B)

`B` ***`Type`***

Types which are `volatile` (or `const`, or `const volatile`) generate stabs which are similar to pointer stabs (see page 99). 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`***

This 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
```

Conformant Array Bounds (C)

C [*Low_bound*] , [*High_bound*] , *Type*

This type describes the range specification for a Pascal-conformant array. *Low_bound* and *high_bound*, if present, are the names of the variables that contain the low and high bound values. *Type* is the type of these variables.

This always appears as part of a conformant array type specification.

Conformant Array (c)

c *Bounds* ; *Type*

This type describes a Pascal-conformant array. *Bounds* is a conformant array bounds specification. *Type* is the type of the array.

The following procedure

```
procedure proc2(a : array[x..y : integer] of integer);
var j,k : integer;
begin
    k := 0;
    for j := x to y
        do k := k + a[j];
end;
```

generates (in part) the following stabs:

```
.stabs   "proc2:F(0,8);(0,17)=cC,,(0,5);(0,5);(0,5);(0,5)",
            N_FUN,0,0,_proc2
.stabs   "x:C(0,5)",N_PSYM,0x0,0x0,72
.stabs   "y:C(0,5)",N_PSYM,0x0,0x0,76
.stabs   "a:xcCx,y,(0,5);(0,5)",N_PSYM,0x0,0x0,68
```

The first stab describes the procedure as a global function that returns *void* (type `(0,8)`), and which takes a conformant array argument (given type number `(0,17)`), and two integer (type `(0,5)`) arguments (for the array bounds).

The second and third stabs describe the array bound variables as conformant array bounds, which are integers.

The last `N_PSYM` stab describes the conformant array value parameter `a` (symbol descriptor `x`), which has conformant array type (`c`). This type has a conformant array bounds that specifies variables `x` and `y`, and also specifies that the array values are integers.

Dope Vector (d)

`d` ***`Type`***

This specifier describes a Fortran90 allocatable array type, i.e., an array which must be accessed via a *dope vector*, a pointer, or an anonymous reference to an array valued object. This specifier may only be applied 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 statements:

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

generate 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 “Fortran90 Dope Vectors” on page 137.

Enumeration (e)

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

This 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***

This type describes a function passed as a parameter to a Pascal procedure or function. *Type* specifies the return value of the function.

The following Pascal program

```
program test(input,output);
function q ( a : integer ) : integer ;
begin
     q := a * 2 ;
end;

procedure p ( function f ( i : integer ) : integer ) ;
begin
     writeln (f(2));
end;

begin
     p(q);
end.
```

generates (in part) the following stabs:

```
.stabs "p:F(0,8);(0,17)=F(0,5)",N_FUN,0,0,_p
.stabs "f:pF(0,5)",N_PSYM,0x0,0x0,68
```

The first stab describes global procedure `p` (symbol descriptor `F`) and indicates that it takes a single argument (assigned type `(0,17)`), which is a function (type `F`).

The second stab describes the parameter `f` (symbol descriptor `p`), which is a function (type `F`) returning an integer.

Function (f)

`f` *`Type`*

This 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]#`

This 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. More examples:

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

```
.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`***

This 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 page 99). 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)",...
```

Pascal File (L)

`L` ***`Type`***

This type describes a Pascal file type by giving its base type.

The Pascal statement

```
var f: file of char;
```

generates the following stab:

```
.stabs  "f:GL(0,3)",N_GSYM,0x0,0x1,0
```

This N_GYSM (global) stab describes the global variable `f` (symbol descriptor `G`), which is a file (type `L`) of characters (type `(0,3)`).

Procedure Parameter (P)

`P`

This type describes a procedure passed as a parameter to a Pascal procedure or function.

The following Pascal program

```
program g (input, output);

procedure q (k : integer);
begin
    writeln (k);
end;

procedure p ( procedure x ( i : integer ) );
begin
    x ( 4 );
```

```
end;

begin
    p ( q );
end.
```

generates the following stabs for the function *p*:

```
.stabs "p:F(0,8);(0,17)=P(0,8)",N_FUN,0,0,_p
.stabs "x:pP(0,8)",N_PSYM,0x0,0x0,68
```

The first stab describes global procedure *p* (symbol descriptor `F`) and indicates that it takes one argument (assigned type `(0,17)`), which is a function (type `P`).

The second stab describes the parameter `x` (symbol descriptor `p`), which is a procedure (type `P`).

Floating Point (R)

`R` *`Format`* `;` *`Nbytes`*

This 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
`NF_COMPLEX16`	4	Fortran double complex
`NF_COMPLEX32`	5	Fortran quad complex
`NF_LDOUBLE`	6	Long double (128-bit float)

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`*

This 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`). 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 (e.g. 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 86.

In the following Pascal program

program g (input, output);

```
var
    j : 5..50;

begin
    j := 9;
end.
```

generates the following stab for the variable `j`:

```
.stabs "j:G(0,17)=r(0,18)=bs1;0;8;5;50",N_GSYM,0x0,0x0,0
```

This stab describes `j` as a global symbol with type `(0,17)`. This is defined as a range based on type `(0,18)` from 5 to 50. Type `(0,18)` is described as a one-byte signed integer.

Set (S)

`S` ***`Type`***

This 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 (and for records, the variants) defined in the record or union. It has the following format:

FieldName **:** { ***Type*** | ***Variant*** } **,** ***BitOffset*** **,** ***BitSize*** **;** }+

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

A *Variant* description has the following format:

***Selector* ;** { ***Case* ;** }**+ ;**

A *Variant* consists of the *Selector* name and a series of *Cases* separated by semicolons. The *Selector* has the following format:

v *Flag* : *Type*

The *Flag* in *Selector* indicates whether a variant record has a discriminant (1) or not (0); the discriminant is used in Pascal to determine the active variant.

Each *Case* specifies the selection value for each case, *CaseValue*, and is followed by a list of fields as described above. It has the following format:

{ ***CaseValue*** **,** }+ **:** ***FieldList***

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.

The Pascal declaration

```
var y: record case k:integer of
  0: ( a: integer; );
  1: ( b: real; );
end;
```

generates (in part) the following stabs;

```
.stabs "y:Gs16k:(0,5),0,32;\\",N_GSYM,0x0,0x0,0
.stabs "$TAG:v1:(0,5);0:a:(0,5),64,32;;1:\\",N_GSYM,0x0,0x0,0
.stabs "b:(0,7),64,64;;;,64,96;;",N_GSYM,0x0,0x0,0
```

These three *.stabs* directives describe the record and represent a single continued description. Note that the backslash that ends the string in the first two stabs describing the record is doubled. The third line defines `y` as a global variable, which is a structure 16 bytes long. The first field is `k`, which is an `integer` (type `(0,5)`) which occupies bits 0 through 32. The second field is `$TAG`, which identifies the previous field as a variant descriptor. This is followed by case 0, which defines field `a` to be an integer in bits 64 to 96, and case 1, which defines field `b` to be a real in bits 64 through 128. The final 64 and 96 represent the bit offset and bit length, respectively, of the variant part.

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 61.

Varying String (y)

y ; Length

This type describes a Pascal varying string type by giving its maximum length. For example, the Pascal statement

```
type aaa = varying[204] of char;
```

generates the following stab description:

```
.stabs "aaa:t(0,17)=y;204",N_LSYM,0x0,0x0,0
```

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
```

Auto-load Stab Processing 7

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 -g. 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 -g, 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 104.

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 is 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.

Stabs Generation 8

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 most of the debugging stabs. `N_LSYM` or `N_GSYM` stabs should be generated for all variables and types and `N_PSYM` stabs for function arguments.

Only one `N_SLINE` stab should be generated for each function. In an a.out file, the `value` field of the `N_SLINE` stab should have the address of the first executable instruction in the function. In an ELF file, the `value` field should be zero, since the address of the function is obtained from the symbol table. A single `N_LBRAC/N_RBRAC` pair should be generated to bracket each function.

Stab Codes

Table 1 Numerical Index

Stab Name	Stab Code	Described on Page	Description Model
N_UNDF	0x00	40	Start of object file `.stabs "Filename.o", N_UNDF, 0, NumberOfStabs, BytesOfStringTable`
N_EXT	0x01	22	External symbol flag (a.out only)
N_ABS	0x02	13	Absolute symbol (a.out only)
N_TEXT	0x04	39	Text segment symbol (a.out only)
N_DATA	0x06	19	Data segment symbol (a.out only)
N_BSS	0x08	17	BSS segment symbol (a.out only)
N_COMM	0x12	18	Common symbol (a.out internal only)
N_FN	0x1f	23	File name (a.out only). (Not used by dbx)
N_GSYM	0x20	24	Global symbol `.stabs "Name : SymDesc Type", N_GSYM, 0, Desc, Value`
N_FUN	0x24	23	Function or procedure `.stabs "Name : SymDesc RtnType [ ; ArgType ]*", N_FUN, 0, RtnSize, 0`

Table 1 Numerical Index *(Continued)*

Stab Name	Stab Code	Described on Page	Description Model
N_STSYM	0x26	38	Initialized static symbol `.stabs "Name : SymDesc Type", N_STSYM, 0, 0, Offset`
N_LCSYM	0x28	27	Unitialized static symbol `.stabs "Name : SymDesc Type", N_LCSYM, 0, 0, Offset`
N_MAIN	0x2a	29	Name of main routine `.stabs "Name", N_MAIN, 0, 0, 0`
N_ROSYM	0x2c	33	Read-only static symbol `.stabs "Name : SymDesc Type", N_ROSYM, 0, 0, Offset`
N_PC	0x30	32	Global pascal symbol `.stabs "Ident", N_PC, 0, 0, 0`
N_CMDLINE	0x34	17	`.stabs "cd path;cc..etc", 0, 0, 0`

N_OBJ	0x38	29	Object file or path name
			`.stabs "ObjectDir", N_OBJ, 0, 0, 0`
			`.stabs "ObjectFile", N_OBJ, 0, 0, 0`

N_OPT	0x3c	29	Compiler options `.stabs "Options", N_OPT, 0, 0, TimeStamp`
N_RSYM	0x40	34	Register symbol `.stabs "Name : SymDesc Type", N_RSYM, 0, Size, Number`
N_SLINE	0x44	35	Source line `.stabn N_SLINE, 0, Line, Offset`
N_XLINE	0x45	44	Source line `.stabn N_XLINE, 0, Hi16bitsLineMask, 0`
N_BROWS	0x48	16	Path to associated *.cb* file `.stabs "bdfile", N_BROWS, 0, 0, 0` (Not used by dbx)

Table 1 Numerical Index *(Continued)*

Stab Name	Stab Code	Described on Page	Description Model
N_ILDPAD	0x4c	*25*	Pad string table (used by linker, not dbx) `.stabs "Objname", N_ILDPAD,0,0, Len` (Formerly N_FLINE)
N_ENDM	0x62	20	End module `.stabn N_ENDM, 0, 0, 0`

N_SO	0x64	36	Compilation source file or path name
			`.stabs "SourceDir", N_SO, 0, 0, 0`
			`.stabs "SourceFile", N_SO, 0, LangCode, 0`

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	28	Local symbol `.stabs "Name : SymDesc Type", N_LSYM, 0, Desc, Value`
N_BINCL	0x82	15	Begin include file `.stabs "FilePath", N_BINCL, 0, 0, HashValue`
N_SOL	0x84	37	Included or referenced source file `.stab "FilePath", N_SOL, 0, 0, 0`
N_PSYM	0xa0	32	Formal parameter `.stabs "Name : SymDesc Type", N_PSYM, 0, 0, Offset`
N_EINCL	0xa2	19	End of include file `.stabn N_EINCL, 0, 0, 0`
N_ENTRY	0xa4	20	Fortram alternate entry `.stabs "Name : e RtnType ; FunName ; ;", N_ENTRY, 0, Line, 0`

Table 1 Numerical Index *(Continued)*

Stab Name	Stab Code	Described on Page	Description Model
N_LBRAC	0xc0	25	Start of scope (left bracket) **.stabn N_LBRAC, 0, *Level*, *Offset***
N_EXCL	0xc2	21	Excluded include file **.stabs "*FilePath*", 0, 0, *HashValue***
N_USING	0xc4	40	**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_ISYM	0xc6	25	**Position independent type, internal** **.stabs "*Name*:*SymDesc Type*", N_ISYM, 0, *Desc*, *Value***
N_ESYM	0xc8	21	**Position independent type, external** **.stabs "*Name*:*SymDesc Type*", N_ESYM, 0, *Desc*, *Value***
N_PATCH	0xd0	30	**.stabn N_PATCH, 0, 0, *Addr***
N_CONSTRUCT	0xd2	18	Constructor description **.stabs "Var:State",N_CONSTRUCT, 0,End-Start,Start-Func**
N_DESTRUCT	0xd4	19	Destructor description **.stabs "FromState:ToState",N_DESTRUCT, 0,End-Start,Start-Func**
N_RBRAC	0xe0	33	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	39	Begin task common block **.stabs "*BlockName*", N_TCOMM, 0, 0, *Offset***

Table 1 Numerical Index *(Continued)*

Stab Name	Stab Code	Described on Page	Description Model
N_ECOMM	0xe4	19	End common block .stabs "*BlockName*", N_ECOMM, 0,0,0
N_XCOMM	0xe6	43	Excluded common block .stabs "*BlockName*", N_XCOMM, 0, 0, *HashValue*
N_WITH	0xea	40	Pascal WITH statement .stabs "*Name* : *Type*", N_WITH ,0, *Offset*

Table 2 Alphabetical Index

Stab Name	Stab Code	Described on Page	Description Model
N_ABS	0x02	13	Absolute symbol (a.out only)
N_ALIAS	0x6c	13	.stabs "F*Newname*:*Oldname*", N_ALIAS, 0, 0, 0 .stabs "V*Newname*:*Oldname*", N_ALIAS, 0, 0, 0 .stabs "*externalname*:R*sourcename*", 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	15	Begin include file .stabs "*FilePath*", N_BINCL, 0, 0, *HashValue*
N_BROWS	0x48	16	Path to associated *.cb* file .stabs "*bdfile*", N_BROWS, 0, 0, 0
N_BSS	0x08	17	BSS segment symbol (a.out only)
N_CMDLINE	0x34	17	.stabs "cd *path*;*cc..etc*", 0, 0, 0
N_COMM	0x12	18	Common symbol (a.out internal only)

Table 2 Alphabetical Index *(Continued)*

Stab Name	Stab Code	Described on Page	Description Model
N_CONSTRUCT	0xd2	18	Constructor description `.stabs "Var:State",N_CONSTRUCT, 0,End-Start,Start-Func`
N_DATA	0x06	19	Data segment symbol (a.out only)
N_DESTRUCT	0xd4	19	Destructor description `.stabs "FromState:ToState",N_DESTRUCT, 0,End-Start,Start-Func`
N_ECOMM	0xe4	19	End common block `.stabs "BlockName", N_ECOMM, 0,0,0`
N_EINCL	0xa2	19	End of include file `.stabn N_EINCL, 0, 0, 0`
N_ENDM	0x62	20	End module `.stabn N_ENDM, 0, 0, 0`
N_ENTRY	0xa4	20	Fortran alternate entry `.stabs "Name : e RtnType ; FunName ; ;", N_ENTRY, 0, Line, 0`
N_ESYM	0xc8	21	Position independent type, external `.stabs "Name:SymDesc Type", N_ESYM, 0, Desc, Value`
N_EXCL	0xc2	21	Excluded include file `.stabs "FilePath", 0, 0, HashValue`
N_EXT	0x01	22	External symbol flag (a.out only)
N_FN	0x1f	23	File name (a.out only)
N_FUN	0x24	23	Function or procedure `.stabs "Name : SymDesc RtnType [ ; ArgType ]*", N_FUN, 0, RtnSize, 0`
N_GSYM	0x20	24	Global symbol .stabs "*Name* : *SymDesc Type*", N_GSYM, 0, *Desc*, *Value*

Table 2 Alphabetical Index *(Continued)*

Stab Name	Stab Code	Described on Page	Description Model
N_ILDPAD	0x4c	25	Pad string table (used by linker, not dbx) .stabs "Objname", N_ILDPAD,0,0, Len (Formerly N_FLINE
N_ISYM	0xc6	25	Position independent type, internal `.stabs "Name:SymDesc Type", N_ISYM, 0, Desc, Value`
N_LBRAC	0xc0	25	Start of scope (left bracket) `.stabn N_LBRAC, 0, Level, Offset`
N_LCSYM	0x28	27	Unitialized static symbol `.stabs "Name : SymDesc Type", N_LCSYM, 0, 0, Offset`
N_LSYM	0x80	28	Local symbol `.stabs "Name : SymDesc Type", N_LSYM, 0, Desc, Value`
N_MAIN	0x2a	29	Name of main routine `.stabs "Name", N_MAIN, 0, 0, 0`

N_OBJ	0x38	29	Object file or path name
			`.stabs "ObjectDir", N_OBJ, 0, 0, 0`
			`.stabs "ObjectFile", N_OBJ, 0, 0, 0`

N_OPT	0x3c	29	Compiler options `.stabs "Options", N_OPT, 0, 0, TimeStamp`
N_PATCH	0xd0	30	`.stabn N_PATCH, 0, 0, Addr`
N_PC	0x30	32	Global pascal symbol `.stabs "Ident", N_PC, 0, 0, 0`
N_PSYM	0xa0	32	Formal parameter `.stabs "Name : SymDesc Type", N_PSYM, 0, 0, Offset`
N_RBRAC	0xe0	33	End of scope (right bracket) `.stabn N_RBRAC, 0, Level, Offset`
N_ROSYM	0x2c	33	Read-only static symbol `.stabs "Name : SymDesc Type", N_ROSYM, 0, 0, Offset`

Table 2 Alphabetical Index *(Continued)*

Stab Name	Stab Code	Described on Page	Description Model
N_RSYM	0x40	34	Register symbol **.stabs "*Name : SymDesc Type*",** **N_RSYM, 0, *Size*, *Number***
N_SLINE	0x44	35	Source line **.stabn N_SLINE, 0, *Line*, *Offset***
N_SO	0x64	36	Compilation source file or path name **.stabs "*SourceDir*", N_SO, 0, 0, 0** **.stabs "*SourceFile*", N_SO, 0,** ***LangCode*, 0**
N_SOL	0x84	37	Included or referenced source file **.stab "*FilePath*", N_SOL, 0, 0, 0**
N_STSYM	0x26	38	Initialized static symbol **.stabs "*Name : SymDesc Type*",** **N_STSYM, 0, 0, *Offset***
N_TCOMM	0xe3	39	Begin task common block **.stabs "*BlockName*", N_TCOMM, 0, 0,** ***Offset***
N_TEXT	0x04	39	Text segment symbol (a.out only)
N_UNDF	0x00	40	Undefined symbol
N_USING	0xc4	40	**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 2 Alphabetical Index *(Continued)*

Stab Name	Stab Code	Described on Page	Description Model
N_WITH	0xea	40	Pascal WITH statement `.stabs "Name : Type", N_WITH ,0, Offset`
N_XCOMM	0xe6	43	Excluded common block `.stabs "BlockName", N_XCOMM, 0, 0, HashValue`
N_XLINE	0x45	44	Source line `.stabn N_XLINE, 0, Hi16bitsLineMask, 0`

Symbol Descriptors B

Table 1 Symbol descriptors — Alphabetical by character code

Code	Description
empty	Local variable, page 46
A	Automatic variable (Fortran90), page 47
b	Based variable, page 47
C	Conformant array bound, page 48
c	Constant symbol (Fortran, pascal), page 49
F	Global function or procedure, page 49
f	Local function or procedure, page 50
G	Global variable, page 50
H	Hosted, non-local variable, page 50
J	Internal procedure (Fortran90), page 51
LT	C++ 4.0 Lines in template, page 52
l	Literal, page 52
M	Module (Fortran90), page 52
P	Prototype, page 54
p	Value parameter, page 53

Table 1 Symbol descriptors — Alphabetical by character code *(Continued)*

Code	Description
r	Register variable, page 54
S	Static global variable, page 55
T	Enumeration, structure or union, page 56
t	Type name, page 56
U	Class declaration, page 57
V	Common or static local variable, page 57
v	Variable parameter by reference, page 59
W	Generic Name (Fortran90), page 59
X	Function result variable, page 60
x	Value parameter by reference, page 61
Y	C++ 4.0 specification (see below), page 61
Z	C++ 3.0 (cfront) specification, page 125

Table 2 C++ 5.0 Specification

Code characters	Description
YA	Anonymous union, nested, page 63, page 65
Ya	Anonymous union, page 63
YC	Class nested, page 65
Yc	Class, page 64
YD	Pointer to class data member, page 70
YI	Template Instantiation, page 73
YM	Pointer to class member function, page 70
Yo	ObjectiveC Class, page 64
YR	Run Time Type Information (RTTI), page 81
YS	Structure nested, page 65
Ys	Structure, page 64
YT	Template definition, page 73

Table 2 C++ 5.0 Specification *(Continued)*

Code characters	Description
YU	Union, nested, page 65
Yu	Union, page 64
Yy	Typedef within class, page 72

Type Codes C

Table 3 Type codes — Alphabetical by character code

Code character	Description
empty	Type reference, page 83
a	Array, page 86
B	Volatile, page 87
b	Basic integer, page 87
C	Conformant array bounds, page 88
c	Conformant array, page 88
d	Dope vector, page 89
e	Enumeration, page 90
F	Function parameter, page 90
f	Function, page 91
g	Function with prototype info, page 92
K	Restricted, page 93
k	Const, page 93
L	Pascal file, page 94
P	Procedure parameter, page 94

Table 3 Type codes — Alphabetical by character code *(Continued)*

Code character	Description
R	Floating point, page 95
r	Range, page 95
S	Set, page 96
s	Structure or record, page 97
u	Union, page 97
x	Forward reference, page 98
Y	C++ specification, page 99
y	Varying string, page 99
*	Pointer, page 99
&	Reference, page 100

Old Symbol Descriptors D

Old C++ specification (Z)

This is the specification of stabs for Sun C++ 3.0, the Cfront-based C++ compiler. The Z format stabs have been superceded by the expanded and improved Y format stabs. Dbx continues to support the Z format, however, it is expected that the Z format will eventually be phased out.

Sun C++ 3.0 generates a "C-struct" stab and a "Zc" class stab for each class, struct, or union. (The "Zc" stabs can actually be any of Zc, Zs, Zu, or Za, for class, struct, union, and anonymous union; and an extra "Zl stab" is used for any (function-)local class.)

The Z symbol descriptor specifies that the symbol is a C++ object. It may appear in either `N_LSYM` (local) or `N_GSYM` (global) symbols.

The Z is followed by a subclass identification, which may be one of the following:

- `a` — anonymous union
- `c` — class
- `l` — function local class/struct/union
- `M` — main function
- `n` — constant
- `P` — template instantiation
- `r` — reference variable

s — structure
T — template definition
u — union

There will frequently be more than one stab that describes a C++ object. One will be generated by the C++ translator and one or more will be generated by the C compiler or assembler. The C++ Z stab may appear before or after the related C stabs. The symbol names in the stabs generated by the C compiler or assembler will use the mangled names that are created from the C++ source name. The information provided by these stabs must be combined to fully describe the C++ object.

C++ anonymous union, class, structure, union (Za, Zc, Zs, Zu)

The data members of an anonymous union, class, structure or union are found from the stab generated by the C compiler by demangling the member names. The Z stabs contain additional information about the object and have the following format:

Name **:Z** [**c** | **s** | **u** | **a**] ***Tag*****;** ***Bases*****;** ***Mem_funs*****;**
Stat_mems**;** ***Friends*****;**

Name is the name of the class, structure, or union. For an anonymous union, this is the name generated by the compiler for the union.

Tag is the class, structure or union tag given to the object. It is usually identical to *Name*.

Bases is a list of base class names separated by spaces. Each of these class names is preceded by an optional `v`, which indicates that the class is virtual and a required `1`, `2`, or `3`, which indicates that the class is `public`, `private`, or `protected` respectively. These codes are separated from the class name by an *at* (@) sign.

Mem_funs is a list of the mangled names of the member functions in this object, separated by spaces.

Stat_mems is a list of the mangled names of the static members in this object, separated by spaces.

Friends is a list of the mangled names of friend classes or friend functions in this object, separated by spaces. Friend classes have an *at* sign before the class name.

The program fragment

```
class foo {
        int a;
};
class bar {
        int b;
};
class A : private foo, public virtual bar {
        static int x;
public :
        A (int) {}
        static int getx () { return x; }
        A () {}
        friend class B;
        friend void scan ();
};
```

generates (in part) the following stabs:

```
.stabs "foo:T(0,23)=s4a__3foo:(0,3),0,32;;",N_LSYM,0,4,0
.stabs "foo:Zcfoo;;;;;",N_LSYM,0,0,0
.stabs "bar:T(0,24)=s4b__3bar:(0,3),0,32;;",N_LSYM,0,4,0
.stabs "bar:Zcbar;;;;;",N_LSYM,0,0,0
.stabs  "A:T(0,25)=s12a__3foo:(0,3),0,32;
.stabs Pbar:(0,26)=*(0,24),32,32;Obar:(0,24),64,32;;",
    N_LSYM,0,12,0
.stabs "A:ZcA;2@foo v1@bar;__ct__1AFi __ct__1AFv
    ~__ct__1AFRC1A *getx__1ASFv ;x__1A ;scan__Fv @B ;",
    N_LSYM,0,0,0
```

The first stab is generated by the C compiler and describes `struct foo`, which was generated by the C++ translator from `class foo`. The second stab is generated by the C++ translator and describes `foo` as a class with tag name `foo`. The third and fourth stabs similarly describe `class bar`.

The fifth stab is generated by the C compiler and it describes `struct A`, generated by the C++ translator from `class A`. Each of the member names listed in the structure description is mangled according to C++ conventions.

The final stab is generated by the C++ translator and describes the class A. There are two base classes described, `foo` and `bar`. `Foo` is specified as a `private` class (by the 2 before the *at* sign) and `bar` is specified as a `virtual public` class (by the v1 before the *at* sign). The member functions list gives the mangled names for four member functions (three represent functions declared in the class, one is the destructor function for the class). The static data member list contains one mangled name for the static data member `x`. The friends section contains the mangled name of the friend function and the friend class declared in the class. The friend class name is preceded by an *at* sign.

C++ local class/struct/union (Zl)

This stab gives additional information for a class, structure or union that is defined within a function. It has the following format:

Mangled_name* : Zl ; *Real_name* ; *Function

Mangled_name is the name of the object mangled according to C++ conventions.

Real_name is the name of the object as it appeared in the source.

Function is the mangled name of the function in which the object is defined.

The function and class definition

```
void foo ()
{
   class A {
        int x;
   } y;
}
```

generates the following stabs:

```
.stabs "A__foo__Fv__L1:T(0,23)=s4x__14A__foo__Fv__L1:
    (0,3),0,32;;",N_LSYM,0,4,0
.stabs "A__foo__Fv__L1:Zl;A;foo__Fv ;",N_LSYM,0,0,0
.stabs "A:ZcA__foo__Fv__L1;;;;;",N_LSYM,0,0,0
```

The first stab is generated by the C compiler and describes the structure that the C++ translator generated for `class A`. It specifies the mangled name generated by the translator.

The second stab specifies that the real name for the class is `A` and that it is contained in the function *foo__Fv,* which is the mangled name for function *foo*.

The last stab is the C++ class description for `class A`, which specifies the mangled name of the tag that was generated by the C++ translator.

C++ main function (ZM)

This stab specifies the line where a call to `_main` is used to initialize static constructors and destructors. It can only appear in an `N_LSYM` (local symbol) stab. It is only used on SunOS 4.x.

The following C++ code

```
class A {
  public:
    int x;
    A() { x = 5; }
} z;

main ()
{
   class A y;
}
```

generates the following `ZM` stab:

```
.stabs ":ZM 8",N_LSYM,0x0,0x0,0x0
```

This stab indicates that the *main* routine starts at line 8 in the current source file. The starting line is the line that contains the left brace starting the function.

C++ constant (Zn)

The symbol name specified is a C++ constant.

This program

```
const int a = 5;
main ()
{
  const int b = 9;
}
```

generates (in part) the following stabs:

```
.stabs "a:S(0,3)",N_STSYM,0,4,_a
.stabs "a :Zn",N_LSYM,0,0,0
.stabs "__1b:(0,3)",N_LSYM,0,4,-4
.stabs "__1b :Zn",N_LSYM,0,0,0
```

The first `N_STSYM` (initialized static) stab describes `a` as a static integer variable.

The second stab indicates that it is a C++ constant.

The third `N_LSYM` (local symbol) stab similarly describes a local variable with a mangled name of `__1b`.

The final stab indicates that this is also a constant.

C++ template instantiation(ZP)

There are four variations on this symbol description that are specified by the characters following the `P`. These are

- `c` — Class instantiation
- `s` — Struct instantiation
- `u` — Union instantiation
- `tF` — Function instantiation

For structure, class or union template instantiations, the description is the same as a `Zc` or `Zs` symbol type except the tag is followed by an *at* sign and the name of the template.

For function template instantiations, the name of the template follows the symbol type.

When a class or struct template is instantiated (see example for templates below) as in the following code:

```
stack<int> S(100);
```

generates the following stabs:

```
    .stabs "stack<int >:ZPsstack__pt__2_i@stack;1@Z
        v2@X;__ct__14stack__pt__2_iFi
        ~__ct__14stack__pt__2_iFRC14stack__pt__2_i
        __dt__14stack__pt__2_iFv push__14stack__pt__2_iFi
        pop__14stack__pt__2_iFv \",N_LSYM,0x0,0x0,0x0
    .stabs "size__14stack__pt__2_iCFv ;cnt__14stack__pt__2_i ;@B
;",
        N_LSYM,0x0,0x0,0x0
```

The symbol descriptor code is `ZPs` in this stab, since this is an instantiation of a *struct*. If it were the instantiation of a *class* or *union*, the symbol descriptor would be `ZPc` or `ZPu`, respectively.

The tag assigned to the structure is followed by an "at" sign (@) and the name of the template, `stack`. Following this is the description of the structure, exactly as described above for C++ structures (symbol descriptor Zs).

C++ reference variable (Zr)

The symbol specified is a C++ reference variable. A stab is also generated with the mangled name of the variable indicating that it is a pointer.

The following C++ code

```
int x;
int &a = x;
```

generates (in part) the following stabs:

```
.stabs "x:G(0,3)",N_GSYM,0,4,0
.stabs "a:G(0,23)=*(0,3)",N_GSYM,0,4,0
.stabs "a :Zr",N_GSYM,0,0,0
```

The first `N_GSYM` stab describes `x` as a global integer (type `(0,3)`) variable. The second describes `a` as a global that has type `(0,23)`, which is defined as a pointer to integer. The third stab specifies that `a` is really a C++ reference, and not a true pointer variable.

C++ template definition (ZT)

The symbol specified is a C++ class or function template name.

C++ class template definition (ZTc)

The class template has the following format:

```
Name : ZTc Decl ; [v] {1|2|3} Bases ; {1|2|3} Mem_funs ;
       {1|2|3} Mem_data ; Stat_mems ; Friends
```

Name is the name of the template.

Decl is the source text that declares the template. This is the all of the source text following the keyword `template` up to the opening left brace of the declaration, with comments and extra spaces removed.

Bases is a list of source text strings that specify base class names. These strings are separated by spaces. Each of these text strings is preceded by an optional `v` (which indicates that the class is virtual) and a required `1`, `2`, or `3`, which indicates whether the class is `public`, `private`, or `protected`, respectively.

Mem_funs is a list of the source text strings that declare member functions in the template. These strings are separated by spaces. Each of these text strings is preceded by a `1`, `2`, or `3`, which indicates whether the class is `public`, `private`, or `protected`, respectively.

Mem_data is a list of the source text strings that declare member variables in the template. These strings are separated by spaces. Each of these text strings is preceded by a `1`, `2`, or `3`, which indicates whether the class is `public`, `private`, or `protected`, respectively.

Stat_mems is a list of the source text strings that declare the static data members in this object. These strings are separated by spaces. Each of these text strings is preceded by a `1`, `2`, or `3`, which indicates whether the class is `public`, `private`, or `protected`, respectively.

Friends is a list of the source text strings that declare the friend classes or friend functions in this object. These strings are separated by spaces.

Each of the text strings in a template description is delimited with *backslash-quote* (**\"**) sequences. This means that both the backslash and the quote must be escaped with a slash if the *.stabs* assembler directive is used.

The following C++ declarations

```
class X {
   int x;
};

class Z {
   float z;
};

template <class T> struct stack : virtual private X, public Z {
   T *v;
   T* p;
   int sz;
   static int cnt;
public:
   stack (int s) { v = p = new T[ sz = s]; };
   ~stack () { delete[] v; }

   void push (T a) { *p++ = a; }
   virtual T pop () { return *--p; }

   int size () const { return p - v; }
   friend T top (stack<T>&);
   friend class B;
};
```

generates (in part) the following stabs:

```
.stabs "X:T(0,23)=s4x__1X:(0,3),0,32;;",N_LSYM,0x0,0x4,0x0
.stabs "X:ZcX;;;;;",N_LSYM,0x0,0x0,0x0
.stabs "Z:T(0,24)=s4z__1Z:(0,16),0,32;;",N_LSYM,0x0,0x4,0x0
.stabs "Z:ZcZ;;;;;",N_LSYM,0x0,0x0,0x0
.stabs "stack:ZTc\"<class T> struct stack\";v2\"X\"
    1\"Z\";1\"int  stack(int );\" 1\"int  stack();\"
    1\"void  push(T );\" \",N_LSYM,0x0,0x0,0x0
.stabs "1\" pop();\" 1\"int  size()const ;\";1\"T * v;\"
    1\"T * p;\" 1\"int  sz;\";1\"static int  cnt;\";\"friend top
    (class stack&);\" \",N_LSYM,0x0,0x0,0x0
.stabs "\"friend class B;\";",N_LSYM,0x0,0x0,0x0
```

The first four `N_LSYM` (local symbol) stabs describe the `X` and `Z` classes. The first of each pair describes a structure (symbol descriptor `S`) and is generated by the C compiler and gives the mangled names of each member. The second of each pair is a C++ class description (symbol descriptor `Zc`).

The fifth stab is the C++ template description stab (symbol descriptor `ZTc`) and is continued in the sixth and seventh stabs. The string in the fifth and sixth stabs ends with a backslash. This indicates that the string is continued in the next stab. This breaks down as follows:

The name of the template is *stack*. The declaration for this is
`<class T> struct stack`
Note that this string, like other text strings in this stab, is bracketed by *backslash-quote* pairs. This allows semicolons to be embedded within these strings.

The next string is
`v2\"X\" 1\"Z\"`
which describes `X` as a `virtual private` base class and `Z` as a `public` base class.

This is followed by
```
1\"int  stack(int );\" 1\"int  stack();\"
1\"void  push(T );\" 1\" pop();\" 1\"int  size()const ;\"
```
which gives the declarations and protection for five member functions. Note that there are semicolons within the strings.

Following this is
`1\"T * v;\" 1\"T * p;\" 1\"int  sz;\"`
which contains the member data declarations for `public` members `v`, `p`, and `sz`.

Next is the source text for the static member cnt:
`1\"static int  cnt;\"`
which is `public`.

Finally, there is the string that declares `friend` functions and classes:
`\"friend top(stack&);\" \"friend class B;\"`

Again, each of the backslashes and quotes must be escaped with a backslash when used in a *.stabs* directive. This results in this last stab being coded as follows:

```
.stabs "stack:ZTc\\\"<class T> struct stack\\\";v2\\\"X\\\"
    1\\\"Z\\\";1\\\"int  stack(int );\\\" 1\\\"int  stack();\\\"
    1\\\"void  push(T );\\\" \\",N_LSYM,0,0,0
.stabs "1\\\" pop();\\\" 1\\\"int  size()const ;\\\";
    1\\\"T * v;\\\" 1\\\"T * p;\\\" 1\\\"int  sz;\\\";
    1\\\"static int  cnt;\\\";\\\"friend top(class stack&);\\\"
    \\",N_LSYM,0,0,0
.stabs "\\\"friend class B;\\\";",N_LSYM,0,0,0
```

C++ function template (ZTtF)

The function template has the following format:

Name* : ZTtF *Decl

Name is the name of the template.

Decl is the source text that declares the template. This is all of the source text following the keyword `template`, up to the opening left brace of the declaration. Comments and extra spaces are removed.

If the following function template follows the code listed above

```
template<class T> T top(stack<T>& s)
{
   return *s.v;
}
```

generates the following stab:

```
.stabs "top:ZTtF\"<class T>  top(stack&)\"",N_LSYM,0x0,0x0,0x0
```

Fortran90 Dope Vectors

E

This is the description for the Solaris operating environment and SunOS operating environment implementation, but represents the layout for dope vectors in general.

A dope vector represents a pointer, an allocatable array, or an anonymous reference to an array-valued object. A dope vector consists of a header followed by bounds and stride information. A pointer to a scalar consists solely of a header.

In the descriptions that follow, a word is a 32-bit word. The bits contained in a word are numbered sequentially from 31 to 0, where 31 is the highest-order bit and 0 is the lowest-order bit.

The header portion of a dope vector , i.e. the local variable `dv_hdr`, consists of the following six words:

word 1: `base_address`
: `base_address` is a 32-bit pointer. If the dope vector is associated (not `NULL`), `base_address` points to the starting address of the referenced data.

word 2: `length`
: `length` is a 32-bit unsigned integer. The `length` field gives the length of the data entities referenced by the dope vector. For a dope vector that references a scalar, the length is the length of the scalar. For a dope vector that references an array, the

length is the length of a single element of the array. For types other than `CHARACTER` (see word 4), the length is given in bits. For type `CHARACTER`, the length is given in bytes.

word 3: `flags`

bit 31: `associated`

If the `associated` flag is zero, the dope vector is not associated. `base_address` contains a meaningless value. If the `associated` flag is one, the dope vector is associated. `base_address` points to the start of the referenced data. If the dope vector references an allocatable array, the field `orig_base` points to the address to be used in deallocation, which must be the same as `base_address`.

bit 30: `ptr_alloc`

If the dope vector is a pointer, the flag `ptr_alloc` indicates whether the pointer references a dynamically allocated object. If `ptr_alloc` is zero, it does not. If `ptr_alloc` is one, it does.

bit 29: `allocatable`

If the flag `allocatable` is one, the dope vector references an allocatable array. If `allocatable` is zero, it does not.

bit 28: `pointer`

If the flag `pointer` is one, the dope vector is a pointer. If `pointer` is zero, the dope vector is not a pointer. The flags `allocatable` and `pointer` should never both be set to one. If neither `allocatable` nor `pointer` is set, the dope vector represents an anonymous value such as an array section.

bits 27 - 3: `unused`

bits 2 - 0: `n_dim`

The field `n_dim` indicates the number of dimensions of the data referenced by the dope vector.

word 4: `type`

bits 31 - 24: `base_type`

The `base_type` field is an 8-bit unsigned integer. The type values currently assigned are:

1 TYPELESS
2 INTEGER
3 REAL
4 COMPLEX
5 LOGICAL
6 CHARACTER
7 DERIVED_CHARACTER
8 DERIVED_WORD

The type TYPELESS is used for BOZ constants and Hollerith data and cannot be used in a dope vector. The type DERIVED_CHARACTER is a derived type (a structure) all of whose members are composed solely of character items. Data of type DERIVED_CHARACTER is byte-aligned. The type DERIVED_WORD is a derived type

that must be word-aligned.

bit 23: dpflag
The flag dpflag is one if the type associated with the dope vector in the source program was DOUBLE PRECISION or DOUBLE COMPLEX and zero otherwise.

bits 22 - 20: kind_or_star
The field kind_or_star is a three-bit unsigned integer. If kind_or_star is zero, no KIND or byte size was specified for the dope vector. If kind_or_star is one, a KIND was specified. If kind_or_star is two, a byte-size was specified.

bits 19 - 8: int_len
The field int_len is a twelve-bit unsigned integer. It is not set for types DERIVED_CHARACTER or DERIVED_WORD. It is the length of an individual datum in bits. For types other than CHARACTER and derived types, it should equal the field length (word 2).

bits 7 - 0: dec_len
The field dec_len is an eight-bit unsigned integer. If the field kind_or_star is set to one, dec_len is the KIND

number declared for the dope vector. If `kind_or_star` is set to two, it is the size of the data in bytes. If `kind_or_star` is zero, `dec_len` is meaningless.

word 5: `orig_base`
: `orig_base` is a 32-bit pointer. If the dope vector references an allocatable array or a pointer to an entire data object, `orig_base` points to the start of the object. `orig_base` is used to deallocate the referenced data object.

word 6: `orig_size`
: `orig_size` is a 32-bit unsigned integer. `orig_size` is the size of the allocated data object in bits.

If the dope vector is not a reference to a scalar, the header is followed by a sequence of three 32-bit words, a triplet, for each dimension. The first triplet corresponds to the first dimension, the second to the second dimension, etc. The first word of a triplet is a 32-bit signed integer giving the lower bound of that dimension. Negative lower bounds are allowed. The second word is a 32-bit signed integer that gives the extent of the dimension. If the upper bound minus the lower bound plus one is greater than or equal to zero, that is the extent. Otherwise, the extent is zero. The third word is the stride. The stride is the distance in bytes between successive elements in the given dimension.

Given a dope vector of dimension `n` and a sequence of `n` subscripts, `Sub_1, ..., Sub_n`, the address of the element addressed by the subscripts is:

```
base_address + SUM((Sub_i - Lower_Bound_i)*Stride_i), for 1<=i<=n.
```

Globalization F

In order 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 will create a unique prefix and prepend it to each file static variable. The unique symbol will consist 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 29, with the form:

```
G=<prefix>
```

The globalization prefix will be of the form:

```
.<internet_address><time_of_day><process_id>
```

The 80-bits will be 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 */
        }
}
```

Glossary

a.out

The file format used by SunOS to store executable files, shared libraries, and the object files created by compilers and assemblers. The format is fully described in section 5 of the SunOS manual pages. All a.out format files may be used on the Solaris operating environment, but debugging is not supported.

ELF

The Executable and Linking Format stores executable files, shared libraries, and object files on the Solaris opearating 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

1. Symbol table entry, used throughout this document to refer to those generated by the debugger. In an a.out file, the assembler and linker generate entries with the same format to support linking and relocating object files.

2. A symbol table entry in the a.out format. The same format is used in both a.out files and ELF files to contain debugging information.

.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. It is ignored on the SunOS operating enrivonment generating a.out files.