MUNIX V.3

Programmer's Guide (Part 2)

Documentation-No.: G0930.005-0688

Table of Contents

The Common Object File Format (COFF)

This section describes the Common Object File Format (COFF) used on CADMUS computers with the UNIX operating system. COFF is the format of the output file produced by the assembler, as, and the link editor, ld.

Some key features of COFF are

- applications can add system-dependent information to the object file without causing access utilities to become obsolete
- space is provided for symbolic information used by debuggers and other applications
- programmers can modify the way the object file is constructed by providing directives at compile time

The object file supports user-defined sections and contains extensive information for symbolic software testing. An object file contains

- a file header
- optional header information
- a table of section headers
- data corresponding to the section headers
- relocation information
- line numbers
- a symbol table
- a string table

Figure 11-1 shows the overall structure.

FILE HEADER
Optional Information
Section 1 Header
...
Section *n* Header
Raw Data for Section 1
...
Raw Data for Section *n*
Relocation Info for Sect. 1
...
Relocation Info for Sect. *n*
Line Numbers for Sect. 1
...
Line Numbers for Sect. *n*
SYMBOL TABLE
STRING TABLE

Figure 11-1: Object File Format

The last four sections (relocation, line numbers, symbol table, and the string table) may be missing if the program is linked with the **–s** option of the **ld** command, or if the line number information, symbol table, and string table are removed by the **strip** command. The line number information does not appear unless the program is compiled with the **–g** option of the **cc** command. Also, if there are no unresolved external references after linking, the relocation information is no longer needed and is absent. The string table is also absent if the source file does not contain any symbols with names longer than eight characters.

An object file that contains no errors or unresolved references is considered executable.

Definitions and Conventions

Before proceeding further, you should become familiar with the following terms and conventions.

Sections

A section is the smallest portion of an object file that is relocated and treated as one separate and distinct entity. In the most common case, there are three sections named **.text**, **.data**, and **.bss**. Additional sections accommodate comments, multiple text or data segments, shared data segments, or user-specified sections. However, the UNIX operating system loads only **.text**, **.data**, and **.bss** into memory when the file is executed.

NOTE It is a mistake to assume that every COFF file will have a certain number of sections, or to assume characteristics of sections such as their order, their location in the object file, or the address at which they are to be loaded. This information is available only after the object file has been created. Programs manipulating COFF files should obtain it from file and section headers in the file.

Physical and Virtual Addresses

The physical address of a section or symbol is the offset of that section or symbol from address zero of the address space. The term physical address as used in COFF does not correspond to general usage. The physical address of an object is not necessarily the address at which the object is placed when the process is executed. For example, on a system with paging, the address is located with respect to address zero of virtual memory and the system performs another address translation. The section header contains two address fields, a physical address, and a virtual address; but in all versions of COFF on UNIX systems, the physical address is equivalent to the virtual address.

Target Machine

Compilers and link editors produce executable object files that are intended to be run on a particular computer. In the case of cross-compilers, the compilation and link editing are done on one computer with the intent of creating an object file that can be executed on another computer. The term target machine refers to the computer on which the object file is destined to run. In the majority of cases, the target machine is the exact same computer on which the object file is being created.

File Header

The file header contains the 20 bytes of information shown in Figure 11-2. The last 2 bytes are flags that are used by **ld** and object file utilities.

Bytes	Declaration	Name	Description
0-1	unsigned short	f_magic	Magic number
2-3	unsigned short	f_nscns	Number of sections
4-7	long int	f_timdat	Time and date stamp indicating when the file was created, expressed as the number of elapsed seconds since 00:00:00 GMT, January 1, 1970
8-11	long int	f_symptr	File pointer containing the starting address of the symbol table
12-15	long int	f_nsyms	Number of entries in the symbol table
16-17	unsigned short	f_opthdr	Number of bytes in the optional header
18-19	unsigned short	f_flags	Flags (see Figure 11-3)

Figure 11-2: File Header Contents

Magic Numbers

The magic number specifies the target machine on which the object file is executable.

Flags

The last 2 bytes of the file header are flags that describe the type of the object file. Currently defined flags are found in the header file **filehdr.h**, and are shown in Figure 11-3.

Mnemonic	Flag	Meaning
F_RELFLG	00001	Relocation information stripped from the file
F_EXEC	00002	File is executable (i.e., no unresolved external references)
F_LNNO	00004	Line numbers stripped from the file
F_LSYMS	00010	Local symbols stripped from the file
F_AR32W	0001000	32 bit word

Figure 11-3: File Header Flags (CADMUS 32bit Computer)

File Header Declaration

The C structure declaration for the file header is given in Figure 11-4. This declaration may be found in the header file **filehdr.h**.

```
struct filehdr
{
    unsigned short   f_magic;    /* magic number */
    unsigned short   f_nscns;    /* number of section */

    long             f_timdat;   /* time and date stamp */

    long             f_symptr;   /* file ptr to symbol table */

    long             f_nsyms;    /* number entries in the symbol table */

    unsigned short   f_opthdr;   /* size of optional header */

    unsigned short   f_flags;    /* flags */
};

#define FILHDR struct filehdr
#define FILHSZ sizeof(FILHDR)
```

Figure 11-4: File Header Declaration

Optional Header Information

The template for optional information varies among different systems that use COFF. Applications place all system-dependent information into this record. This allows different operating systems access to information that only that operating system uses without forcing all COFF files to save space for that information. General utility programs (for example, the symbol table access library functions, the disassembler, etc.) are made to work properly on any common object file. This is done by seeking past this record using the size of optional header information in the file header field **f_opthdr**.

Standard UNIX System a.out Header

By default, files produced by the link editor for a UNIX system always have a standard UNIX system **a.out** header in the optional header field. The UNIX system **a.out** header is 28 bytes. The fields of the optional header are described in Figure 11-5.

Bytes	Declaration	Name	Description
0-1	**short**	**magic**	Magic number
2-3	**short**	**vstamp**	Version stamp
4-7	**long int**	**tsize**	Size of text in bytes
8-11	**long int**	**dsize**	Size of initialized data in bytes
12-15	**long int**	**bsize**	Size of uninitialized data in bytes
16-19	**long int**	**entry**	Entry point
20-23	**long int**	**text_start**	Base address of text
24-27	**long int**	**data_start**	Base address of data

Figure 11-5: Optional Header Contents (CADMUS 32bit Computers)

Whereas the magic number in the file header specifies the machine on which the object file runs, the magic number in the optional header supplies information telling the operating system on that machine how that file should be executed. The magic numbers recognized by the CADMUS MUNIX operating system are given in Figure 11-6.

Value	Meaning
0407	The text segment is not write-protected or sharable; the data segment is contiguous with the text segment.
0410	The data segment starts at the next segment following the text segment and the text segment is write protected.
0413	Text and data segments are aligned within **a.out** so it can be directly paged.

Figure 11-6: UNIX System Magic Numbers (CADMUS 32bit Computers)

Optional Header Declaration

The C language structure declaration currently used for the UNIX system **a.out** file header is given in Figure 11-7. This declaration may be found in the header file **aouthdr.h**.

```
typedef struct aouthdr
{
        short    magic;         /* magic number */
        short    vstamp;        /* version stamp */
        long     tsize;         /* text size in bytes, padded */

                                /* to full word boundary */

        long     dsize;         /* initialized data size */

        long     bsize;         /* uninitialized data size */

        long     entry;         /* entry point */

        long     text_start;    /* base of text for this file */

        long     data_start     /* base of data for this file */

} AOUTHDR;
```

Figure 11-7: **aouthdr** Declaration

Section Headers

Every object file has a table of section headers to specify the layout of data within the file. The section header table consists of one entry for every section in the file. The information in the section header is described in Figure 11-8.

Bytes	Declaration	Name	Description
0-7	**char**	**s_name**	8-character null padded section name
8-11	**long int**	**s_paddr**	Physical address of section
12-15	**long int**	**s_vaddr**	Virtual address of section
16-19	**long int**	**s_size**	Section size in bytes
20-23	**long int**	**s_scnptr**	File pointer to raw data
24-27	**long int**	**s_relptr**	File pointer to relocation entries
28-31	**long int**	**s_lnnoptr**	File pointer to line number entries
32-33	**unsigned short**	**s_nreloc**	Number of relocation entries
34-35	**unsigned short**	**s_nlnno**	Number of line number entries
36-39	**long int**	**s_flags**	Flags (see Figure 11-9)

Figure 11-8: Section Header Contents

The size of a section is padded to a multiple of 4 bytes. File pointers are byte offsets that can be used to locate the start of data, relocation, or line number entries for the section. They can be readily used with the UNIX system function **fseek(3S)**.

Flags

The lower 2 bytes of the flag field indicate a section type. The flags are described in Figure 11-9.

Mnemonic	Flag	Meaning
STYP_REG	0x00	Regular section (allocated, relocated, loaded)
STYP_DSECT	0x01	Dummy section (not allocated, relocated, not loaded)
STYP_NOLOAD	0x02	Noload section (allocated, relocated, not loaded)
STYP_GROUP	0x04	Grouped section (formed from input sections)
STYP_PAD	0x08	Padding section (not allocated, not relocated, loaded)
STYP_COPY	0x10	Copy section (for a decision function used in updating fields; not allocated, not relocated, loaded, relocation and line number entries processed normally)
STYP_TEXT	0x20	Section contains executable text
STYP_DATA	0x40	Section contains initialized data
STYP_BSS	0x80	Section contains only uninitialized data
STYP_INFO	0x200	Comment section (not allocated, not relocated, not loaded)
STYP_OVER	0x400	Overlay section (relocated, not allocated, not loaded)
STYP_LIB	0x800	For .lib section (treated like STYP_INFO)

Figure 11-9: Section Header Flags

Section Header Declaration

The C structure declaration for the section headers is described in Figure 11-10. This declaration may be found in the header file **scnhdr.h**.

```
struct scnhdr
{
        char        s_name[8];          /* section name */
        long        s_paddr;            /* physical address */
        long        s_vaddr;            /* virtual address */
        long        s_size;             /* section size */
        long        s_scnptr;           /* file ptr to section raw data */

        long        s_relptr;           /* file ptr to relocation */

        long        s_lnnoptr;          /* file ptr to line number */

        unsigned short   s_nreloc;      /* number of relocation entries */

        unsigned short   s_nlnno;       /* number of line number entries */

        long        s_flags;            /* flags */

};

#define  SCNHDR  struct scnhdr
#define  SCNHSZ  sizeof(SCNHDR)
```

Figure 11-10: Section Header Declaration

.bss Section Header

The one deviation from the normal rule in the section header table is the entry for uninitialized data in a .bss section. A .bss section has a size and symbols that refer to it, and symbols that are defined in it. At the same time, a .bss section has no relocation entries, no line number entries, and no data. Therefore, a .bss section has an entry in the section header table but occupies no space elsewhere in the file. In this case, the number of relocation and line number entries, as well as all file pointers in a .bss section header, are 0. The same is true of the STYP_NOLOAD and STYP_DSECT sections.

Sections

Figure 11-1 shows that section headers are followed by the appropriate number of bytes of text or data. The raw data for each section begins on a 4-byte boundary in the file.

Link editor SECTIONS directives (see Chapter 12) allow users to, among other things:

- describe how input sections are to be combined
- direct the placement of output sections
- rename output sections

If no SECTIONS directives are given, each input section appears in an output section of the same name. For example, if a number of object files, each with a **.text** section, are linked together the output object file contains a single **.text** section made up of the combined input **.text** sections.

Relocation Information

Object files have one relocation entry for each relocatable reference in the text or data. The relocation information consists of entries with the format described in Figure 11-11.

Bytes	**Declaration**	**Name**	**Description**
0-3	**long int**	**r_vaddr**	(Virtual) address of reference
4-7	**long int**	**r_symndx**	Symbol table index
8-9	**unsigned short**	**r_type**	Relocation type

Figure 11-11: Relocation Section Contents

The first 4 bytes of the entry are the virtual address of the text or data to which this entry applies. The next field is the index, counted from 0, of the symbol table entry that is being referenced. The type field indicates the type of relocation to be applied.

As the link editor reads each input section and performs relocation, the relocation entries are read. They direct how references found within the input section are treated. The currently recognized relocation types are given in Figure 11-12.

Mnemonic	Flag	Meaning
R_ABS	0	Reference is absolute; no relocation is necessary. The entry will be ignored.
R_RELLONG	021	The 32-bit reference to the symbol's virtual address is added to the 32-bit value in the code.

Figure 11-12: Relocation Types (CADMUS 32bit Computers)

Relocation Entry Declaration

The structure declaration for relocation entries is given in Figure 11-13. This declaration may be found in the header file **reloc.h**.

```
struct reloc
{
    long            r_vaddr;    /* virtual address of reference */

    long            r_symndx;   /* index into symbol table */

    unsigned short  r_type;     /* relocation type */
};

#define RELOC   struct reloc

#define RELSZ   10
```

Figure 11-13: Relocation Entry Declaration

Line Numbers

When invoked with the –g option, the cc, and f77 commands cause an entry in the object file for every source line where a breakpoint can be inserted. You can then reference line numbers when using a software debugger like **sdb**. All line numbers in a section are grouped by function as shown in Figure 11-14.

symbol index	0
physical address	line number
physical address	line number
.	.
.	.
.	.
symbol index	0
physical address	line number
physical address	line number

Figure 11-14: Line Number Grouping

The first entry in a function grouping has line number 0 and has, in place of the physical address, an index into the symbol table for the entry containing the function name. Subsequent entries have actual line numbers and addresses of the text corresponding to the line numbers. The line number entries are relative to the beginning of the function, and appear in increasing order of address.

Line Number Declaration

The structure declaration currently used for line number entries is given in Figure 11-15.

```
struct lineno
{
        union
        {
                long    l_symndx;    /* symtbl index of func name */

                long    l_paddr;     /* paddr of line number */
        } l_addr;
        unsigned short  l_lnno;      /* line number */

};

#define LINENO      struct lineno
#define LINESZ      6
```

Figure 11-15: Line Number Entry Declaration

Symbol Table

Because of symbolic debugging requirements, the order of symbols in the symbol table is very important. Symbols appear in the sequence shown in Figure 11-16.

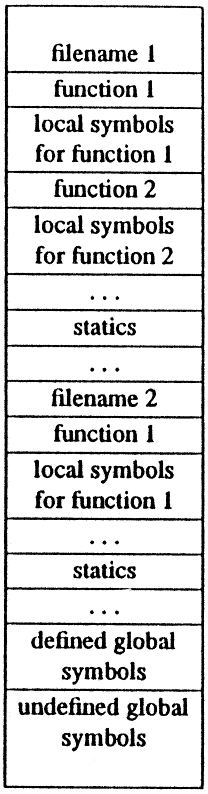


Figure 11-16: COFF Symbol Table

The word statics in Figure 11-16 means symbols defined with the C language storage class static outside any function. The symbol table consists of at least one fixed-length entry per symbol with some symbols followed by auxiliary entries of the same size. The entry for each symbol is a structure that holds the value, the type, and other information.

Special Symbols

The symbol table contains some special symbols that are generated by as, and other tools. These symbols are given in Figure 11-17.

Symbol	Meaning
.file	filename
.text	address of **.text** section
.data	address of **.data** section
.bss	address of **.bss** section
.bb	address of start of inner block
.eb	address of end of inner block
.bf	address of start of function
.ef	address of end of function
.target	pointer to the structure or union returned by a function
.xfake	dummy tag name for structure, union, or enumeration
.eos	end of members of structure, union, or enumeration
etext	next available address after the end of the output section **.text**
edata	next available address after the end of the output section **.data**
end	next available address after the end of the output section **.bss**

Figure 11-17: Special Symbols in the Symbol Table

Six of these special symbols occur in pairs. The **.bb** and **.eb** symbols indicate the boundaries of inner blocks; a **.bf** and **.ef** pair brackets each function. An **.xfake** and **.eos** pair names and defines the limit of structures, unions, and enumerations that were not named. The **.eos** symbol also appears after named structures, unions, and enumerations.

When a structure, union, or enumeration has no tag name, the compiler invents a name to be used in the symbol table. The name chosen for the symbol table is **.*x*fake**, where *x* is an integer. If there are three unnamed structures, unions, or enumerations in the source, their tag names are **.0fake**, **.1fake**, and **.2fake**. Each of the special symbols has different information stored in the symbol table entry as well as the auxiliary entries.

Inner Blocks

The C language defines a block as a compound statement that begins and ends with braces, {, and }. An inner block is a block that occurs within a function (which is also a block).

For each inner block that has local symbols defined, a special symbol, **.bb**, is put in the symbol table immediately before the first local symbol of that block. Also a special symbol, **.eb**, is put in the symbol table immediately after the last local symbol of that block. The sequence is shown in Figure 11-18.

.bb
local symbols for that block
.eb

Figure 11-18: Special Symbols (**.bb** and **.eb**)

Because inner blocks can be nested by several levels, the **.bb**-**.eb** pairs and associated symbols may also be nested. See Figure 11-19.

```
 {                            /* block 1 */
         int i;
         char c;
         ...
         {                        /* block 2 */
                 long a;
                 ...
             {                        /* block 3 */
                       int x;
                       ....
             }                        /* block 3 */

         }                        /* block 2 */

         {                        /* block 4 */
                       long i;
                       ...
         }                        /* block 4 */
}                            /* block 1 */
```

Figure 11-19: Nested blocks

The symbol table would look like Figure 11-20.

.bb for block 1
i
c
.bb for block 2
a
.bb for block 3
x
.eb for block 3
.eb for block 2
.bb for block 4
i
.eb for block 4
.eb for block 1

Figure 11-20: Example of the Symbol Table

Symbols and Functions

For each function, a special symbol **.bf** is put between the function name and the first local symbol of the function in the symbol table. Also, a special symbol **.ef** is put immediately after the last local symbol of the function in the symbol table. The sequence is shown in Figure 11-21.

function name
.bf
local symbol
.ef

Figure 11-21: Symbols for Functions

Symbol Table Entries

All symbols, regardless of storage class and type, have the same format for their entries in the symbol table. The symbol table entries each contain 18 bytes of information. The meaning of each of the fields in the symbol table entry is described in Figure 11-22. It should be noted that indices for symbol table entries begin at 0 and count upward. Each auxiliary entry also counts as one symbol.

Bytes	Declaration	Name	Description
0-7	(see text below)	**_n**	These 8 bytes contain either a symbol name or an index to a symbol
8-11	**long int**	**n_value**	Symbol value; storage class dependent
12-13	**short**	**n_scnum**	Section number of symbol
14-15	**unsigned short**	**n_type**	Basic and derived type specification
16	**char**	**n_sclass**	Storage class of symbol
17	**char**	**n_numaux**	Number of auxiliary entries

Figure 11-22: Symbol Table Entry Format

Symbol Names

The first 8 bytes in the symbol table entry are a union of a character array and two longs. If the symbol name is eight characters or less, the (null-padded) symbol name is stored there. If the symbol name is longer than eight characters, then the entire symbol name is stored in the string table. In this case, the 8 bytes contain two long integers, the first is zero, and the second is the offset (relative to the beginning of the string table) of the name in the string table. Since there can be no symbols with a null name, the zeroes on the first 4 bytes serve to distinguish a symbol table entry with an offset from one with a name in the first 8 bytes as shown in Figure 11-23.

Bytes	Declaration	Name	Description
0-7	**char**	**n_name**	8-character null-padded symbol name
0-3	**long**	**n_zeroes**	Zero in this field indicates the name is in the string table
4-7	**long**	**n_offset**	Offset of the name in the string table

Figure 11-23: Name Field

Special symbols generated by the C Compilation System are discussed above in "Special Symbols."

Storage Classes

The storage class field has one of the values described in Figure 11-24. These **#define**'s may be found in the header file **storclass.h**.

Mnemonic	Value	Storage Class
C_EFCN	–1	physical end of a function
C_NULL	0	–
C_AUTO	1	automatic variable
C_EXT	2	external symbol
C_STAT	3	static
C_REG	4	register variable
C_EXTDEF	5	external definition
C_LABEL	6	label
C_ULABEL	7	undefined label
C_MOS	8	member of structure
C_ARG	9	function argument
C_STRTAG	10	structure tag
C_MOU	11	member of union
C_UNTAG	12	union tag
C_TPDEF	13	type definition
C_USTATIC	14	uninitialized static
C_ENTAG	15	enumeration tag
C_MOE	16	member of enumeration
C_REGPARM	17	register parameter
C_FIELD	18	bit field

Figure 11-24: Storage Classes (Sheet 1 of 2)

Mnemonic	Value	Storage Class
C_BLOCK	100	beginning and end of block
C_FCN	101	beginning and end of function
C_EOS	102	end of structure
C_FILE	103	filename
C_LINE	104	used only by utility programs
C_ALIAS	105	duplicated tag
C_HIDDEN	106	like static, used to avoid name conflicts

Figure 11-24: Storage Classes (Sheet 2 of 2)

All of these storage classes except for C_ALIAS and C_HIDDEN are generated by the cc or as commands. The compress utility, **cprs**(1), generates the C_ALIAS mnemonic. This utility removes duplicated structure, union, and enumeration definitions and puts alias entries in their places. The storage class C_HIDDEN is not used by any UNIX system tools.

Some of these storage classes are used only internally by the C Compilation Systems. These storage classes are C_EFCN, C_EXTDEF, C_ULABEL, C_USTATIC, and C_LINE.

Storage Classes for Special Symbols

Some special symbols are restricted to certain storage classes. They are given in Figure 11-25.

Special Symbol	Storage Class
.file	C_FILE
.bb	C_BLOCK
.eb	C_BLOCK
.bf	C_FCN
.ef	C_FCN
.target	C_AUTO
.xfake	C_STRTAG, C_UNTAG, C_ENTAG
.eos	C_EOS
.text	C_STAT
.data	C_STAT
.bss	C_STAT

Figure 11-25: Storage Class by Special Symbols

Also some storage classes are used only for certain special symbols. They are summarized in Figure 11-26.

Storage Class	Special Symbol
C_BLOCK	.bb, .eb
C_FCN	.bf, .ef
C_EOS	.eos
C_FILE	.file

Figure 11-26: Restricted Storage Classes

Symbol Value Field

The meaning of the value of a symbol depends on its storage class. This relationship is summarized in Figure 11-27.

Storage Class	Meaning of Value
C_AUTO	stack offset in bytes
C_EXT	relocatable address
C_STAT	relocatable address
C_REG	register number
C_LABEL	relocatable address
C_MOS	offset in bytes
C_ARG	stack offset in bytes
C_STRTAG	0
C_MOU	0
C_UNTAG	0
C_TPDEF	0
C_ENTAG	0
C_MOE	enumeration value
C_REGPARM	register number
C_FIELD	bit displacement
C_BLOCK	relocatable address
C_FCN	relocatable address
C_EOS	size
C_FILE	(see text below)
C_ALIAS	tag index
C_HIDDEN	relocatable address

Figure 11-27: Storage Class and Value

If a symbol has storage class C_FILE, the value of that symbol equals the symbol table entry index of the next .file symbol. That is, the .file entries form a one-way linked list in the symbol table. If there are no more .file entries in the symbol table, the value of the symbol is the index of the first global symbol.

Relocatable symbols have a value equal to the virtual address of that symbol. When the section is relocated by the link editor, the value of these symbols changes.

Section Number Field

Section numbers are listed in Figure 11-28.

Mnemonic	Section Number	Meaning
N_DEBUG	-2	Special symbolic debugging symbol
N_ABS	-1	Absolute symbol
N_UNDEF	0	Undefined external symbol
N_SCNUM	1-077777	Section number where symbol is defined

Figure 11-28: Section Number

A special section number (-2) marks symbolic debugging symbols, including structure/union/enumeration tag names, typedefs, and the name of the file. A section number of -1 indicates that the symbol has a value but is not relocatable. Examples of absolute-valued symbols include automatic and register variables, function arguments, and **.eos** symbols.

With one exception, a section number of 0 indicates a relocatable external symbol that is not defined in the current file. The one exception is a multiply defined external symbol (i.e., FORTRAN common or an uninitialized variable defined external to a function in C). In the symbol table of each file where the symbol is defined, the section number of the symbol is 0 and the value of the symbol is a positive number giving the size of the symbol. When the files are combined to form an executable object file, the link editor combines all the input symbols of the same name into one symbol with the section number of the **.bss** section. The maximum size of all the input symbols with the same name is used to allocate space for the symbol and the value becomes the address of the symbol. This is the only case where a symbol has a section number of 0 and a non-zero value.

Section Numbers and Storage Classes

Symbols having certain storage classes are also restricted to certain section numbers. They are summarized in Figure 11-29.

Storage Class	Section Number
C_AUTO	N_ABS
C_EXT	N_ABS, N_UNDEF, N_SCNUM
C_STAT	N_SCNUM
C_REG	N_ABS
C_LABEL	N_UNDEF, N_SCNUM
C_MOS	N_ABS
C_ARG	N_ABS
C_STRTAG	N_DEBUG
C_MOU	N_ABS
C_UNTAG	N_DEBUG
C_TPDEF	N_DEBUG
C_ENTAG	N_DEBUG
C_MOE	N_ABS
C_REGPARM	N_ABS
C_FIELD	N_ABS
C_BLOCK	N_SCNUM
C_FCN	N_SCNUM
C_EOS	N_ABS
C_FILE	N_DEBUG
C_ALIAS	N_DEBUG

Figure 11-29: Section Number and Storage Class

Type Entry

The type field in the symbol table entry contains infomation about the basic and derived type for the symbol. This information is generated by the C Compilation System only if the **-g** option is used. Each symbol has exactly one basic or fundamental type but can have more than one derived type. The format of the 16-bit type entry is

d6	d5	d4	d3	d2	d1	typ

Bits 0 through 3, called **typ**, indicate one of the fundamental types given in Figure 11-30.

Mnemonic	Value	Type
T_NULL	0	type not assigned
T_VOID	1	void
T_CHAR	2	character
T_SHORT	3	short integer
T_INT	4	integer
T_LONG	5	long integer
T_FLOAT	6	floating point
T_DOUBLE	7	double word
T_STRUCT	8	structure
T_UNION	9	union
T_ENUM	10	enumeration
T_MOE	11	member of enumeration
T_UCHAR	12	unsigned character
T_USHORT	13	unsigned short
T_UINT	14	unsigned integer
T_ULONG	15	unsigned long

Figure 11-30: Fundamental Types

Bits 4 through 15 are arranged as six 2-bit fields marked **d1** through **d6**. These **d** fields represent levels of the derived types given in Figure 11-31.

Mnemonic	Value	Type
DT_NON	0	no derived type
DT_PTR	1	pointer
DT_FCN	2	function
DT_ARY	3	array

Figure 11-31: Derived Types

The following examples demonstrate the interpretation of the symbol table entry representing type.

```
char *func();
```

Here **func** is the name of a function that returns a pointer to a character. The fundamental type of **func** is 2 (character), the **d1** field is 2 (function), and the **d2** field is 1 (pointer). Therefore, the type word in the symbol table for **func** contains the hexadecimal number 0x62, which is interpreted to mean a function that returns a pointer to a character.

```
short *tabptr[10][25][3];
```

Here **tabptr** is a three-dimensional array of pointers to short integers. The fundamental type of **tabptr** is 3 (short integer); the **d1**, **d2**, and **d3** fields each contains a 3 (array), and the **d4** field is 1 (pointer). Therefore, the type entry in the symbol table contains the hexadecimal number 0x7f3 indicating a three-dimensional array of pointers to short integers.

Type Entries and Storage Classes

Figure 11-32 shows the type entries that are legal for each storage class.

Storage Class	d Entry			typ Entry Basic Type
	Function?	Array?	Pointer?	
C_AUTO	no	yes	yes	Any except T_MOE
C_EXT	yes	yes	yes	Any except T_MOE
C_STAT	yes	yes	yes	Any except T_MOE
C_REG	no	no	yes	Any except T_MOE
C_LABEL	no	no	no	T_NULL
C_MOS	no	yes	yes	Any except T_MOE
C_ARG	yes	no	yes	Any except T_MOE
C_STRTAG	no	no	no	T_STRUCT
C_MOU	no	yes	yes	Any except T_MOE
C_UNTAG	no	no	no	T_UNION

Figure 11-32: Type Entries by Storage Class (Sheet 1 of 2)

Storage Class	d Entry			typ Entry Basic Type
	Function?	Array?	Pointer?	
C_TPDEF	no	yes	yes	Any except T_MOE
C_ENTAG	no	no	no	T_ENUM
C_MOE	no	no	no	T_MOE
C_REGPARM	no	no	yes	Any except T_MOE
C_FIELD	no	no	no	T_ENUM, T_UCHAR, T_USHORT, T_UNIT, T_ULONG
C_BLOCK	no	no	no	T_NULL
C_FCN	no	no	no	T_NULL
C_EOS	no	no	no	T_NULL
C_FILE	no	no	no	T_NULL
C_ALIAS	no	no	no	T_STRUCT, T_UNION, T_ENUM

Figure 11-32: Type Entries by Storage Class (Sheet 2 of 2)

Conditions for the **d** entries apply to **d1** through **d6**, except that it is impossible to have two consecutive derived types of function.

Although function arguments can be declared as arrays, they are changed to pointers by default. Therefore, no function argument can have array as its first derived type.

Structure for Symbol Table Entries

The C language structure declaration for the symbol table entry is given in Figure 11-33. This declaration may be found in the header file **syms.h**.

```
struct syment
{
   union
   {
          char              _n_name[SYMNMLEN];    /* symbol name*/
          struct
          {
                 long     _n_zeroes;      /* symbol name */

                 long     _n_offset;      /* location in string table */
          } _n_n;
          char              *_n_nptr[2];    /* allows overlaying */
   } _n;
   unsigned long      n_value;                 /* value of symbol */

   short              n_scnum;                 /* section number */

   unsigned short     n_type;                  /* type and derived */

   char               n_sclass;                /* storage class */

   char               n_numaux;                /* number of aux entries */
};

#define  n_name          _n._n_name
#define  n_zeroes        _n._n_n._n_zeroes
#define  n_offset        _n._n_n._n_offset
#define  n_nptr          _n._n_nptr[1]

#define  SYMNMLEN  8
#define  SYMESZ    18    /* size of a symbol table entry */
```

Figure 11-33: Symbol Table Entry Declaration

Auxiliary Table Entries

An auxiliary table entry of a symbol contains the same number of bytes as the symbol table entry. However, unlike symbol table entries, the format of an auxiliary table entry of a symbol depends on its type and storage class. They are summarized in Figure 11-34.

Name	Storage Class	Type Entry		Auxiliary Entry Format
		d1	typ	
.file	C_FILE	DT_NON	T_NULL	filename
.text,.data, .bss	C_STAT	DT_NON	T_NULL	section
tagname	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_NULL	tag name
.eos	C_EOS	DT_NON	T_NULL	end of structure
fcname	C_EXT C_STAT	DT_FCN	(Note 1)	function
arrname	(Note 2)	DT_ARY	(Note 1)	array
.bb,.eb	C_BLOCK	DT_NON	T_NULL	beginning and end of block
.bf,.ef	C_FCN	DT_NON	T_NULL	beginning and end of function
name related to structure, union, enumeration	(Note 2)	DT_PTR, DT_ARR, DT_NON	T_STRUCT, T_UNION, T_ENUM	name related to structure, union, enumeration

Figure 11-34: Auxiliary Symbol Table Entries

Notes to Figure 11-34:
1. Any except T_MOE.
2. C_AUTO, C_STAT, C_MOS, C_MOU, C_TPDEF.

In Figure 11-34, *tagname* means any symbol name including the special symbol .xfake, and *fcname* and *arrname* represent any symbol name for a function or an array respectively. Any symbol that satisfies more than one condition in Figure 11-34 should have a union format in its auxiliary entry.

NOTE It is a mistake to assume how many auxiliary entries are associated with any given symbol table entry. This information is available, and should be obtained from the **n_numaux** field in the symbol table.

Filenames

Each of the auxiliary table entries for a filename contains a 14-character filename in bytes 0 through 13. The remaining bytes are 0.

Sections

The auxiliary table entries for sections have the format as shown in Figure 11-35.

Bytes	Declaration	Name	Description
0-3	**long int**	**x_scnlen**	section length
4-5	**unsigned short**	**x_nreloc**	number of relocation entries
6-7	**unsigned short**	**x_nlinno**	number of line numbers
8-17	–	–	unused (filled with zeroes)

Figure 11-35: Format for Auxiliary Table Entries for Sections

Tag Names

The auxiliary table entries for tag names have the format shown in Figure 11-36.

Bytes	Declaration	Name	Description
0-5	–	–	unused (filled with zeroes)
6-7	**unsigned short**	**x_size**	size of structure, union, and enumeration
8-11	–	–	unused (filled with zeroes)
12-15	**long int**	**x_endndx**	index of next entry beyond this structure, union, or enumeration
16-17	–	–	unused (filled with zeroes)

Figure 11-36: Tag Names Table Entries

End of Structures

The auxiliary table entries for the end of structures have the format shown in Figure 11-37:

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	–	–	unused (filled with zeroes)
6-7	unsigned short	x_size	size of structure, union, or enumeration
8-17	–	–	unused (filled with zeroes)

Figure 11-37: Table Entries for End of Structures

Functions

The auxiliary table entries for functions have the format shown in Figure 11-38:

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-7	long int	x_fsize	size of function (in bytes)
8-11	long int	x_lnnoptr	file pointer to line number
12-15	long int	x_endndx	index of next entry beyond this point
16-17	unsigned short	x_tvndx	index of the function's address in the transfer vector table (not used in UNIX system)

Figure 11-38: Table Entries for Functions

Arrays

The auxiliary table entries for arrays have the format shown in Figure 11-39. Defining arrays having more than four dimensions produces a warning message.

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	unsigned short	x_lnno	line number of declaration
6-7	unsigned short	x_size	size of array
8-9	unsigned short	x_dimen[0]	first dimension
10-11	unsigned short	x_dimen[1]	second dimension
12-13	unsigned short	x_dimen[2]	third dimension
14-15	unsigned short	x_dimen[3]	fourth dimension
16-17	–	–	unused (filled with zeroes)

Figure 11-39: Table Entries for Arrays

End of Blocks and Functions

The auxiliary table entries for the end of blocks and functions have the format shown in Figure 11-40:

Bytes	Declaration	Name	Description
0-3	–	–	unused (filled with zeroes)
4-5	unsigned short	x_lnno	C-source line number
6-17	–	–	unused (filled with zeroes)

Figure 11-40: End of Block and Function Entries

Beginning of Blocks and Functions

The auxiliary table entries for the beginning of blocks and functions have the format shown in Figure 11-41:

Bytes	Declaration	Name	Description
0-3	–	–	unused (filled with zeroes)
4-5	unsigned short	x_lnno	C-source line number
6-11	–	–	unused (filled with zeroes)
12-15	long int	x_endndx	index of next entry past this block
16-17	–	–	unused (filled with zeroes)

Figure 11-41: Format for Beginning of Block and Function

Names Related to Structures, Unions, and Enumerations

The auxiliary table entries for structure, union, and enumeration symbols have the format shown in Figure 11-42:

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	–	–	unused (filled with zeroes)
6-7	unsigned short	x_size	size of the structure, union, or enumeration
8-17	–	–	unused (filled with zeroes)

Figure 11-42: Entries for Structures, Unions, and Enumerations

Aggregates defined by **typedef** may or may not have auxiliary table entries. For example,

```
typedef struct people STUDENT;

struct people
{
        char name[20];
        long id;
};

typedef struct people EMPLOYEE;
```

The symbol EMPLOYEE has an auxiliary table entry in the symbol table but symbol STUDENT will not because it is a forward reference to a structure.

Auxiliary Entry Declaration

The C language structure declaration for an auxiliary symbol table entry is given in Figure 11-43. This declaration may be found in the header file syms.h.

```
union auxent
{
      struct
      {
            long   x_tagndx;
            union
            {
                struct
                {
                      unsigned short   x_lnno;
                      unsigned short   x_size;
                } x_lnsz;
                long    x_fsize;
            } x_misc;
            union
            {
                struct
  .
  .
  .
```

Figure 11-43: Auxiliary Symbol Table Entry (Sheet 1 of 2)

```
.
.
.
                {
                     long    x_lnnoptr;
                     long    x_endndx;
                } x_fcn;
                struct
                {
                     unsigned short    x_dimen[DIMNUM];
                } x_ary;
            } x_fcnary;
            unsigned short    x_tvndx;
        } x_sym;
        struct
        {
            char    x_fname[FILNMLEN];
        } x_file;
        struct
        {
            long    x_scnlen;
            unsigned short    x_nreloc;
            unsigned short    x_nlinno;
        } x_scn;
        struct
        {
            long    x_tvfill;
            unsigned short    x_tvlen;
            unsigned short    x_tvran[2];
        } x_tv;
}
#define FILNMLEN  14
#define DIMNUM    4
#define AUXENT    union auxent
#define AUXESZ    18
```

Figure 11-43: Auxiliary Symbol Table Entry (Sheet 2 of 2)

String Table

Symbol table names longer than eight characters are stored contiguously in the string table with each symbol name delimited by a null byte. The first four bytes of the string table are the size of the string table in bytes; offsets into the string table, therefore, are greater than or equal to 4. For example, given a file containing two symbols (with names longer then eight characters, **long_name_1** and **another_one**) the string table has the format as shown in Figure 11-44:

‘l’	‘o’	‘n’	‘g’
‘_’	‘n’	‘a’	‘m’
‘e’	‘_’	‘1’	‘\0’
‘a’	‘n’	‘o’	‘t’
‘h’	‘e’	‘r’	‘_’
‘o’	‘n’	‘e’	‘\0’

Figure 11-44: String Table

The index of **long_name_1** in the string table is 4 and the index of **another_one** is 16.

Access Routines

UNIX system releases contain a set of access routines that are used for reading the various parts of a common object file. Although the calling program must know the detailed structure of the parts of the object file it processes, the routines effectively insulate the calling program from the knowledge of the overall structure of the object file.

The access routines can be divided into four categories:

1. functions that open or close an object file
2. functions that read header or symbol table information
3. functions that position an object file at the start of a particular section of the object file
4. a function that returns the symbol table index for a particular symbol

These routines can be found in the library **libld.a** and are listed in Section 3 of **Manual Ib**. A summary of what is available can be found under **ldfcn**(4).

Table of Contents

Chapter 12: The Link Editor

The Link Editor

In Chapter 2 there was a discussion of link editor command line options (some of which may also be provided on the cc(1) command line). This chapter contains information on the Link Editor Command Language.

The command language enables you to

- specify the memory configuration of the target machine
- combine the sections of an object file in arrangements other than the default
- bind sections to specific addresses or within specific portions of memory
- define or redefine global symbols

Under most normal circumstances there is no compelling need to have such tight control over object files and where they are located in memory. When you do need to be very precise in controlling the link editor output, you do it by means of the command language.

Link editor command language directives are passed in a file named on the ld(1) command line. Any file named on the command line that is not identifiable as an object module or an archive library is assumed to contain directives. The following paragraphs define terms and describe conditions with which you need to be familiar before you begin to use the command language.

Memory Configuration

The virtual memory of the target machine is, for purposes of allocation, partitioned into configured and unconfigured memory. The default condition is to treat all memory as configured. It is common with microprocessor applications, however, to have different types of memory at different addresses. For example, an application might have 3K of PROM (Programmable Read-Only Memory) beginning at address 0, and 8K of ROM (Read-Only Memory) starting at 20K. Addresses in the range 3K to 20K–1 are then not configured. Unconfigured memory is treated as reserved or unusable by ld(1). Nothing can ever be linked into unconfigured memory. Thus, specifying a certain memory range to be unconfigured is one way of marking the addresses (in that range) illegal or nonexistent with respect to the linking process. Memory configurations other than the default must be explicitly specified by you (the user).

Unless otherwise specified, all discussion in this document of memory, addresses, etc. are with respect to the configured sections of the address space.

Sections

A section of an object file is the smallest unit of relocation and must be a contiguous block of memory. A section is identified by a starting address and a size. Information describing all the sections in a file is stored in section headers at the start of the file. Sections from input files are combined to form output sections that contain executable text, data, or a mixture of both. Although there may be holes or gaps between input sections and between output sections, storage is allocated contiguously within each output section and may not overlap a hole in memory.

Addresses

The physical address of a section or symbol is the relative offset from address zero of the address space. The physical address of an object is not necessarily the location at which it is placed when the process is executed. For example, on a system with paging, the address is with respect to address zero of the virtual space, and the system performs another address translation.

Binding

It is often necessary to have a section begin at a specific, predefined address in the address space. The process of specifying this starting address is called binding, and the section in question is said to be bound to or bound at the required address. While binding is most commonly relevant to output sections, it is also possible to bind special absolute global symbols with an assignment statement in the **ld**(1) command language.

Object File

Object files are produced both by the assembler (typically as a result of calling the compiler) and by **ld**(1). **ld**(1) accepts relocatable object files as input and produces an output object file that may or may not be relocatable. Under certain special circumstances, the input object files given to **ld**(1) can also be absolute files.

Files produced from the compilation system may contain, among others, sections called **.text** and **.data**. The **.text** section contains the instruction text (executable instructions), **.data** contains initialized data variables. For example, if a C program contained the global (i.e., not inside a function) declaration

```
int i = 100;
```

and the assignment

```
i = 0;
```

then compiled code from the C assignment is stored in **.text**, and the variable **i** is located in **.data**.

Link Editor Command Language

Expressions

Expressions may contain global symbols, constants, and most of the basic C language operators. (See Figure 12-2, "Syntax Diagram for Input Directives.") Constants are as in C with a number recognized as decimal unless preceded with 0 for octal or 0x for hexadecimal. All numbers are treated as long integers's. Symbol names may contain uppercase or lowercase letters, digits, and the underscore, _. Symbols within an expression have the value of the address of the symbol only. **ld**(1) does not do symbol table lookup to find the contents of a symbol, the dimensionality of an array, structure elements declared in a C program, etc.

ld(1) uses a **lex**-generated input scanner to identify symbols, numbers, operators, etc. The current scanner design makes the following names reserved and unavailable as symbol names or section names:

ADDR	BLOCK	GROUP	NEXT	RANGE	SPARE
ALIGN	COMMON	INFO	NOLOAD	REGIONS	PHY
ASSIGN	COPY	LENGTH	ORIGIN	SECTIONS	TV
BIND	DSECT	MEMORY	OVERLAY	SIZEOF	

addr	block	length	origin	sizeof
align	group	next	phy	spare
assign	l	o	range	
bind	len	org	s	

The operators that are supported, in order of precedence from high to low, are shown in Figure 12-1:

symbol
! ~ – (UNARY Minus)
* / %
+ – (BINARY Minus)
>> <<
== != > < <= >=
&
|
&&
||
= += –= *= /=

Figure 12-1: Operator Symbols

The above operators have the same meaning as in the C language. Operators on the same line have the same precedence.

Assignment Statements

External symbols may be defined and assigned addresses via the assignment statement. The syntax of the assignment statement is

```
symbol = expression;
```

or

```
symbol op= expression;
```

where *op* is one of the operators +, –, *, or / . Assignment statements must be terminated by a semicolon.

All assignment statements (with the exception of the one case described in the following paragraph) are evaluated after allocation has been performed. This occurs after all input-file-defined symbols are appropriately relocated but before the actual relocation of the text and data itself. Therefore, if an assignment statement expression contains any symbol name, the address used for that symbol in the evaluation of the expression reflects the symbol address in the output object file. References within text and data (to symbols given a value through an assignment statement) access this latest assigned value. Assignment statements are processed in the same order in which they are input to ld(1).

Assignment statements are normally placed outside the scope of section-definition directives (see "Section Definition Directives" under "Link Editor Command Language"). However, there exists a special symbol, called **dot**, **.**, that can occur only within a section-definition directive. This symbol refers to the current address of **ld**(1)'s location counter. Thus, assignment expressions involving **.** are evaluated during the allocation phase of **ld**(1). Assigning a value to the **.** symbol within a section-definition directive can increment (but not decrement) **ld**(1)'s location counter and can create holes within the section, as described in "Section Definition Directives." Assigning the value of the **.** symbol to a conventional symbol permits the final allocated address (of a particular point within the link edit run) to be saved.

align is provided as a shorthand notation to allow alignment of a symbol to an *n*-byte boundary within an output section, where *n* is a power of 2. For example, the expression

```
align(n)
```

is equivalent to

```
(. + n - 1) &~(n - 1)
```

SIZEOF and ADDR are pseudo-functions that, given the name of a section, return the size or address of the section respectively. They may be used in symbol definitions outside of section directives.

Link editor expressions may have either an absolute or a relocatable value. When **ld**(1) creates a symbol through an assignment statement, the symbol's value takes on that type of expression. That type depends on the following rules:

- An expression with a single relocatable symbol (and zero or more constants or absolute symbols) is relocatable.
- The difference of two relocatable symbols from the same section is absolute.
- All other expressions are combinations of the above.

Specifying a Memory Configuration

MEMORY directives are used to specify

1. The total size of the virtual space of the target machine.
2. The configured and unconfigured areas of the virtual space.

If no directives are supplied, **ld**(1) assumes that all memory is configured. The size of the default memory is dependent upon the target machine.

By means of MEMORY directives, an arbitrary name of up to eight characters is assigned to a virtual address range. Output sections can then be forced to be bound to virtual addresses within specifically named memory areas. Memory names may contain uppercase or lowercase letters, digits, and the special characters $, ., or _. Names of memory ranges are used by **ld**(1) only and are not carried in the output file symbol table or headers.

When MEMORY directives are used, all virtual memory not described in a MEMORY directive is considered to be unconfigured. Unconfigured memory is not used in **ld**(1)'s allocation process; hence nothing except DSECT sections can be link edited or bound to an address within unconfigured memory.

As an option on the MEMORY directive, attributes may be associated with a named memory area. In future releases this may be used to provide error checking. Currently, error checking of this type is not implemented.

The attributes currently accepted are

1. R : readable memory
2. W : writable memory
3. X : executable, i.e., instructions may reside in this memory
4. I : initializable, i.e., stack areas are typically not initialized

Other attributes may be added in the future if necessary. If no attributes are specified on a MEMORY directive or if no MEMORY directives are supplied, memory areas assume the attributes of R, W, X, and I.

The syntax of the MEMORY directive is

```
MEMORY
{
        name1 (attr) :origin = n1, length = n2
        name2 (attr) :origin = n3, length = n4
        etc.
}
```

The keyword **origin** (or **org** or **o**) must precede the origin of a memory range, and **length** (or **len** or **l**) must precede the length as shown in the above prototype. The **origin** operand refers to the virtual address of the memory range. **origin** and **length** are entered as long integer constants in either decimal, octal, or hexadecimal (standard C syntax). **origin** and **length** specifications, as well as individual MEMORY directives, may be separated by white space or a comma.

By specifying MEMORY directives, **ld**(1) can be told that memory is configured in some manner other than the default. For example, if it is necessary to prevent anything from being linked to the first 0x10000 words of memory, a MEMORY directive can accomplish this.

```
MEMORY
{
        valid : org = 0x10000, len = 0xFE0000
}
```

Section Definition Directives

The purpose of the SECTIONS directive is to describe how input sections are to be combined, to direct where to place output sections (both in relation to each other and to the entire virtual memory space), and to permit the renaming of output sections.

In the default case where no SECTIONS directives are given, all input sections of the same name appear in an output section of that name. If two object files are linked, one containing sections s1 and s2 and the other containing sections s3 and s4, the output object file contains the four sections s1, s2, s3, and s4. The order of these sections would depend on the order in which the link editor sees the input files.

The basic syntax of the SECTIONS directive is

```
SECTIONS
{
        secname1 :
        {
                file_specifications,
                assignment_statements
        }
        secname2 :
        {
                file_specifications,
                assignment_statements
        }
etc.
}
```

The various types of section definition directives are discussed in the remainder of this section.

File Specifications

Within a section definition, the files and sections of files to be included in the output section are listed in the order in which they are to appear in the output section. Sections from an input file are specified by

```
filename ( secname )
```

or

```
filename ( secnam1 secnam2 . . . )
```

Sections of an input file are separated either by white space or commas as are the file specifications themselves.

```
filename [COMMON]
```

may be used in the same way to refer to all the uninitialized, unallocated global symbols in a file.

If a file name appears with no sections listed, then all sections from the file (but not the uninitialized, unallocated globals) are linked into the current output section. For example,

```
SECTIONS
{
        outsecl:
        {
                file1.o (secl)
                file2.o
                file3.o (secl, sec2)
        }
}
```

According to this directive, the order in which the input sections appear in the output section **outsecl** would be

1. section **secl** from file **filel.o**
2. all sections from **file2.o**, in the order they appear in the file
3. section **secl** from file **file3.o**, and then section **sec2** from file **file3.o**

If there are any additional input files that contain input sections also named **outsecl**, these sections are linked following the last section named in the definition of **outsecl**. If there are any other input sections in **filel.o** or **file3.o**, they will be placed in output sections with the same names as the input sections unless they are included in other file specifications.

The code

```
* (secname)
```

may be used to indicate all previously unallocated input sections of the given name, regardless of what input file they are contained in.

Load a Section at a Specified Address

Bonding of an output section to a specific virtual address is accomplished by an **ld**(1) option as shown in the following SECTIONS directive example:

```
SECTIONS
{
        outsec addr:
        {
                . . .
        }
        etc.
}
```

The *addr* is the bonding address expressed as a C constant. If **outsec** does not fit at *addr* (perhaps because of holes in the memory configuration or because **outsec** is too large to fit without overlapping some other output section), **ld(1)** issues an appropriate error message. *addr* may also be the word BIND, followed by a parenthesized expression. The expression may use the pseudo-functions SIZEOF, ADDR or NEXT. NEXT accepts a constant and returns the first multiple of that value that falls into configured unallocated memory; SIZEOF and ADDR accept previously defined sections.
As long as output sections do not overlap and there is enough space, they can be bound anywhere in configured memory. The SECTIONS directives defining output sections need not be given to **ld(1)** in any particular order, unless SIZEOF or ADDR is used.
ld(1) does not ensure that each section's size consists of an even number of bytes or that each section starts on an even byte boundary. The assembler ensures that the size (in bytes) of a section is evenly divisible by 4. **ld(1)** directives can be used to force a section to start on an odd byte boundary although this is not recommended. If a section starts on an odd byte boundary, the section's contents are either accessed incorrectly or are not executed properly. When a user specifies an odd byte boundary, **ld(1)** issues a warning message.

Aligning an Output Section

It is possible to request that an output section be bound to a virtual address that falls on an *n*-byte boundary, where *n* is a power of 2. The ALIGN option of the SECTIONS directive performs this function, so that the option

```
ALIGN(n)
```

is equivalent to specifying a bonding address of

```
( . + n - 1) &~(n - 1)
```

For example

```
SECTIONS
{
        outsec  ALIGN(0x20000)  :
        {
                . . .
        }
        etc.
}
```

The output section **outsec** is not bound to any given address but is placed at some virtual address that is a multiple of 0x20000 (e.g., at address 0x0, 0x20000, 0x40000, 0x60000, etc.).

Grouping Sections Together

The default allocation algorithm for **ld**(1)

1. Links all input **.init** sections together, followed by **.text** sections, into one output section. This output section is called **.text** and is bound to an address of 0x0 plus the size of all headers in the output file.
2. Links all input **.data** sections together into one output section. This output section is called **.data** and, in paging systems, is bound to an address aligned to a machine dependent constant plus a number dependent on the size of headers and text.
3. Links all input **.bss** sections together with all uninitialized, unallocated global symbols, into one output section. This output section is called **.bss** and is allocated so as to immediately follow the output section **.data**. Note that the output section **.bss** is not given any particular address alignment.

Specifying any SECTIONS directives results in this default allocation not being performed. Rather than relying on the **ld**(1) default algorithm, if you are manipulating COFF files, the one certain way of determining address and order information is to take it from the file and section headers. The default allocation of **ld**(1) is equivalent to supplying the following directive:

```
SECTIONS
{
          .text sizeof_headers : { *(.init) *(.text) }
        GROUP BIND( NEXT(align_value) +
                    ((SIZEOF(.text) + ADDR(.text)) % 0x2000)) :
        {
                  .data     : { }
                  .bss      : { }
        }
}
```

where *align_value* is a machine dependent constant. The GROUP command ensures that the two output sections, **.data** and **.bss**, are allocated (e.g., grouped) together. Bonding or alignment information is supplied only for the group and not for the output sections contained within the group. The sections making up the group are allocated in the order listed in the directive.

If **.text**, **.data**, and **.bss** are to be placed in the same segment, the following SECTIONS directive is used:

```
SECTIONS
{
        GROUP                :
        {
                  .text        : { }
                  .data        : { }
                  .bss         : { }
        }
}
```

Note that there are still three output sections (**.text**, **.data**, and **.bss**), but now they are allocated into consecutive virtual memory.

This entire group of output sections could be bound to a starting address or aligned simply by adding a field to the GROUP directive. To bind to 0xC0000, use

```
        GROUP 0xC0000 : {
```

To align to 0x10000, use

```
        GROUP ALIGN(0x10000)  : {
```

With this addition, first the output section **.text** is bound at 0xC0000 (or is aligned to 0x10000); then the remaining members of the group are allocated in order of their appearance into the next available memory locations.

When the GROUP directive is not used, each output section is treated as an independent entity:

```
SECTIONS
{
        .text   : { }
        .data ALIGN(0x20000)   : { }
        .bss    : { }
}
```

The **.text** section starts at virtual address 0x0 (if it is in configured memory) and the **.data** section at a virtual address aligned to 0x20000. The **.bss** section follows immediately after the **.text** section if there is enough space. If there is not, it follows the **.data** section. The order in which output sections are defined to **ld**(1) cannot be used to force a certain allocation order in the output file.

Creating Holes Within Output Sections

The special symbol dot, **.**, appears only within section definitions and assignment statements. When it appears on the left side of an assignment statement, **.** causes **ld**(1)'s location counter to be incremented or reset and a hole left in the output section. Holes built into output sections in this manner take up physical space in the output file and are initialized using a fill character (either the default fill character (0x00) or a supplied fill character). See the definition of the **–f** option in "Using the Link Editor" and the discussion of filling holes in "Initialized Section Holes" or **.bss** Sections." in this chapter.

Consider the following section definition:

```
outsec:
{
        . += 0x1000;
        f1.o (.text)
        . += 0x100;
        f2.o (.text)
        . = align (4);
        f3.o (.text)
}
```

The effect of this command is as follows:

1. A 0x1000 byte hole, filled with the default fill character, is left at the beginning of the section. Input section **f1.o (.text)** is linked after this hole.

2. The **.text** section of input file **f2.o** begins at 0x100 bytes following the end of **f1.o (.text)**.

3. The **.text** section of **f3.o** is linked to start at the next full word boundary following the **.text** section of **f2.o** with respect to the beginning of **outsec**.

For the purposes of allocating and aligning addresses within an output section, **ld(1)** treats the output section as if it began at address zero. As a result, if, in the above example, **outsec** ultimately is linked to start at an odd address, then the part of **outsec** built from **f3.o (.text)** also starts at an odd address—even though **f3.o (.text)** is aligned to a full word boundary. This is prevented by specifying an alignment factor for the entire output section.

```
outsec ALIGN(4) : {
```

It should be noted that the assembler, **as**, always pads the sections it generates to a full word length making explicit alignment specifications unnecessary. This also holds true for the compiler.

Expressions that decrement **.** are illegal. For example, subtracting a value from the location counter is not allowed since overwrites are not allowed. The most common operators in expressions that assign a value to **.** are **+=** and **align.**

Creating and Defining Symbols at Link-Edit Time

The assignment instruction of **ld(1)** can be used to give symbols a value that is link-edit dependent. Typically, there are three types of assignments:

1. Use of **.** to adjust **ld(1)**'s location counter during allocation.
2. Use of **.** to assign an allocation-dependent value to a symbol.
3. Assigning an allocation-independent value to a symbol.

Case 1) has already been discussed in the previous section.

Case 2) provides a means to assign addresses (known only after allocation) to symbols. For example,

```
SECTIONS
{
        outsc1: {...}
        outsc2:
        {
                file1.o (s1)
                s2_start = . ;
                file2.o (s2)
                s2_end = . - 1;
        }
}
```

The symbol **s2_start** is defined to be the address of **file2.o(s2)**, and **s2_end** is the address of the last byte of **file2.o(s2)**.

Consider the following example:

```
SECTIONS
{
        outsc1:
        {
                file1.o ( .data)
                mark = .;
                . += 4;
                file2.o ( .data)
        }
}
```

In this example, the symbol **mark** is created and is equal to the address of the first byte beyond the end of **file1.o**'s **.data** section. Four bytes are reserved for a future run-time initialization of the symbol **mark**. The type of the symbol is a long integer (32 bits).

Assignment instructions involving **.** must appear within SECTIONS definitions since they are evaluated during allocation. Assignment instructions that do not involve **.** can appear within SECTIONS definitions but typically do not. Such instructions are evaluated after allocation is complete. Reassignment of a defined symbol to a different address is dangerous. For example, if a symbol within **.data** is defined, initialized, and referenced within a set of object files being link-edited, the symbol table entry for that symbol is changed to reflect the new, reassigned physical address. However, the associated initialized data is not moved to the new address, and there may be references to the old address. The **ld**(1) issues warning messages for each defined symbol that is being redefined within an ifile. However, assignments of absolute values to new symbols are safe because there are no references or initialized data associated with the symbol.

Allocating a Section Into Named Memory

It is possible to specify that a section be linked (somewhere) within a specific named memory (as previously specified on a MEMORY directive). (The > notation is borrowed from the UNIX system concept of redirected output.) For example,

```
MEMORY
{
        mem1:               o=0x000000    l=0x10000
        mem2  (RW):         o=0x020000    l=0x40000
        mem3  (RW):         o=0x070000    l=0x40000
        mem1:               o=0x120000    l=0x04000
}

SECTIONS
{
        outsec1: { f1.o( .data) } > mem1
        outsec2: { f2.o( .data) } > mem3
}
```

This directs **ld(1)** to place **outsec1** anywhere within the memory area named **mem1** (i.e., somewhere within the address range 0x0-0xFFFF or 0x120000-0x123FFF). The **outsec2** is to be placed somewhere in the address range 0x70000-0xAFFFF.

Initialized Section Holes or .bss Sections

When holes are created within a section (as in the example in "Creating Holes within Output Sections"), **ld(1)** normally puts out bytes of zero as fill. By default, **.bss** sections are not initialized at all; that is, no initialized data is generated for any **.bss** section by the assembler nor supplied by the link editor, not even zeros.

Initialization options can be used in a SECTIONS directive to set such holes or output **.bss** sections to an arbitrary 2-byte pattern. Such initialization options apply only to **.bss** sections or holes. As an example, an application might want an uninitialized data table to be initialized to a constant value without recompiling the **.o** file or a hole in the text area to be filled with a transfer to an error routine.

Either specific areas within an output section or the entire output section may be specified as being initialized. However, since no text is generated for an uninitialized **.bss** section, if part of such a section is initialized, then the entire section is initialized. In other words, if a **.bss** section is to be combined with a **.text** or **.data** section (both of which are initialized) or if part of an output **.bss** section is to be initialized, then one of the following will hold:

1. Explicit initialization options must be used to initialize all .bss sections in the output section.
2. **ld**(1) will use the default fill value to initialize all .bss sections in the output section.

Consider the following **ld**(1) ifile:

```
SECTIONS
{
        sec1:
        {
                f1.o
                . =+ 0x200;
                f2.o (.text)
        } = 0xDFFF
        sec2:
        {
                f1.o (.bss)
                f2.o (.bss) = 0x1234
        }
        sec3:
        {
                f3.o (.bss)
                . . .
        } = 0xFFFF
         sec4: { f4.o (.bss) }
}
```

In the example above, the 0x200 byte hole in section sec1 is filled with the value 0xDFFF. In section sec2, **f1.o(.bss)** is initialized to the default fill value of 0x00, and **f2.o(.bss)** is initialized to 0x1234. All .bss sections within sec3 as well as all holes are initialized to 0xFFFF. Section sec4 is not initialized; that is, no data is written to the object file for this section.

Notes and Special Considerations

Changing the Entry Point

The UNIX system **a.out** optional header contains a field for the (primary) entry point of the file. This field is set using one of the following rules (listed in the order they are applied):

1. The value of the symbol specified with the **–e** option, if present, is used.
2. The value of the symbol **__entry**, if present, is used.
3. The value of the symbol **_main**, if present, is used.
4. The value zero is used.

Thus, an explicit entry point can be assigned to this **a.out** header field through the **–e** option or by using an assignment instruction in an ifile of the form

```
        __entry    =    expression;
```

If **ld**(1) is called through **cc**(1), a startup routine is automatically linked in. Then, when the program is executed, the routine **exit**(1) is called after the main routine finishes to close file descriptors and do other cleanup. The user must therefore be careful when calling **ld**(1) directly or when changing the entry point. The user must supply the startup routine or make sure that the program always calls exit rather than falling through the end. Otherwise, the program will dump core.

Use of Archive Libraries

Each member of an archive library (e.g., **libc.a**) is a complete object file. Archive libraries are created with the **ar**(1) command from object files generated by **cc** or **as**. An archive library is always processed using selective inclusion: only those members that resolve existing undefined-symbol references are taken from the library for link editing. Libraries can be placed both inside and outside section definitions. In both cases, a member of a library is included for linking whenever

1. There exists a reference to a symbol defined in that member.
2. The reference is found by **ld**(1) prior to the actual scanning of the library.

When a library member is included by searching the library inside a SECTIONS directive, all input sections from the library member are included in the output section being defined. When a library member is included by searching the library outside of a SECTIONS directive, all input sections from the library member are included into the output section with the same name. If necessary, new output sections are defined to provide a place to put the input sections. Note, however, that

1. Specific members of a library cannot be referenced explicitly in an ifile.
2. The default rules for the placement of members and sections cannot be overridden when they apply to archive library members.

The **–l** option is a shorthand notation for specifying an input file coming from a predefined set of directories and having a predefined name. By convention, such files are archive libraries. However, they need not be so. Furthermore, archive libraries can be specified without using the **–l** option by simply giving the (full or relative) UNIX system file path.

The ordering of archive libraries is important since for a member to be extracted from the library it must satisfy a reference that is known to be unresolved at the time the library is searched. Archive libraries can be specified more than once. They are searched every time they are encountered. Archive files have a symbol table at the beginning of the archive. **ld**(1) will cycle through this symbol table until it has determined that it cannot resolve any more references from that library.

Consider the following example:

1. The input files **file1.o** and **file2.o** each contain a reference to the external function FCN.
2. Input **file1.o** contains a reference to symbol ABC.
3. Input **file2.o** contains a reference to symbol XYZ.
4. Library **liba.a**, member 0, contains a definition of XYZ.
5. Library **libc.a**, member 0, contains a definition of ABC.
6. Both libraries have a member 1 that defines FCN.

If the **ld**(1) command were entered as

```
        ld file1.o -la file2.o -lc
```

then the FCN references are satisfied by **liba.a**, member 1, ABC is obtained from **libc.a**, member 0, and XYZ remains undefined (because the library **liba.a** is searched before **file2.o** is specified). If the **ld**(1) command were entered as

```
        ld file1.o file2.o -la -lc
```

then the FCN references is satisfied by **liba.a**, member 1, ABC is obtained from **libc.a**, member 0, and XYZ is obtained from **liba.a**, member 0. If the **ld**(1) command were entered as

```
        ld file1.o file2.o -lc -la
```

then the FCN references is satisfied by **libc.a**, member 1, ABC is obtained from **libc.a**, member 0, and XYZ is obtained from **liba.a**, member 0.

The **-u** option is used to force the linking of library members when the link edit run does not contain an actual external reference to the members. For example,

```
        ld -u rout1 -la
```

creates an undefined symbol called **rout1** in **ld**(1)'s global symbol table. If any member of library **liba.a** defines this symbol, it (and perhaps other members as well) is extracted. Without the **-u** option, there would have been no unresolved references or undefined symbols to cause **ld**(1) to search the archive library.

Dealing With Holes in Physical Memory

When memory configurations are defined such that unconfigured areas exist in the virtual memory, each application or user must assume the responsibility of forming output sections that will fit into memory. For example, assume that memory is configured as follows:

```
MEMORY
{
        mem1:        o = 0x00000        l = 0x02000
        mem2:        o = 0x40000        l = 0x05000
        mem3:        o = 0x20000        l = 0x10000
}
```

Let the files **f1.o**, **f2.o**, . . . **f*n*.o** each contain three sections **.text**, **.data**, and **.bss**, and suppose the combined **.text** section is 0x12000 bytes. There is no configured area of memory in which this section can be placed. Appropriate directives must be supplied to break up the **.text** output section so ld(1) may do allocation. For example,

```
SECTIONS
{
        txt1:
        {
                f1.o ( .text)
                f2.o ( .text)
                f3.o ( .text)
        }
        txt2:
        {
                f4.o ( .text)
                f5.o ( .text)
                f6.o ( .text)
        }
        etc.
}
```

Allocation Algorithm

An output section is formed either as a result of a SECTIONS directive, by combining input sections of the same name, or by combining **.text** and **.init** into **.text**. An output section can have zero or more input sections comprising it. After the composition of an output section is determined, it must then be allocated into configured virtual memory. **ld**(1) uses an algorithm that attempts to minimize fragmentation of memory, and hence increases the possibility that a link edit run will be able to allocate all output sections within the specified virtual memory configuration. The algorithm proceeds as follows:

1. Any output sections for which explicit bonding addresses were specified are allocated.
2. Any output sections to be included in a specific named memory are allocated. In both this and the succeeding step, each output section is placed into the first available space within the (named) memory with any alignment taken into consideration.
3. Output sections not handled by one of the above steps are allocated.

If all memory is contiguous and configured (the default case), and no SECTIONS directives are given, then output sections are allocated in the order they appear to **ld**(1). Otherwise, output sections are allocated in the order they were defined or made known to **ld**(1) into the first available space they fit.

Incremental Link Editing

As previously mentioned, the output of **ld**(1) can be used as an input file to subsequent **ld**(1) runs providing that the relocation information is retained (**–r** option). Large applications may find it desirable to partition their C programs into subsystems, link each subsystem independently, and then link edit the entire application. For example,

Step 1:

ld –r –o outfile1 ifile1 infile1.o

```
/* ifile1  */
SECTIONS
{
        ss1:
        {
                f1.o
                f2.o
                . . .
                fn.o
        }
}
```

Step 2:

ld –r –o outfile2 ifile2 infile2.o

```
/* ifile2  */
SECTIONS
{
        ss2:
        {
                g1.o
                g2.o
                . . .
                gn.o
        }
}
```

Step 3:

ld –a –o final.out outfile1 outfile2

By judiciously forming subsystems, applications may achieve a form of incremental link editing whereby it is necessary to relink only a portion of the total link edit when a few files are recompiled.

To apply this technique, there are two simple rules

1. Intermediate link edits should contain only SECTIONS declarations and be concerned only with the formation of output sections from input files and input sections. No binding of output sections should be done in these runs.
2. All allocation and memory directives, as well as any assignment statements, are included only in the final **ld**(1) call.

DSECT, COPY, NOLOAD, INFO, and OVERLAY Sections

Sections may be given a type in a section definition as shown in the following example:

```
SECTIONS
{
        name1 0x200000 (DSECT)       : { file1.o }
        name2 0x400000 (COPY)        : { file2.o }
        name3 0x600000 (NOLOAD)      : { file3.o }
        name4          (INFO)        : { file4.o }
        name5 0x900000 (OVERLAY)     : { file5.o }
}
```

The DSECT option creates what is called a dummy section. A dummy section has the following properties:

1. It does not participate in the memory allocation for output sections. As a result, it takes up no memory and does not show up in the memory map generated by **ld**(1).

2. It may overlay other output sections and even unconfigured memory. DSECTs may overlay other DSECTs.

3. The global symbols defined within the dummy section are relocated normally. That is, they appear in the output file's symbol table with the same value they would have had if the DSECT were actually loaded at its virtual address. DSECT-defined symbols may be referenced by other input sections. Undefined external symbols found within a DSECT cause specified archive libraries to be searched and any members which define such symbols are link edited normally (i.e., not as a DSECT).

4. None of the section contents, relocation information, or line number information associated with the section is written to the output file.

In the above example, none of the sections from **file1.o** are allocated, but all symbols are relocated as though the sections were link edited at the specified address. Other sections could refer to any of the global symbols and they are resolved correctly.

A copy section created by the COPY option is similar to a dummy section. The only difference between a copy section and a dummy section is that the contents of a copy section and all associated information is written to the output file.

An INFO section is the same as a COPY section but its purpose is to carry information about the object file whereas the COPY section may contain valid text and data. INFO sections are usually used to contain file version identification information.

A section with the type of NOLOAD differs in only one respect from a normal output section: its text and/or data is not written to the output file. A NOLOAD section is allocated virtual space, appears in the memory map, etc.

An OVERLAY section is relocated and written to the output file. It is different from a normal section in that it is not allocated and may overlay other sections or unconfigured memory.

Output File Blocking

The BLOCK option (applied to any output section or GROUP directive) is used to direct **ld(1)** to align a section at a specified byte offset in the output file. It has no effect on the address at which the section is allocated nor on any part of the link edit process. It is used purely to adjust the physical position of the section in the output file.

```
SECTIONS
{
        .text BLOCK(0x200) : { }
        .data ALIGN(0x20000) BLOCK(0x200) : { }
}
```

With this SECTIONS directive, **ld**(1) assures that each section, **.text** and **.data**, is physically written at a file offset, which is a multiple of 0x200 (e.g., at an offset of 0, 0x200, 0x400, and so forth, in the file).

Nonrelocatable Input Files

If a file produced by **ld**(1) is intended to be used in a subsequent **ld**(1) run, the first **ld**(1) run should have the **-r** option set. This preserves relocation information and permits the sections of the file to be relocated by the subsequent run.

If an input file to **ld**(1) does not have relocation or symbol table information (perhaps from the action of a **strip**(1) command, or from being link edited without a **-r** option or with a **-s** option), the link edit run continues using the nonrelocatable input file.

For such a link edit to be successful (i.e., to actually and correctly link edit all input files, relocate all symbols, resolve unresolved references, etc.), two conditions on the nonrelocatable input files must be met.

1. Each input file must have no unresolved external references.
2. Each input file must be bound to the exact same virtual address as it was bound to in the **ld**(1) run that created it.

NOTE

If these two conditions are not met for all nonrelocatable input files, no error messages are issued. Because of this fact, extreme care must be taken when supplying such input files to ld(1).

Syntax Diagram for Input Directives

Directives	Expanded Directives
<ifile>	{<cmd>}
<cmd>	<memory>
	<sections>
	<assignment>
	<filename>
	<flags>
<memory>	MEMORY { <memory_spec>
<memory_spec>	<name> [<attributes>] :
<attributes>	({ R | W | X | I })
<origin_spec>	<origin> = <long>
<lenth_spec>	<length> = <long>
<origin>	ORIGIN | o | org | origin
<length>	LENGTH | l | len | length

Figure 12-2: Syntax Diagram for Input Directives (Sheet 1 of 4)

NOTE

Two punctuation symbols, square brackets and curly braces, do double duty in this diagram.

Where the actual symbols, [] and { } are used, they are part of the syntax and must be present when the directive is specified.

Where you see the symbols **[** and **]** (larger and in bold), it means the material enclosed is optional.

Where you see the symbols **{** and **}** (larger and in bold), it means multiple occurrences of the material enclosed are permitted.

Directives	Expanded Directives
<sections>	SECTIONS { {<sec_or_group>} }
<sec_or_group>	<section> | <group> | <library>
<group>	GROUP <group_options> : {
<section_list>	<section> { [,] <section> }
<section>	<name> <sec_options> :
<group_options>	[<addr>] | [<align_option>] [<block_option>]
<sec_options>	[<addr>] | [<align_option>]
<addr>	<long> | <bind>(<expr>)
<alignoption>	<align> (<expr>)
<align>	ALIGN | align
<block_option>	<block> (<long>)
<block>	BLOCK | block
<type_option>	(DSECT) | (NOLOAD) | (COPY)
<fill>	= <long>
<mem_spec>	> <name>
	> <attributes>
<statement>	<filename>
	<filename> (<name_list>) | [COMMON]
	* (<name_list>) | [COMMON]
	<assignment>
	<library>
	null

Figure 12-2: Syntax Diagram for Input Directives (Sheet 2 of 4)

Directives	Expanded Directives
<name_list>	<section_name> [,] { <section_name> }
<library>	–l<name>
<bind>	BIND | bind
<assignment>	<lside> <assign_op> <expr> <end>
<lside>	<name> | .
<assign_op>	= | += | –= | *= |/ =
<end>	; | ,
<expr>	<expr> <binary_op> <expr>
	<term>
<binary_op>	* | / | %
	+ | –
	>> | <<
	== | != | > | < | <= | >=
	&
	|
	&&
	||
<term>	<long>
	<name>
	<align> (<term>)
	(<expr>)
	<unary_op> <term>
	<phy> (<lside>)
	<sizeof>(<sectionname>)
	<next>(<long>)
	<addr>(<sectionname>)
<unary_op>	! | –
<phy>	PHY | phy
<sizeof>	SIZEOF | sizeof

Figure 12-2: Syntax Diagram for Input Directives (Sheet 3 of 4)

Directives	Expanded Directives
<next>	NEXT | next
<addr>	ADDR | addr
<flags>	–e<wht_space><name>
	–f<wht_space><long>
	–h<wht_space><long>
	–l<name>
	–m
	–o<wht_space><filename>
	–r
	–s
	–t
	–u<wht_space><name>
	–z
	–H
	–L<path_name>
	–M
	–N
	–S
	–V
	–VS<wht_space><long>
	–a
	–x
<name>	Any valid symbol name
<long>	Any valid long integer constant
<wht_space>	Blanks, tabs, and newlines
<filename>	Any valid UNIX operating system
<sectionname>	Any valid section name,
<path_name>	Any valid UNIX operating system

Figure 12-2: Syntax Diagram for Input Directives (Sheet 4 of 4)

Table of Contents

Chapter 13: **make**

Introduction

The trend toward increased modularity of programs means that a project may have to cope with a large assortment of individual files. There may also be a wide range of generation procedures needed to turn the assortment of individual files into the final executable product.

make(1) provides a method for maintaining up-to-date versions of programs that consist of a number of files that may be generated in a variety of ways.

An individual programmer can easily forget

- file-to-file dependencies
- files that were modified and the impact that it has on other files
- the exact sequence of operations needed to generate a new version of the program

In a description file, **make** keeps track of the commands that create files and the relationship between files. Whenever a change is made in any of the files that make up a program, the **make** command creates the finished program by recompiling only those portions directly or indirectly affected by the change.

The basic operation of **make** is to

- find the target in the description file
- ensure that all the files on which the target depends, the files needed to generate the target, exist and are up to date
- create the target file if any of the generators have been modified more recently than the target

The description file that holds the information on interfile dependencies and command sequences is conventionally called **makefile**, **Makefile**, or **s.[mM]akefile**. If this naming convention is followed, the simple command **make** is usually sufficient to regenerate the target regardless of the number files edited since the last **make**. In most cases, the description file is not difficult to write and changes infrequently. Even if only a single file has been edited, rather than typing all the commands to regenerate the target, typing the **make** command ensures the regeneration is done in the prescribed way.

Basic Features

The basic operation of **make** is to update a target file by ensuring that all of the files on which the target file depends exist and are up to date. The target file is regenerated if it has not been modified since the dependents were modified. The **make** program searches the graph of dependencies. The operation of **make** depends on its ability to find the date and time that a file was last modified.

The **make** program operates using three sources of information:

- a user-supplied description file
- filenames and last-modified times from the file system
- built-in rules to bridge some of the gaps

To illustrate, consider a simple example in which a program named **prog** is made by compiling and loading three C language files **x.c**, **y.c**, and **z.c** with the **math** library. By convention, the output of the C language compilations will be found in files named **x.o**, **y.o**, and **z.o**. Assume that the files **x.c** and **y.c** share some declarations in a file named **defs.h**, but that **z.c** does not. That is, **x.c** and **y.c** have the line

```
#include "defs.h"
```

The following specification describes the relationships and operations:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o   -lm  -o  prog

x.o  y.o :   defs.h
```

If this information were stored in a file named **makefile**, the command

make

would perform the operations needed to regenerate **prog** after any changes had been made to any of the four source files **x.c**, **y.c**, **z.c**, or **defs.h**. In the example above, the first line states that **prog** depends on three **.o** files. Once these object files are current, the second line describes how to load them to create **prog**. The third line states that **x.o** and **y.o** depend on the file **defs.h**. From the file system, **make** discovers that there are three **.c** files corresponding to the needed **.o** files and uses built-in rules on how to generate an object from a C source file (i.e., issue a **cc –c** command).

If **make** did not have the ability to determine automatically what needs to be done, the following longer description file would be necessary:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o  -lm  -o  prog
x.o :  x.c  defs.h
        cc  -c  x.c
y.o :  y.c  defs.h
        cc  -c  y.c
z.o :  z.c
        cc  -c  z.c
```

If none of the source or object files have changed since the last time **prog** was made, and all of the files are current, the command **make** announces this fact and stops. If, however, the **defs.h** file has been edited, **x.c** and **y.c** (but not **z.c**) are recompiled; and then **prog** is created from the new **x.o** and **y.o** files, and the existing **z.o** file. If only the file **y.c** had changed, only it is recompiled; but it is still necessary to reload **prog**. If no target name is given on the **make** command line, the first target mentioned in the description is created; otherwise, the specified targets are made. The command

make x.o

would regenerate **x.o** if **x.c** or **defs.h** had changed.

A method often useful to programmers is to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of **make**'s ability to generate files and substitute macros (for information about macros, see "Description Files and Substitutions" below.) Thus, an entry "save" might be included to copy a certain set of files, or an entry "clean" might be used to throw away unneeded intermediate files.

If a file exists after such commands are executed, the file's time of last modification is used in further decisions. If the file does not exist after the commands are executed, the current time is used in making further decisions.

You can maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

A simple macro mechanism for substitution in dependency lines and command strings is used by **make**. Macros can either be defined by command-line arguments or included in the description file. In either case, a macro consists of a name followed by an equals sign followed by what the macro stands for. A macro is invoked by preceding the name by a dollar sign. Macro names longer than one character must be parenthesized.

The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two are equivalent.

$*, $@, $?,** and **$< are four special macros that change values during the execution of the command. (These four macros are described later in this chapter under "Description Files and Substitutions.") The following fragment shows assignment and use of some macros:

```
OBJECTS = x.o y.o z.o
LIBES = -lm
prog: $(OBJECTS)
        cc $(OBJECTS)  $(LIBES)  -o prog
  . . .
```

The command

make LIBES="-ll -lm"

loads the three objects with both the **lex** (-ll) and the **math** (-lm) libraries, because macro definitions on the command line override definitions in the description file. (In UNIX system commands, arguments with embedded blanks must be quoted.)

As an example of the use of **make**, a description file that might be used to maintain the **make** command itself is given. The code for **make** is spread over a number of C language source files and has a **yacc** grammar. The description file contains the following:

```
# Description file for the make command

FILES = Makefile defs.h main.c doname.c misc.c
        files.c dosys.c gram.y
OBJECTS = main.o doname.o misc.o files.o
          dosys.o gram.o
LIBES= -lld
LINT = lint -p
CFLAGS = -O
LP = /usr/bin/lp

make:   $(OBJECTS)
        $(CC) $(CFLAGS) $(OBJECTS) $(LIBES) -o make
        @size make

$(OBJECTS):  defs.h

cleanup:
        -rm *.o gram.c
        -du

install:
        @size make /usr/bin/make
        cp make /usr/bin/make && rm make

lint :  dosys.c doname.c files.c main.c misc.c gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c \
        gram.c

                  # print files that are out-of-date
                  # with respect to "print" file.

print:   $(FILES)
         pr $? | $(LP)
         touch print
```

The make program prints out each command before issuing it.

The following output results from typing the command **make** in a directory containing only the source and description files:

```
cc  -O -c main.c
cc  -O -c doname.c
cc  -O -c misc.c
cc  -O -c files.c
cc  -O -c dosys.c
yacc  gram.y
mv y.tab.c gram.c
cc  -O -c gram.c
cc  main.o doname.o misc.o files.o dosys.o
     gram.o  -lld -o make
13188 + 3348 + 3044 = 19580
```

The string of digits results from the size make command. The printing of the command line itself was suppressed by an at sign, @, in the description file.

Description Files and Substitutions

The following section will explain the customary elements of the description file.

Comments

The comment convention is that a sharp, #, and all characters on the same line after a sharp are ignored. Blank lines and lines beginning with a sharp are totally ignored.

Continuation Lines

If a noncomment line is too long, the line can be continued by using a backslash. If the last character of a line is a backslash, then the backslash, the new line, and all following blanks and tabs are replaced by a single blank.

Macro Definitions

A macro definition is an identifier followed by an equal sign. The identifier must not be preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped). The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lm
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as its value. Remember, however, that some macros are explicitly defined in **make**'s own rules. (See Figure 13-2 at the end of the chapter.)

General Form

The general form of an entry in a description file is

```
target1 [target2 ...] :[:] [dependent1 ...] [; commands] [# ...]
[ \t commands] [# ...]
 . . .
```

Items inside brackets may be omitted and targets and dependents are strings of letters, digits, periods, and slashes. Shell metacharacters such as * and ? are expanded when the line is evaluated. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line. A command is any string of characters not including a sharp, #, except when the sharp is in quotes.

Dependency Information

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type. For the more common single-colon case, a command sequence may be associated with at most one dependency line. If the target is out of date with any of the dependents on any of the lines and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise, a default rule may be invoked. In the double-colon case, a command sequence may be associated with more than one dependency line. If the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. The double colon form is particularly useful in updating archive-type files, where the target is the archive library itself. (An example is included in the "Archive Libraries" section later in this chapter.)

Executable Commands

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the shell after substituting for macros. The printing is suppressed in the silent mode (**–s** option of the **make** command) or if the command line in the description file begins with an @ sign. **make** normally stops if any command signals an error by returning a nonzero error code. Errors are ignored if the **–i** flag has been specified on the **make** command line, if the fake target name .IGNORE appears in the description file, or if the command string in the description file begins with a hyphen. If a program is known to return a meaningless status, a hyphen in front of the command that invokes it is appropriate. Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (e.g., **cd** and shell control commands) that have meaning only within a single shell process. These results are forgotten before the next line is executed.

Before issuing any command, certain internally maintained macros are set. The **$@** macro is set to the full target name of the current target. The **$@** macro is evaluated only for explicitly named dependencies. The **$?** macro is set to the string of names that were found to be younger than the target. The **$?** macro is evaluated when explicit rules from the **makefile** are evaluated. If the command was generated by an implicit rule, the **$<** macro is the name of the related file that caused the action; and the **$*** macro is the prefix shared by the current and the dependent filenames. If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name DEFAULT are used. If there is no such name, **make** prints a message and stops.

In addition, a description file may also use the following related macros: **$(@D)**, **$(@F)**, **$(*D)**, **$(*F)**, **$(<D)**, and **$(<F)** (see below).

Extensions of $*, $@, and $<

The internally generated macros **$***, **$@**, and **$<** are useful generic terms for current targets and out-of-date relatives. To this list has been added the following related macros: **$(@D)**, **$(@F)**, **$(*D)**, **$(*F)**, **$(<D)**, and **$(<F)**. The **D** refers to the directory part of the single character macro. The **F** refers to the filename part of the single character macro. These additions are useful when building hierarchical **makefiles**. They allow access to directory names for purposes of using the **cd** command of the shell. Thus, a command can be

```
cd $(<D); $(MAKE) $(<F)
```

Output Translations

Macros in shell commands are translated when evaluated. The form is as follows:

```
$(macro:string1=string2)
```

The meaning of **$(macro)** is evaluated. For each appearance of **string1** in the evaluated macro, **string2** is substituted. The meaning of finding **string1** in **$(macro)** is that the evaluated **$(macro)** is considered as a series of strings each delimited by white space (blanks or tabs). Thus, the occurrence of **string1** in

$(macro) means that a regular expression of the following form has been found:

```
.*<string1>[TAB|BLANK]
```

This particular form was chosen because **make** usually concerns itself with suffixes. The usefulness of this type of translation occurs when maintaining archive libraries. Now, all that is necessary is to accumulate the out-of-date members and write a shell script, which can handle all the C language programs (i.e., those files ending in .c). Thus, the following fragment optimizes the executions of **make** for maintaining an archive library:

```
$(LIB): $(LIB)(a.o) $(LIB)(b.o) $(LIB)(c.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        $(AR) $(ARFLAGS) $(LIB) $?
        rm $?
```

A dependency of the preceding form is necessary for each of the different types of source files (suffixes) that define the archive library. These translations are added in an effort to make more general use of the wealth of information that **make** generates.

Recursive Makefiles

Another feature of **make** concerns the environment and recursive invocations. If the sequence $(MAKE) appears anywhere in a shell command line, the line is executed even if the **–n** flag is set. Since the **–n** flag is exported across invocations of **make** (through the MAKEFLAGS variable), the only thing that is executed is the **make** command itself. This feature is useful when a hierarchy of **makefile**(s) describes a set of software subsystems. For testing purposes, **make –n ...** can be executed and everything that would have been done will be printed including output from lower level invocations of **make**.

Suffixes and Transformation Rules

make uses an internal table of rules to learn how to transform a file with one suffix into a file with another suffix. If the **–r** flag is used on the **make** command line, the internal table is not used.

The list of suffixes is actually the dependency list for the name .SUFFIXES. **make** searches for a file with any of the suffixes on the list. If it finds one, **make** transforms it into a file with another suffix. The transformation rule names are the concatenation of the before and after suffixes. The name of the rule to transform a **.r** file to a **.o** file is thus **.r.o**. If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule **.r.o** is used. If a command is generated by using one of these suffixing rules, the macro **$*** is given the value of the stem (everything but the suffix) of the name of the file to be made; and the macro **$<** is the full name of the dependent that caused the action.

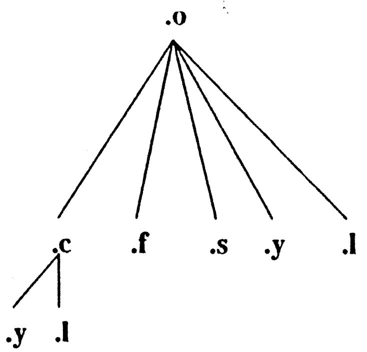
The order of the suffix list is significant since the list is scanned from left to right. The first name formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can add an entry for .SUFFIXES in the description file. The dependents are added to the usual list. A .SUFFIXES line without any dependents deletes the current list. It is necessary to clear the current list if the order of names is to be changed.

Implicit Rules

make uses a table of suffixes and a set of transformation rules to supply default dependency information and implied commands. The default suffix list is as follows:

.o	Object file
.c	C source file
.c~	SCCS C source file
.f	FORTRAN source file
.f~	SCCS FORTRAN source file
.s	Assembler source file
.s~	SCCS Assembler source file
.y	yacc source grammar
.y~	SCCS yacc source grammar
.l	lex source grammar
.l~	SCCS ex source grammar
.h	Header file
.h~	SCCS header file
.sh	Shell file
.sh~	SCCS shell file

Figure 13-1 summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.



If the file **x.o** is needed and an **x.c** is found in the description or directory, the **x.o** file would be compiled. If there is also an **x.l**, that source file would be run through **lex** before compiling the result. However, if there is no **x.c** but there is an **x.l**, **make** would discard the intermediate C language file and use the direct link as shown in Figure 13-1.

It is possible to change the names of some of the compilers used in the default or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, F77, YACC, and LEX. The command

```
make CC=newcc
```

Figure 13-1: Summary of Default Transformation Path

will cause the **newcc** command to be used instead of the usual C language compiler. The macros ASFLAGS, CFLAGS, F77FLAGS, YFLAGS, and LFLAGS may be set to cause these commands to be issued with optional flags. Thus

```
make "CFLAGS=-g"
```

causes the cc command to include debugging information.

Archive Libraries

The **make** program has an interface to archive libraries. A user may name a member of a library in the following manner:

```
projlib(object.o)
   or
projlib((entrypt))
```

where the second method actually refers to an entry point of an object file within the library. (**make** looks through the library, locates the entry point, and translates it to the correct object filename.)

To use this procedure to maintain an archive library, the following type of **makefile** is required:

```
projlib::    projlib(pfile1.o)
        $(CC) -c -O pfile1.c
        $(AR) $(ARFLAGS) projlib pfile1.o
        rm pfile1.o
projlib::    projlib(pfile2.o)
        $(CC) -c -O pfile2.c
        $(AR) $(ARFLAGS) projlib pfile2.o
        rm pfile2.o
```

... and so on for each object ...

This is tedious and error prone. Obviously, the command sequences for adding a C language file to a library are the same for each invocation; the filename being the only difference each time. (This is true in most cases.)

The **make** command also gives the user access to a rule for building libraries. The handle for the rule is the **.a** suffix. Thus, a **.c.a** rule is the rule for compiling a C language source file, adding it to the library, and removing the **.o** cadaver. Similarly, the **.y.a**, the **.s.a**, and the **.l.a** rules rebuild **yacc**, assembler, and **lex** files, respectively. The archive rules defined internally are **.c.a**, **.c~.a**, **.f.a**, **.f~.a**, and **.s~.a**. (The tilde, **~**, syntax will be described shortly.) The user may define other needed rules in the description file.

The above two-member library is then maintained with the following shorter **makefile**:

```
projlib:        projlib(pfile1.o) projlib(pfile2.o)
            @echo projlib up-to-date.
```

The internal rules are already defined to complete the preceding library maintenance. The actual **.c.a** rule is as follows:

```
.c.a:
        $(CC) -c $(CFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.o
```

Thus, the **$@** macro is the **.a** target (**projlib**); the **$<** and **$*** macros are set to the out-of-date C language file; and the filename minus the suffix, respectively (**pfile1.c** and **pfile1**). The **$<** macro (in the preceding rule) could have been changed to **$*.c**.

It might be useful to go into some detail about exactly what **make** does when it sees the construction

```
projlib:    projlib(pfile1.o)
        @echo projlib up-to-date
```

Assume the object in the library is out of date with respect to **pfile1.c**. Also, there is no **pfile1.o** file.

1. **make projlib.**
2. Before **makeing projlib**, check each dependent of **projlib**.
3. **projlib(pfile1.o)** is a dependent of **projlib** and needs to be generated.
4. Before generating **projlib(pfile1.o)**, check each dependent of **projlib(pfile1.o)**. (There are none.)
5. Use internal rules to try to create **projlib(pfile1.o)**. (There is no explicit rule.) Note that **projlib(pfile1.o)** has a parenthesis in the name to identify the target suffix as **.a**. This is the key. There is no explicit **.a** at the end of the **projlib** library name. The parenthesis implies the **.a** suffix. In this sense, the **.a** is hard-wired into **make**.
6. Break the name **projlib(pfile1.o)** up into **projlib** and **pfile1.o**. Define two macros, **$@** (**=projlib**) and **$*** (**=pfile1**).
7. Look for a rule ***X*.a** and a file **$****X*. The first *X* (in the .SUFFIXES list) which fulfills these conditions is **.c** so the rule is **.c.a**, and the file is **pfile1.c**. Set **$<** to be **pfile1.c** and execute the rule. In fact, **make** must then compile **pfile1.c**.
8. The library has been updated. Execute the command associated with the **projlib:** dependency; namely

```
    @echo projlib up-to-date
```

It should be noted that to let **pfile1.o** have dependencies, the following syntax is required:

```
projlib(pfile1.o):        $(INCDIR)/stdio.h  pfile1.c
```

There is also a macro for referencing the archive member name when this form is used. The **$%** macro is evaluated each time **$@** is evaluated. If there is no current archive member, **$%** is null. If an archive member exists, then **$%** evaluates to the expression between the parenthesis.

Command Usage

The **make** command description is found under **make(1)**.

The make Command

The **make** command takes macro definitions, options, description filenames, and target filenames as arguments in the form:

make [*options*] [*macro definitions*] [*targets*]

The following summary of command operations explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files. Next, the option arguments are examined. The permissible options are as follows:

–i Ignore error codes returned by invoked commands. This mode is entered if the fake target name .IGNORE appears in the description file.

–s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name .SILENT appears in the description file.

–r Do not use the built-in rules.

–n No execute mode. Print commands, but do not execute them. Even lines beginning with an @ sign are printed.

–t Touch the target files (causing them to be up to date) rather than issue the usual commands.

–q Question. The **make** command returns a zero or nonzero status code depending on whether the target file is or is not up to date.

–p Print out the complete set of macro definitions and target descriptions.

–k Abandon work on the current entry if something goes wrong, but continue on other branches that do not depend on the current entry.

–e Environment variables override assignments within **makefiles**.

–f Description filename. The next argument is assumed to be the name of a description file. A filename of – denotes the standard input. If there are no –f arguments, the file named **makefile** or **Makefile** or **s.[mM]akefile** in the current directory is read. The contents of the description files override the built-in rules if they are present.

The following two arguments are evaluated in the same manner as flags:

.DEFAULT	If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name .DEFAULT are used if it exists.
.PRECIOUS	Dependents on this target are not removed when quit or interrupt is pressed.

Finally, the remaining arguments are assumed to be the names of targets to be made and the arguments are done in left-to-right order. If there are no such arguments, the first name in the description file that does not begin with a period is made.

Environment Variables

Environment variables are read and added to the macro definitions each time **make** executes. Precedence is a prime consideration in doing this properly. The following describes **make's** interaction with the environment. A macro, MAKEFLAGS, is maintained by **make**. The macro is defined as the collection of all input flag arguments into a string (without minus signs). The macro is exported and thus accessible to further invocations of **make**. Command line flags and assignments in the **makefile** update MAKEFLAGS. Thus, to describe how the environment interacts with **make**, the MAKEFLAGS macro (environment variable) must be considered.

When executed, **make** assigns macro definitions in the following order:

1. Read the MAKEFLAGS environment variable. If it is not present or null, the internal **make** variable MAKEFLAGS is set to the null string. Otherwise, each letter in MAKEFLAGS is assumed to be an input flag argument and is processed as such. (The only exceptions are the **–f**, **–p**, and **–r** flags.)

2. Read the internal list of macro definitions.

3. Read the environment. The environment variables are treated as macro definitions and marked as **exported** (in the shell sense).

4. Read the **makefile**(s). The assignments in the **makefile**(s) overrides the environment. This order is chosen so that when a **makefile** is read and executed, you know what to expect. That is, you get what is seen unless the **–e** flag is used. The **–e** is the line flag, which tells **make** to have the environment override the **makefile** assignments. Thus, if **make –e ...** is typed, the variables in the environment override the definitions in the **makefile**. Also MAKEFLAGS override the environment if assigned. This is useful for further invocations of **make** from the current **makefile**.

It may be clearer to list the precedence of assignments. Thus, in order from least binding to most binding, the precedence of assignments is as follows:

1. internal definitions
2. environment
3. **makefile**(s)
4. command line

The **–e** flag has the effect of rearranging the order to:

1. internal definitions
2. **makefile**(s)
3. environment
4. command line

This order is general enough to allow a programmer to define a makefile or set of **makefiles** whose parameters are dynamically definable.

Suggestions and Warnings

The most common difficulties arise from **make**'s specific meaning of dependency. If file **x.c** has a

```
#include "defs.h"
```

line, then the object file **x.o** depends on **defs.h**; the source file **x.c** does not. If **defs.h** is changed, nothing is done to the file **x.c** while file **x.o** must be recreated.

To discover what **make** would do, the **–n** option is very useful. The command

```
make –n
```

orders **make** to print out the commands that **make** would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be mild in character (e.g., adding a comment to an **include** file), the **–t** (touch) option can save a lot of time. Instead of issuing a large number of superfluous recompilations, **make** updates the modification times on the affected file. Thus, the command

```
make –ts
```

(touch silently) causes the relevant files to appear up to date. Obvious care is necessary because this mode of operation subverts the intention of **make** and destroys all memory of the previous relationships.

Internal Rules

The standard set of internal rules used by **make** are reproduced below.

```
#
#       SUFFIXES RECOGNIZED BY MAKE
#
.SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~ .h .h~ .sh .sh~ .f .f~
#
#       PREDEFINED MACROS
#
MAKE=make
AR=ar
ARFLAGS=-rv
AS=as
ASFLAGS=
CC=cc
CFLAGS=-O
F77=f77
F77FLAGS=
GET=get
GFLAGS=
LEX=lex
LFLAGS=
LD=ld
LDFLAGS=
YACC=yacc
YFLAGS=
```

Figure 13-2: **make** Internal Rules (Sheet 1 of 5)

```
#
#       SINGLE SUFFIX RULES
#

.c:
        $(CC) $(CFLAGS) $(LDFLAGS) $< -o $@
.c~:
        $(GET) $(GFLAGS) $<
        $(CC) $(CFLAGS) $(LDFLAGS) $*.c -o $*
        -rm -f $*.c
.f:
      * $(F77) $(F77FLAGS) $(LDFLAGS) $< -o $@
.f~:
        $(GET) $(GFLAGS) $<
        $(F77) $(F77FLAGS) $(LDFLAGS) $< -o $*
        -rm -f $*.f
.sh:
        cp $< $@; chmod 0777 $@
.sh~:
        $(GET) $(GFLAGS) $<
        cp $*.sh $*; chmod 0777 $@
        -rm -f $*.sh
```

Figure 13-2: make Internal Rules (Sheet 2 of 5)

```
#
#       DOUBLE SUFFIX RULES
#

.c~.c  .f~.f  .s~.s  .sh~.sh  .y~.y  .l~.l  .h~.h:
        $(GET) $(GFLAGS) $<

.c.a:
        $(CC) -c $(CFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.o

.c~.a:
        $(GET) $(GFLAGS) $<
        $(CC) -c $(CFLAGS) $*.c
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.[co]

.c.o:
        $(CC) $(CFLAGS) -c $<

.c~.o:
        $(GET) $(GFLAGS) $<
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c

.f.a:
        $(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
        $(AR) $(ARFLAGS) $@ $*.o
        -rm -f $*.o

.f~.a:
        $(GET) $(GFLAGS) $<
        $(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
        $(AR) $(ARFLAGS) $@ $*.o
        -rm -f $*.[fo]
```

Figure 13-2: make Internal Rules (Sheet 3 of 5)

```
.f.o:
        $(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
.f~.o:
        $(GET) $(GFLAGS) $<
        $(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
        -rm -f $*.f
.s~.a:
        $(GET) $(GFLAGS) $<
        $(AS) $(ASFLAGS) -o $*.o $*.s
        $(AR) $(ARFLAGS) $@ $*.o
        -rm -f $*.[so]
.s.o:
        $(AS) $(ASFLAGS) -o $@ $<
.s~.o:
        $(GET) $(GFLAGS) $<
        $(AS) $(ASFLAGS) -o $*.o $*.s
        -rm -f $*.s
.l.c :
        $(LEX) $(LFLAGS) $<
        mv lex.yy.c $@
.l~.c:
        $(GET) $(GFLAGS) $<
      • $(LEX) $(LFLAGS) $*.l
        mv lex.yy.c $@
```

Figure 13-2: make Internal Rules (Sheet 4 of 5)

```
.l.o:
        $(LEX) $(LFLAGS) $<
        $(CC) $(CFLAGS) -c lex.yy.c
        rm lex.yy.c
        mv lex.yy.o $@
        -rm -f $*.l
.l~.o:
        $(GET) $(GFLAGS) $<
        $(LEX) $(LFLAGS) $*.l
        $(CC) $(CFLAGS) -c lex.yy.c
        rm -f lex.yy.c $*.l
        mv lex.yy.o $*.o
.y.c :
        $(YACC) $(YFLAGS) $<
        mv y.tab.c $@
.y~.c :
        $(GET) $(GFLAGS) $<
        $(YACC) $(YFLAGS) $*.y
        mv y.tab.c $*.c
        -rm -f $*.y
.y.o:
        $(YACC) $(YFLAGS) $<
        $(CC) $(CFLAGS) -c y.tab.c
        rm y.tab.c
        mv y.tab.o $@
.y~.o:
        $(GET) $(GFLAGS) $<
        $(YACC) $(YFLAGS) $*.y
        $(CC) $(CFLAGS) -c y.tab.c
        rm -f y.tab.c $*.y
        mv y.tab.o $*.o
```

CADMUS added to the above rules a set of rules for pascal programs, which are very similar to the C rules.
Instead of .c we have .p, CC is PC, and CFLAGS is PFLAGS.

Figure 13-2: make Internal Rules (Sheet 5 of 5)

Table of Contents

Introduction

This chapter describes the symbolic debugger, **sdb**(1), as implemented for C language and Fortran 77 programs on the UNIX operating system and the debugger **adb**.

The **sdb** program is useful both for examining core images of aborted programs and for providing an environment in which execution of a program can be monitored and controlled.

The **sdb** program allows interaction with a debugged program at the source language level. When debugging a core image from an aborted program, **sdb** reports which line in the source program caused the error and allows all variables to be accessed symbolically and to be displayed in the correct format.

When executing, breakpoints may be placed at selected statements or the program may be single stepped on a line-by-line basis. To facilitate specification of lines in the program without a source listing, **sdb** provides a mechanism for examining the source text. Procedures may be called directly from the debugger. This feature is useful both for testing individual procedures and for calling user-provided routines, which provide formatted printouts of structured data.

The debugger **adb** also provides capabilities to examine "core" and other program files in a variety of formats, to run programs with embedded breakpoints, and to patch files. With the availability of the source oriented debugger **sdb**, **adb** has however become mostly obsolescent for the normal UNIX programmer. But in some circumstances, for example to look at binary files, to patch files or programs or to look at stripped programs, it can still be useful. This tutorial explains the various formatting options, techniques for debugging C programs, examples of printing file system information and patching.

Using sdb

In order to use **sdb** to its full capabilities, it is necessary to compile the source program with the **–g** option. This causes the compiler to generate additional information about the variables and statements of the compiled program. When the **–g** option has been specified, **sdb** can be used to obtain a trace of the called functions at the time of the abort and interactively display the values of variables.

A typical sequence of shell commands for debugging a core image is

```
cc –g prgm.c –o prgm
prgm
Bus error – core dumped
sdb prgm
main:25:        x[i] = 0;
*
```

The program **prgm** was compiled with the **–g** option and then executed. An error occurred, which caused a core dump. The **sdb** program is then invoked to examine the core dump to determine the cause of the error. It reports that the bus error occurred in function **main** at line 25 (line numbers are always relative to the beginning of the file) and outputs the source text of the offending line. The **sdb** program then prompts the user with an *, which shows that it is waiting for a command.

It is useful to know that **sdb** has a notion of current function and current line. In this example, they are initially set to **main** and 25, respectively.

Here **sdb** was called with one argument, **prgm**. In general, it takes three arguments on the command line. The first is the name of the executable file that is to be debugged; it defaults to **a.out** when not specified. The second is the name of the core file, defaulting to **core**; and the third is the list of the directories (separated by colons) containing the source of the program being debugged. The default is the current working directory. In the example, the second and third arguments defaulted to the correct values, so only the first was specified.

If the error occurred in a function that was not compiled with the **–g** option, **sdb** prints the function name and the address at which the error occurred. The current line and function are set to the first executable line in **main**. If **main** was not compiled with the **–g** option, **sdb** will print an error message, but debugging can continue for those routines that were compiled with the **–g** option.

Figure 15-1 at the end of the chapter, shows a more extensive example of **sdb** use.

Printing a Stack Trace

It is often useful to obtain a listing of the function calls that led to the error. This is obtained with the **t** command. For example:

```
*t
sub(x=2,y=3)          [prgm.c:25]
inter(i=16012)        [prgm.c:96]
main(argc=1,argv=0x7ffff54,envp=0x7ffff5c)  [prgm.c:15]
```

This indicates that the program was stopped within the function **sub** at line 25 in file **prgm.c**. The **sub** function was called with the arguments **x**=2 and **y**=3 from **inter** at line 96. The **inter** function was called from **main** at line 15. The **main** function is always called by a startup routine with three arguments often referred to as **argc**, **argv**, and **envp**. Note that **argv** and **envp** are pointers, so their values are printed in hexadecimal.

Examining Variables

The **sdb** program can be used to display variables in the stopped program. Variables are displayed by typing their name followed by a slash, so

```
*errflag/
```

causes **sdb** to display the value of variable **errflag**. Unless otherwise specified, variables are assumed to be either local to or accessible from the current function. To specify a different function, use the form

```
*sub:i/
```

to display variable **i** in function **sub**. FORTRAN 77 users can specify a common block variable in the same manner, provided it is on the call stack.

The **sdb** program supports a limited form of pattern matching for variable and function names. The symbol * is used to match any sequence of characters of a variable name and ? to match any single character. Consider the following commands

```
*x*/
*sub:y?/
**/
```

The first prints the values of all variables beginning with **x**, the second prints the values of all two letter variables in function **sub** beginning with **y**, and the last prints all variables. In the first and last examples, only variables accessible from the current function are printed. The command

```
**:*/
```

displays the variables for each function on the call stack.

The **sdb** program normally displays the variable in a format determined by its type as declared in the source program. To request a different format, a specifier is placed after the slash. The specifier consists of an optional length specification followed by the format. The length specifiers are:

b one byte

h two bytes (half word)

l four bytes (long word)

The length specifiers are effective only with the formats **d**, **o**, **x**, and **u**. If no length is specified, the word length of the host machine is used. A number can be used with the **s** or **a** formats to control the number of characters printed. The **s** and **a** formats normally print characters until either a null is reached or 128 characters have been printed. The number specifies exactly how many characters should be printed.

There are a number of format specifiers available:

c character

d decimal

u decimal unsigned

o octal

x hexadecimal

f 32-bit single-precision floating point

g 64-bit double-precision floating point

s Assume variable is a string pointer and print characters starting at the address pointed to by the variable until a null is reached.

a Print characters starting at the variable's address until a null is reached.

p Pointer to function.

i Interpret as a machine-language instruction.

For example, the variable **i** can be displayed with

```
*i/x
```

which prints out the value of **i** in hexadecimal.

sdb also knows about structures, arrays, and pointers so that all of the following commands work.

```
*array[2][3]/
*sym.id/
*psym->usage/
*xsym[20].p->usage/
```

The only restriction is that array subscripts must be numbers. Note that as a special case:

```
*psym[0]
```

displays the structure pointed to by **psym** in decimal.

Core locations can also be displayed by specifying their absolute addresses. The command

```
*1024/
```

displays location 1024 in decimal. As in C language, numbers may also be specified in octal or hexadecimal so the above command is equivalent to both

```
*02000/
```

and

```
*0x400/
```

It is possible to mix numbers and variables so that

```
*1000.x/
```

refers to an element of a structure starting at address 1000, and

```
*1000->x/
```

refers to an element of a structure whose address is at 1000. For commands of the type *1000.x/ and *1000->x/, the **sdb** program uses the structure template of the last structured referenced.

The address of a variable is printed with =, so

```
*i=
```

displays the address of **i**. Another feature whose usefulness will become apparent later is the command

```
*./
```

which redisplays the last variable typed.

Source File Display and Manipulation

The **sdb** program has been designed to make it easy to debug a program without constant reference to a current source listing. Facilities are provided that perform context searches within the source files of the program being debugged and that display selected portions of the source files. The commands are similar to those of the UNIX system text editor **ed**(1). Like the editor, **sdb** has a notion of current file and line within the current file. **sdb** also knows how the lines of a file are partitioned into functions, so it also has a notion of current function. As noted in other parts of this document, the current function is used by a number of **sdb** commands.

Displaying the Source File

Four commands exist for displaying lines in the source file. They are useful for perusing the source program and for determining the context of the current line. The commands are:

p	Prints the current line.
w	Window; prints a window of ten lines around the current line.
z	Prints ten lines starting at the current line. Advances the current line by ten.
control-d	Scrolls; prints the next ten lines and advances the current line by ten. This command is used to cleanly display long segments of the program.

When a line from a file is printed, it is preceded by its line number. This not only gives an indication of its relative position in the file, but it is also used as input by some **sdb** commands.

Changing the Current Source File or Function

The **e** command is used to change the current source file. Either of the forms

```
*e function
*e file.c
```

may be used. The first causes the file containing the named function to become the current file, and the current line becomes the first line of the function. The other form causes the named file to become current. In this case, the current line is set to the first line of the named file. Finally, an **e** command with no argument causes the current function and file named to be printed.

Changing the Current Line in the Source File

The **z** and **control-d** commands have a side effect of changing the current line in the source file. The following paragraphs describe other commands that change the current line.

There are two commands for searching for instances of regular expressions in source files. They are

```
*/regular expression/
*?regular expression?
```

The first command searches forward through the file for a line containing a string that matches the regular expression and the second searches backwards. The trailing / and ? may be omitted from these commands. Regular expression matching is identical to that of **ed**(1).

The + and – commands may be used to move the current line forward or backward by a specified number of lines. Typing a new-line advances the current line by one, and typing a number causes that line to become the current line in the file. These commands may be combined with the display commands so that

```
*+15z
```

advances the current line by 15 and then prints ten lines.

A Controlled Environment for Program Testing

One very useful feature of **sdb** is breakpoint debugging. After entering **sdb**, breakpoints can be set at certain lines in the source program. The program is then started with an **sdb** command. Execution of the program proceeds as normal until it is about to execute one of the lines at which a breakpoint has been set. The program stops and **sdb** reports the breakpoint where the program stopped. Now, **sdb** commands may be used to display the trace of function calls and the values of variables. If the user is satisfied the program is working correctly to this point, some breakpoints can be deleted and others set; then program execution may be continued from the point where it stopped.

A useful alternative to setting breakpoints is single stepping. **sdb** can be requested to execute the next line of the program and then stop. This feature is especially useful for testing new programs, so they can be verified on a statement-by-statement basis. If an attempt is made to single step through a function that has not been compiled with the **–g** option, execution proceeds until a statement in a function compiled with the **–g** option is reached. It is also possible to have the program execute one machine level instruction at a time. This is particularly useful when the program has not been compiled with the **–g** option.

Setting and Deleting Breakpoints

Breakpoints can be set at any line in a function compiled with the **–g** option. The command format is:

```
*12b
*proc:12b
*proc:b
*b
```

The first form sets a breakpoint at line 12 in the current file. The line numbers are relative to the beginning of the file as printed by the source file display commands. The second form sets a breakpoint at line 12 of function **proc**, and the third sets a breakpoint at the first line of **proc**. The last sets a breakpoint at the current line.

Breakpoints are deleted similarly with the **d** command:

```
*12d
*proc:12d
*proc:d
```

In addition, if the command **d** is given alone, the breakpoints are deleted interactively. Each breakpoint location is printed, and a line is read from the user. If the line begins with a **y** or **d**, the breakpoint is deleted.

A list of the current breakpoints is printed in response to a **B** command, and the **D** command deletes all breakpoints. It is sometimes desirable to have **sdb** automatically perform a sequence of commands at a breakpoint and then have execution continue. This is achieved with another form of the **b** command.

```
*12b t;x/
```

causes both a trace back and the value of *x* to be printed each time execution gets to line 12. The **a** command is a variation of the above command. There are two forms:

```
*proc:a
*proc:12a
```

The first prints the function name and its arguments each time it is called, and the second prints the source line each time it is about to be executed. For both forms of the **a** command, execution continues after the function name or source line is printed.

Running the Program

The **r** command is used to begin program execution. It restarts the program as if it were invoked from the shell. The command

```
*r args
```

runs the program with the given arguments as if they had been typed on the shell command line. If no arguments are specified, then the arguments from the last execution of the program within **sdb** are used. To run a program with no arguments, use the **R** command.

After the program is started, execution continues until a breakpoint is encountered, a signal such as INTERRUPT or QUIT occurs, or the program terminates. In all cases after an appropriate message is printed, control returns to the user.

The **c** command may be used to continue execution of a stopped program. A line number may be specified, as in:

```
*proc:12c
```

This places a temporary breakpoint at the named line. The breakpoint is deleted when the **c** command finishes. There is also a **C** command that continues but passes the signal that stopped the program back to the program. This is useful for testing user-written signal handlers. Execution may be continued at a specified line with the g command. For example:

```
*17 g
```

continues at line 17 of the current function. A use for this command is to avoid executing a section of code that is known to be bad. The user should not attempt to continue execution in a function different than that of the breakpoint.

The s command is used to run the program for a single statement. It is useful for slowly executing the program to examine its behavior in detail. An important alternative is the S command. This command is like the s command but does not stop within called functions. It is often used when one is confident that the called function works correctly but is interested in testing the calling routine.

The **i** command is used to run the program one machine level instruction at a time while ignoring the signal that stopped the program. Its uses are similar to the s command. There is also an **I** command that causes the program to execute one machine level instruction at a time, but also passes the signal that stopped the program back to the program.

Calling Functions

It is possible to call any of the functions of the program from **sdb**. This feature is useful both for testing individual functions with different arguments and for calling a user-supplied function to print structured data. There are two ways to call a function:

```
*proc(arg1, arg2, . . .)
*proc(arg1, arg2, . . .)/m
```

The first simply executes the function. The second is intended for calling functions (it executes the function and prints the value that it returns). The value is printed in decimal unless some other format is specified by *m*. Arguments to functions may be integer, character or string constants, or variables that are accessible from the current function.

An unfortunate bug in the current implementation is that if a function is called when the program is not stopped at a breakpoint (such as when a core image is being debugged) all variables are initialized before the function is started. This makes it impossible to use a function that formats data from a dump.

Machine Language Debugging

The **sdb** program has facilities for examining programs at the machine language level. It is possible to print the machine language statements associated with a line in the source and to place breakpoints at arbitrary addresses. The **sdb** program can also be used to display or modify the contents of the machine registers.

Displaying Machine Language Statements

To display the machine language statements associated with line 25 in function **main**, use the command

```
*main:25?
```

The **?** command is identical to the **/** command except that it displays from text space. The default format for printing text space is the **i** format, which interprets the machine language instruction. The **control-d** command may be used to print the next ten instructions.

Absolute addresses may be specified instead of line numbers by appending a **:** to them so that

```
*0x1024:?
```

displays the contents of address 0x1024 in text space. Note that the command

```
*0x1024?
```

displays the instruction corresponding to line 0x1024 in the current function. It is also possible to set or delete a breakpoint by specifying its absolute address;

```
*0x1024:b
```

sets a breakpoint at address 0x1024.

Manipulating Registers

The **x** command prints the values of all the registers. Also, individual registers may be named by appending a **%** sign to their name so that

```
*r3%
```

displays the value of register **r3**.

Other Commands

To exit **sdb**, use the **q** command.

The **!** command (when used immediately after the * prompt) is identical to that in **ed**(1) and is used to have the shell execute a command. The **!** can also be used to change the values of variables or registers when the program is stopped at a breakpoint. This is done with the command

```
*variable!value
*r3!value
```

which sets the variable or the named register to the given value. The value may be a number, character constant, register, or the name of another variable. If the variable is of type **float** or **double**, the value can also be a floating-point constant (specified according to the standard C language format).

An sdb Session

An example of a debugging session using **sdb** is shown in Figure 15-1. Comments (preceded by a pound sign, #) have been added to help you see what is happening.

```
sdb myoptim - .:../common  # enter sdb command
Source path: .:../common
No core image
*window:b                   # set a breakpoint at start of window
0x80802462 (window:1459+2) b
*r < m.s > out.m.s          # run the program
Breakpoint at
0x80802462 in window:1459: window(size, func) register int size;
boolean(*func)(); {
*t                          # print stack trace
window(size=2,func=w2opt)    [optim.c:1459]
peep()    [peep.c:34]
pseudo(s=.def^Imain;^I.val^I.;^I.scl^I-1;^I.endef)    [local.c:483]
yylex()    [local.c:229]
main(argc=0,argv=0xc00201bc,-1073610300)    [optim.c:227]
```

Figure 14-1: Example of sdb Usage (Sheet 1 of 3)

```
*z                          # print 10 lines of source
1459: window(size, func) register int size; boolean (*func)(); {
1460:
1461:   extern NODE *initw();
1462:   register NODE *pl;
1463:   register int i;
1464:
1465:   TRACE(window);
1466:
1467:   /* find first window */
1468:
*s                          # step
window:1459: window(size, func) register int size; boolean (*func)(); {
*s                          # step
window:1465:    TRACE(window);
*s                          # step
window:1469:    wsize = size;
*s                          # step
window:1470:    if ((pl = initw(n0.forw)) == NULL)
*S                          # step through procedure call
window:1475:    for (opf = pf->back; ; opf = pf->back) {
*pl                         # show variable pl
0x80886b38
*x                          # print the register contents
  r0/ 0x80886b38            r1/ 0                  r2/ 0x8088796c
  r3/ 0x80885830            r4/ 0xc0020470         r5/ 0xc00203f0
  r6/ 0xc0020478            r7/ 0x80886b38         r8/ 2
  ap/ 0xc00202dc            fp/ 0xc0020308         sp/ 0xc0020308
 psw/ 0x201f73              pc/ 0x808024b0
0x808024b0 (window:1475):       MOVW    0x80880d8c,%r0   [-0x7f77f274,%r0]
```

Figure 14-1: Example of sdb Usage (Sheet 2 of 3)

```
*pl[0]                          # dereference the pointer
pl[0].forw/ 0x80886b6c
pl[0].back/ 0x80886ac8
pl[0].ops[0]/ pushw
pl[0].uniqid/ 0
pl[0].op/ 123
pl[0].nlive/ 3588
pl[0].ndead/ 4096
*pl->forw[0]                    # dereference the pointer
pl->forw[0].forw/ 0x80886ca0
pl->forw[0].back/ 0x80886b38
pl->forw[0].ops[0]/ call
pl->forw[0].uniqid/ 0
pl->forw[0].op/ 9
pl->forw[0].nlive/ 3584
pl->forw[0].ndead/ 4099
*pl!pl->forw                    # replace pl with pl->forw
*pl                             # show pl
0x80886b6c
*c                              # continue
Breakpoint at
0x80802462 in window:1459: window(size, func) register int size;
boolean (*func)(); {
*s                              # step
window:1459: window(size, func) register int size; boolean (*func)(); {
*s                              # step
window:1465:    TRACE(window);
*size                           # show function argument size
3
*D                              # delete all breakpoints
All breakpoints deleted
*c                              # continue
Process terminated
*q                              # quit sdb
$
```

Figure 14-1: Example of sdb Usage (Sheet 3 of 3)

Using adb

A Quick Survey

Invocation

adb is invoked as:

```
adb objfile corefile
```

where *objfile* is an executable UNIX file and *corefile* is a core image file. Many times this will look like:

```
adb a.out core
```

or more simply:

```
adb
```

where the defaults are *a.out* and *core* respectively. The filename minus (–) means ignore this argument as in:

```
adb - core
```

adb has requests for examining locations in either file. The **?** request examines the contents of *objfile*, the **/** request examines the *corefile*. The general form of these requests is:

```
address ? format
```

or

```
address / format
```

Current Address

adb maintains a current address, called dot, similar in function to the current pointer in the UNIX editor. When an address is entered, the current address is set to that location, so that:

```
#126?i
```

sets dot to hex 126 and prints the instruction at that address.

The request:

```
.,10/d
```

prints 10 decimal numbers starting at dot. Dot ends up referring to the address of the last item printed. When used with the ? or / requests, the current address can be advanced by typing newline; it can be decremented by typing ^.

Addresses are represented by expressions. Expressions are made up from decimal, octal, and hexadecimal integers, and symbols from the program under test. These may be combined with the operators +, –, *, % (integer division), & (bitwise and), | (bitwise inclusive or), # (round up to the next multiple), and (not) (internal arithmetic uses 32 bits). When typing a symbolic address for a C program, the user can type *name* or *_name;* **adb** will recognize both forms.

Formats

To print data, a user specifies a collection of letters and characters that describe the format of the printout. Formats are "remembered" in the sense that typing a request without one will cause the new printout to appear in the previous format. The following are the most commonly used format letters. Words, in this document, are 16 bit words.

b	one byte in octal
c	one byte as a character
x	one word in hexadecimal
o	one word in octal
d	one word in decimal
e	two words in floating point (old ffp format)
f	two words in floating point (IEEE 754 single prec.)
i	MC68020 instruction
s	a null terminated character string
a	the value of dot
u	one word as unsigned integer
n	print a newline
r	print a blank space
^	backup dot

Format letters are also available for "long" values, for example, '**D**' for long decimal, '**X**' for long hex, and '**F**' for double floating point (IEEE 754 double precision). For other formats see the **adb** manual page.

General Request Meanings

The general form of a request is:

```
address,count command modifier
```

which sets 'dot' to *address* and executes the command *count* times.

The following table illustrates some general **adb** command meanings:

Command	**Meaning**
?	Print contents from *a.out* file
/	Print contents from *core* file
=	Print value of "dot"
:	Breakpoint control
$	Miscellaneous requests
;	Request separator
!	Escape to shell

adb catches signals, so a user cannot use a quit signal to exit from **adb**. The request $q or $Q (or crtl-Z) must be used to exit from **adb**.

Debugging C Programs

Debugging A Core Image

Consider the C program in Figure 1. The program is used to illustrate a common error made by C programmers. The object of the program is to change the lower case "t" to upper case in the string pointed to by *charp* and then write the character string to the file indicated by argument 1. The bug shown is that the character "T" is stored in the pointer *charp* instead of the string pointed to by *charp*. If the program has been compiled with the -z option, to catch zero pointer references, executing the program produces a core file because of an out of bounds memory reference. It must also be compiled with the -g option, if information about parameters or local variables is desired.

adb is invoked by:

```
adb a.out core
```

The first debugging request:

```
$c
```

is used to give a C backtrace through the subroutines called. As shown in Figure 2 only one function (*main*) was called and the arguments *argc* and *argv* have hex

values #2 and #3f7feca6 respectively. Both of these values look reasonable; #2 = two arguments, #3f7feca6 = address on stack of parameter vector.
The next request:

```
$C
```

is used to give a C backtrace plus an interpretation of all the local variables in each function and their values in hex. Note that of the value for *cc* only the first byte is significant, because *cc* is declared as char. Were it declared as short, only the first two bytes would be significant. If however *cc* would have been declared as register char or register short, the name *cc* would have been prefixed with a '<', the shown value would be the value of the register, and the significant byte or word would be the last byte or word of the value!

The next request:

```
$r
```

prints out the registers including the program counter and an interpretation of the instruction at that location.

The request:

```
$e
```

prints out the values of all external variables.

A map exists for each file handled by **adb**. The map for the *a.out* file is referenced by ? whereas the map for *core* file is referenced by /. Furthermore, a good rule of thumb is to use ? for instructions and / for data when looking at programs. To print out information about the maps type:

```
$m
```

This produces a report of the contents of the maps. More about these maps later.

In our example, it is useful to try to see the contents of the string pointed to by *charp*. This is done by:

```
*charp/s
```

which says use *charp* as a pointer in the *core* file and print the information as a character string. This gives the message 'data address not found', because #55 is not a valid data address. Using **adb** similarly, we could print information about the arguments to a function. The request:

```
main.argc/D
```

prints the long decimal *core* image value of the argument *argc* in the function *main*.
The request:

```
*main.argv/3X
```

prints the long hex values of the three consecutive pointers pointed to by *argv* in the function *main*. Note that these values are the addresses of the arguments to main. Therefore:

```
#3f7fed30/s
```

prints the ASCII value of the first argument. Another way to print this value would have been

```
*"/s
```

The " means ditto which remembers the last address typed, in this case *main.argc* ; the * instructs **adb** to use the address field of the *core* file as a pointer.

The request:

```
.=X
```

prints the current address (not its contents) in long hex which has been set to the address of the first argument. The current address, dot, is used by **adb** to "remember" its current location. It allows the user to reference locations relative to the current address, for example:

```
.-10/d
```

Multiple Functions

Consider the C program illustrated in Figure 3. This program calls functions *f,g,* and *h* until the stack is exhausted and a core image is produced. Note that on a virtual system the stack may grow very large!

Again you can enter the debugger via:

```
adb
```

which assumes the names *a.out* and *core* for the executable file and core image file respectively. The request:

```
$c
```

will fill a page of backtrace references to *f,g,* and *h*. Figure 4 shows an abbreviated list (typing *Ctrl-C* will terminate the output and bring you back to **adb** request level).

The request:

```
,5$C
```

prints the five most recent activations.

Notice that each function (*f,g,h*) has a counter of the number of times it was called.

The request:

```
fcnt/D
```

prints the decimal value of the counter for the function *f*. Similarly *gcnt* and *hcnt* could be printed. To print the value of an automatic variable, for example the decimal value of *x* in the last call of the function *h*, type:

```
h.x/D
```

It is not possible to print stack frames other than the most recent activation of a function. Therefore, a user can print everything with *$C* or the occurrence of a variable in the most recent call of a function. It is possible with the *$C* request, however, to print the stack frame starting at some address as *address$C*.

Setting Breakpoints

Consider the C program in Figure 5. This program, which changes tabs into blanks, is adapted from *Software Tools* by Kernighan and Plauger, pp. 18-27.

We will run this program under the control of **adb** (see Figure 6) by:

```
adb a.out -
```

Breakpoints are set in the program as:

```
address:b  [request]
```

The requests:

```
settab+4:b
fopen+4:b
getc+4:b
tabpos+4:b
```

set breakpoints at the start of these functions, except for getc, which is called as a function, but in reality a macro defined in <stdio.h>. The above addresses are entered as

```
symbol+4
```

so that they will appear in any C backtrace since the first instruction of each function is the 68020 LINK instruction. If you want to see the correct values of the register variables, if any, in the topmost procedure, you have to set the breakpoint after the MOVEM instruction which comes second. Note that some of the functions are from the C library.

To print the location of breakpoints one types:

```
$b
```

The display indicates a *count* field. A breakpoint is bypassed *count* −1times before causing a stop. The *command* field indicates the **adb** requests to be executed each time the breakpoint is encountered. In our example no *command* fields are present.

By displaying the original instructions at the function *settab* we see that the breakpoint is set after the LINK instruction. We can display the instructions using the **adb** request:

```
settab,5?ia
```

This request displays five instructions starting at *settab* with the addresses of each location displayed. Another variation is:

```
settab,5?i
```

which displays the instructions with only the starting address.

Notice that we accessed the addresses from the *a.out* file with the **?** command. In general when asking for a printout of multiple items, **adb** will advance the current address the number of bytes necessary to satisfy the request; in the above example five instructions were displayed and the current address was advanced 22 (decimal) bytes.

To run the program one simply types:

```
:r
```

To delete a breakpoint, for instance the entry to the function *settab*, one types:

```
settab+4:d
```

To continue execution of the program from the breakpoint type:

```
:c
```

Once the program has stopped (in this case at the breakpoint for *fopen)*, **adb** requests can be used to display the contents of memory. For example:

```
$C
```

to display a stack trace, or:

```
tabs,3/8d
```

to print three lines of 8 locations each from the array called *tabs*. By this time (at location *fopen)* in the C program, *settab* has been called and should have set a one in every eighth location of *tabs*.

Advanced Breakpoint Usage

We can set a breakpoint at the end of a procedure by searching for the UNLK instruction at the end of it, which has the opcode #4e5e. The command

```
tabpos?l #4e5e
```

searches from the start address of tabpos until it finds the value #4e5e. This value is reported to be at location tabpos+#22, and dot is set to this address. With :b we set a breakpoint to dot, and with :c we continue. When we reach the breakpoint, we can look at the value tabpos returns, by displaying the contents of register D0.

The UNIX quit and interrupt signals act on **adb** itself rather than on the program being debugged. If such a signal occurs then the program being debugged is stopped and control is returned to **adb**. The signal is saved by **adb** and is passed on to the test program if:

```
:c
```

is typed. This can be useful when testing interrupt handling routines. The signal is not passed on to the test program if:

```
:c 0
```

is typed.

A breakpoint can be overwritten without first deleting the old breakpoint. For example:

```
settab+4:b  settab,5?ia; ptab/x
```

could be entered after typing the above requests.

Other Breakpoint Facilities

- Arguments and change of standard input and output are passed to a program as:

  ```
  :r arg1  arg2 ... <infile  >outfile
  ```

  This request
  kills any existing program under test and
  starts the
  a.out afresh.

- The program being debugged can be single stepped by:

  ```
  :s
  ```

  If necessary, this request will start up the program being

debugged and stop after executing
the first instruction.

- adb allows a program to be entered at a specific address by typing:

```
address:r
```

- The count field can be used to skip the first *n* breakpoints as:

```
,n:r
```

  The request:

```
,n:c
```

  may also be used for skipping the first *n* breakpoints
  when continuing a program.

- A program can be continued at an address different from the breakpoint by:

```
address:c
```

- The program being debugged runs as a separate process and can be killed by:

```
:k
```

Maps

UNIX knows several object file formats. These are used to tell the loader how to load the program file. File type 0407 is the format of object modules generated by a C compiler invocation such as cc -c pgm.c. A 0410 or 0413 file is produced by the loader, when all references are resolved. This is the format of all executable files. **adb** provides access to the program or module through a set of maps. To print the maps type:

```
$m
```

The b, e, and f fields are used by **adb** to map addresses into file addresses. The "b" and "e" fields are the starting and ending locations for a segment, i.e. logical addresses. When an address A followed by ? or / is given, adb goes to the specified map, and sees if A lies in the interval [b1,e1] or [b2,e2]. If yes, it subtracts the corresponding b value from A and adds the corresponding f value, to obtain the physical address in the file. More formally, given an address, A, the location in the file (either *a.out* or *core*) is calculated as:

$b1 \le A \le e1 \Rightarrow$ file address = (A-b1)+f1
$b2 \le A \le e2 \Rightarrow$ file address = (A-b2)+f2

In the normal case, the first segment of the ? map corresponds to the a.out text, the second to the a.out data. The first segment of the / map corresponds to the core data segment, and the second segment to the core stack segment. A core file does not include the text of the crashed program. Note that data can be accessed in two ways. However, the a.out data segment is the data segment as it looked at load time. It also contains only the initialized data. The core data segment is the data at the time of the core dump. Note that the b2 value of the / map is the lower limit of the stack.

A user can access locations by using the **adb** defined variables. The **$v** request prints the variables initialized by **adb** (Figure 8):

b	base address of data segment
d	length of the data segment
e	entry point of the program
m	execution type (0407,0410,0411)
s	length of the stack
t	length of the text

Use can be made of these variables by expressions such as:

```
<s
```

in the address field. Similarly the value of the variable can be changed by an assignment request such as:

```
02000>b
```

that sets **b** to octal 2000. These variables are useful to know if the file under examination is an executable or *core* image file.

adb reads the header of the *core* image file to find the values for these variables. If the second file specified does not seem to be a *core* file, then standard values for identical mapping are used instead.

Advanced Usage

It is possible with **adb** to combine formatting requests to provide elaborate displays. Below are several examples.

Formatted dump

The line:

```
<b,-1/4x4^8Cn
```

prints 4 hex words followed by their ASCII interpretation from the data space of. the core image file. Broken down, the various request pieces mean:

- **<b**
  The base address of the data segment.
- **<b,–1**
  Print from the base address to the end of file. A negative count is used here and elsewhere to loop indefinitely or until some error condition (like end of file) is detected.

  The format **4x4^8Cn** is broken down as follows:
- **4x**
  Print 4 hex locations.
- **4^**
  Backup the current address 4 locations (to the original start of the field).
- **8C**
  Print 8 consecutive characters using an escape convention; each character in the range 0 to 037 is printed as @ followed by the corresponding character in the range 0140 to 0177. An @ is printed as @@.
- **n**
  Print a newline.

The request:

```
<b,<d/4x4^8Cn
```

could have been used instead to allow the printing to stop at the end of the data segment (<d provides the data segment size in bytes).

The formatting requests can be combined with **adb**'s ability to read in a script to produce a core image dump script. **adb** is invoked as:

```
adb a.out core < dump
```

to read in a script file, *dump*, of requests. An example of such a script is:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b,-1/8xna
```

adb attempts to print addresses as:

```
symbol + offset
```

The request **4095$s** sets the maximum permissible offset to the nearest symbolic address to 4095 (default 32767). The request = can be used to print literal strings. Thus, headings are provided in this *dump* program with requests of the form:

```
=3n"C Stack Backtrace"
```

that spaces three lines and prints the literal string. The request **$v** prints all non-zero **adb** variables (see Figure 8). The request **0$s** sets the maximum offset for symbol matches to zero thus suppressing the printing of symbolic labels in favor of hex values. Note that this is only done for the printing of the data segment. Try dumping the data segment without changing the maximum offset for a comparison. The request:

```
<b,-1/8xna
```

prints a dump from the base of the data segment to the end of file with a hex address field and eight hex numbers per line.

Figure 10 shows the results of some formatting requests on the C program of Figure 9.

Directory Dump

As another illustration (Figure 11) consider a set of requests to dump the contents of a directory (which is made up of an integer *inumber* followed by a 14 character name):

```
adb dir -
=n8t"Inum"8t"Name"
0,-1? u8t14cn
```

In this example, the **u** prints the *inumber* as an unsigned decimal integer, the **8t** means that **adb** will space to the next multiple of 8 on the output line, and the **14c** prints the 14 character file name.

Ilist Dump

Similarly the contents of the *ilist* of a file system, (e.g. /dev/sw0.0, see inode(4)) could be dumped with the following set of requests:

```
adb /dev/sw0.0 -
2048,-1?"flags"8ton"links,uid,gid"8t3on"size"8tDn"addr"8t10Xn"times"8t3Y2na
```

```
adb dir -
=n8t"Inum"8t"Name"
0,-1? u8t14cn
```

In this example the dump begins at location 2048, since that is the start of an *ilist* within a 1024 byte per block file system. The last access time, last modify time and creation time are printed with the **3Y** operator. Figure 11 shows portions of these requests as applied to a directory and file system.

Converting values

adb may be used to convert values from one representation to another. For example:

```
072 = odx
```

will print

```
072                                58#3a
```

which are the octal, decimal and hexadecimal representations of 072 (octal). The format is remembered so that typing subsequent numbers will print them in the given formats. Character values may be converted similarly, for example:

```
'a' = co
```

prints

```
a                                  0141
```

It may also be used to evaluate expressions but be warned that all binary operators have the same precedence which is lower than that for unary operators.

Patching

Patching files with **adb** is accomplished with the *write,* **w** or **W**, request (which is not like the *ed* editor write command). This is often used in conjunction with the *locate,* **l** or **L** request. In general, the request syntax for **l** and **w** are similar as follows:

```
?l value
```

The request **l** is used to match on two bytes, **L** is used for four bytes. The request **w** is used to write two bytes, whereas **W** writes four bytes. The value field in either *locate* or *write* requests is an expression. Therefore, decimal, octal and hex numbers, or character strings are supported.

In order to modify a file, **adb** must be called as:

```
adb -w file1
```

When called with this option, *file1* is created if necessary and opened for both reading and writing.

For example, consider the C program shown in Figure 9. We can change the word "This" to "The " in the executable file for this program, *ex7*, by using the following requests:

```
adb -w ex7 -
<b?l 'Th'
?W 'The '
```

The request **?l** starts at dot and stops at the first match of "Th" having set dot to the address of the location found. Note the use of **?** to write to the *a.out* file.

More frequently the request will be typed as:

```
?l 'Th'; ?s
```

and locates the first occurrence of "Th" and print the entire string. Execution of this **adb** request will set dot to the address of the "Th" characters.

As another example of the utility of the patching facility, consider a C program that has an internal logic flag. The flag could be set by the user through **adb** and the program run. For example:

```
adb a.out -
:s arg1 arg2
flag/w 1
:c
```

The **:s** request is normally used to single step through a process or start a process in single step mode. In this case it starts *a.out* as a subprocess with arguments **arg1** and **arg2**. If there is a subprocess running **adb** writes to it rather than to the file so the **w** request causes *flag* to be changed in the memory of the subprocess.

```
#include <stdio.h>

char *charp = "this is a sentence.";
FILE *obuf;

main(argc,argv)
int argc;
char **argv;
{
                        charcc;

                        if(argc < 2) {
                        printf("Output file argument missing\n);
                        exit(1);
                        }

                        if((obuf = fopen(argv[1],"w")) == NULL){
                        printf("%s : not found\n, argv[1]);
                        exit(1);
                        }
                        cc = 'X';       /* for DEMO purpose only */
                        charp = 'T';    /* BUG: should be *charp = 'T';  !! */
                        while(cc = *charp++)
                        putc(cc,obuf);
                        fclose(obuf);
}
```

Figure 14-2: C program with pointer bug

```
$ cc -g -z ptr.c
$ a.out outfile
Bus error - core dumped
$ adb a.out core
$c
_main($0000,$0002,$3f7f,$eca6,$3f7f,$ecb2) line0

$C
_main($0000,$0002,$3f7f,$eca6,$3f7f,$ecb2) line0
                         _argc:$00000002
                         _argv:$3f7feca6
                         _cc:$58000000

$r
d0                       $001010aa    __iob+$002a
d1                       $00000004
d2                       $00000000
d3                       $00000000
d4                       $00000000
d5                       $00000000
d6                       $00000000
d7                       $00000000
a0                       $00000054
a1                       $3f7febd0
a2                       $3f7fef3c
a3                       $00000000
a4                       $00000000
a5                       $00000000
a6                       $3f7fec64
a7                       $3f7fec44
```

Figure 14-3: adb output for C program of Figure 14-2 (Sheet 1 of 3)

```
ns                      $00020151
ac                      $00000054
ip                      $00190000
pc                      $000011a8   _main+$0084
_main+$0084:            move.b (A0),(-29,A6)   $1d50$ffe3

$e
_environ:                   $3f7fecb2
_charp:                     $00000055
_obuf:                      $001010aa
__dbargs:                   $00000000
__ctype_:                   $00202020
__iob:                      $00000000
__lastbuf:                  $0010133c
__sibuf:                    $00000000
__sobuf:                    $00000000
__smbuf:                    $00000000
__bufendtab:                $00000000
_errno:                     $00000000
__currbrk:                  $00102060

$m
? map                       'a.out'
b1 = $000010a8               e1= $00002e0c  f1 = $000000a8
b2 = $00100e0c               e2= $0010149c  f2 = $00001e0c
/ map                       'core'
b1 = $00100000               e1= $00103000  f1 = $00000122
b2 = $3f7fd000               e2= $3f7ff000  f2 = $00003122
```

Figure 14-3: adb output for C program of Figure 14-2 (Sheet 2 of 3)

```
charp/X
_charp:                     #00000055

*charp/s
#00000055:
data address not found

main.argc/d
$3f7fec6c:                  2

*main.argv/3X
$3f7feca6:                  $3f7fed30$3f7fed36$00000000

#3f7fed30/s
$3f7fed30:                  a.out

*main.argv/3X
$3f7feca6:                  $3f7fed30$3f7fed36$00000000

*"/s
$3f7fed30:                  a.out

 .=X
                            $3f7fed30

$q
```

Figure 14-3: adb output for C program of Figure 14-2 (Sheet3 of 3)

```
int                     fcnt,gcnt,hcnt;
h(x,y)
{
                        int hi; register int hr;
                        hi = x+1;
                        hr = x-y+1;
                        hcnt++ ;
                        hj:
                        f(hr,hi);
}

g(p,q)
{
                        int gi; register int gr;
                        gi = q-p;
                        gr = q-p+1;
                        gcnt++ ;
                        gj:
                        h(gr,gi);
}

f(a,b)
{
                        int fi; register int fr;
                        fi = a+2*b;
                        fr = a+b;
                        fcnt++ ;
                        fj:
                        g(fr,fi);
}

main()
{
                        f(1,1);
}
```

Figure 14-4: Multiple function C program for stack trace illustration

```
$ adb
Program                         killed by bus error
$c
_h($0000,$aa9e,$0000,$aa9d) line 0
_g($0000,$aa9f,$0001,$553c) line 0
_f($0000,$0002,$0000,$aa9d) line 0
_h($0000,$aa9c,$0000,$aa9b) line 0
_g($0000,$aa9d,$0001,$5538) line 0
_f($0000,$0002,$0000,$aa9b) line 0
_h($0000,$aa9a,$0000,$aa99) line 0
_g($0000,$aa9b,$0001,$5534) line 0
_f($0000,$0002,$0000,$aa99) line 0
_h($0000,$aa98,$0000,$aa97) line 0
_g($0000,$aa99,$0001,$5530) line 0

HIT INTR KEY
adb

,5$C
_h($0000,$aa9e,$0000,$aa9d) line 0
                            _x:$0000aa9e
                            _y:$0000aa9d
                            _hi:?
                                    <_hr:$0000aa9e
_g($0000,$aa9f,$0001,$553c) line 0
                            _p:$0000aa9f
                            _q:$0001553c
                            _gi:$0000aa9d
                                    <_gr:?
```

Figure 14-5: adb output for C program of Figure 14-4 (Sheet 1 of 2)

```
_f($0000,$0002,$0000,$aa9d) line 0
                            _a:$00000002
                            _b:$0000aa9d
                            _fl:$0001553c
                                    <_fr:$0000aa9f
_h($0000,$aa9c,$0000,$aa9b) line 0
                            _x:$0000aa9c
                            _y:$0000aa9b
                            _hl:$0000aa9d
                                    <_hr:$00000002
_g($0000,$aa9d,$0001,$5538) line 0
                            _p:$0000aa9d
                            _q:$00015538
                            _gl:$0000aa9b
                                    <_gr:$0000aa9c

fcnt/D
_fcnt:                      21839

gcnt/D
_gcnt:                      21839

hcnt/D
_hcnt:                      21838

h.x/D
$3f4ff00c:                  43678
$q
```

Figure 14-5: adb output for C program of Figure 14-4 (Sheet 2 of 2)

```
#include <stdio.h>
#define MAXLINE 80
#define YES          1
#define NO           0
#define TABSP        8

char    input[] = "data";
FILE    *ibuf;
short   tabs[MAXLINE];

main()
{
        int col; short *ptab;
        char c;

        ptab = tabs;
        settab(ptab);    /*Set initial tab stops */
        col = 1;
        if ((ibuf = fopen(input,"r")) == NULL) {
                printf("%s : not found\n",input);
                exit(1);
        }
        while((c = getc(ibuf)) != EOF) {
                switch(c) {
                        case '\t':      /* TAB */
                                while(tabpos(col) != YES) {
                                putchar(' ');
                                        col++ ;
                                }
                                break;
                        case '\n':      /*NEWLINE */
                                putchar('\n');
                                col = 1;
                                break;
```

Figure 14-6: C program to decode tabs (Sheet 1 of 2)

```
                        default:
                                putchar(c);
                                col++ ;
                }
        }
}

/* Tabpos return YES if col is a tab stop */
tabpos(col)
int col;
{
        if(col > MAXLINE)
                return(YES);
        else
                return(tabs[col]);
}

/* Settab - Set initial tab stops */
settab(tabp)
int *tabp;
{
        int i;

        for(i = 0; i<= MAXLINE; i++)
                (i%TABSP) ? (tabs[i] = NO) : (tabs[i] = YES);
}
```

Figure 14-6: C program to decode tabs (Sheet 2 of 2)

```
$ adb a.out -
settab+4:b
fopen+4:b
getc+4:b
symbol not found
tabpos+4:b
$b
breakpoints
breakpoints
count                   bkptcommand
1                       _tabpos+$0004
1                       _fopen+$0004
1                       _settab+$0004

settab,5?ia
_settab:        link.w A6,-32         $4e56 $ffe0
_settab+$0004:  clr.l (-32,A6)        $42ae $ffe0
_settab+$0008:  cmpi.l #80,(-32,A6)   $0cae $0000 $0050 $ffe0
_settab+$0010:  bgt _settab+$0048     $6e36
_settab+$0012:  move.l (-32,A6),D1    $222e $ffe0
_settab+$0016:

settab,5?i
_settab:        link.w A6,-32                      $4e56 $ffe0
                        clr.l (-32,A6)  $42ae$ffe0
                        cmpi.l #80,(-32,A6)  $0cae$0000 $0050 $ffe0
                        bgt _settab+$0048  $6e36
                        move.l (-32,A6),D1  $222e$ffe0

:r
a.out: running
breakpoint              _settab+$0004:clr.l (-32,A6)  $42ae$ffe0
```

Figure 14-7: adb output for C program of Figure 14-6 (Sheet 1 of 2)

```
settab+4:d
:c
a.out: running
breakpoint                     _fopen+$0004:jsr __findiop  $4eb9$0000 $11ac

$C
_fopen($0010,$0024,$0010,$002c)line 0
_main($0000,$0001,$3f7f,$ecb2,$3f7f,$ecba) line0
                               _col:$00000001
                               _ptab:$001006a0
                               _c:$00001006

tabs,3/8dn
_tabs:  1       0       0       0       0       0       0       0
        1       0       0       0       0       0       0       0
        1       0       0       0       0       0       0       0
```

Figure 14-8: adb output for C program of Figure 14-6 (Sheet 2 of 2)

```
adb a.out -

tabpos?l #4e5e
_tabpos+#0022
:b
:c
a.out: running
breakpoint       _tabpos+#0022:  unlk    a6

<d0=X
                            #00000000
```

Figure 14-9: adb output for C program of Figure 14-6

```
$ adb ptr core
Program                    killed by bus error
$m
? map       'ptr'
b1 = $000010a8                e1= $00002e0c  f1 = $000000a8
b2 = $00100e0c                e2= $0010149c  f2 = $00001e0c
/ map                        'core'
b1 = $00100000                e1= $00103000  f1 = $00000122
b2 = $3f7fd000                e2= $3f7ff000  f2 = $00003122

$v
variables
b = $00100000
d = $00003000
e = $000010b0
m = $00000108
s = $00002000
t = $00002000
```

Figure 14-10: **adb** output for maps

```
char    str1[] = "This is a character string";
int     one    = 1;
int     number = 456;
long    lnum   = 1234;
float   fpt    = 1.25;
char    str2[] = "This is the second character string";
main()
{
                               one = 2;
}
```

Figure 14-11: Simple C program for illustrating formatting and patching

```
adb ex10 -
:s
stopped                     at__entry:jsr i_start  $4eb9$0000 $00ac
<b,6/8xna
_str1:                      $5468$6973$2069$7320$6120$6368$6172$6163
_str1+$0010:                $7465$7220$7374$7269$6e67$0000$0000$0001

_number:
_number:                    $0000$01c8$0000$04d2$3fa0$0000$5468$6973

_str2+$0004:                $2069$7320$7468$6520$7365$636f$6e64$2063

_str2+$0014:                $6861$7261$6374$6572$2073$7472$696e$6700

__ctype_:
__ctype_:                   $0020$2020$2020$2020$2020$2828$2828$2820

__ctype_+$0010:

<b,10/4x4^8Cn
_str1:                      $5468$6973$2069$7320This is
                            $6120$6368$6172$6163a charac
                            $7465$7220$7374$7269ter stri
                            $6e67$0000$0000$0001ng@`@`@`@`@`@a

_number:                    $0000$01c8$0000$04d2@`@`@aH@`@`@dR

_fpt:                       $3fa0$0000$5468$6973? @`@`This
                            $2069$7320$7468$6520 is the
                            $7365$636f$6e64$2063second c
                            $6861$7261$6374$6572haracter
                            $2073$7472$696e$6700 string@`
```

Figure 14-12: adb output illustrating fancy formats (Sheet 1 of 2)

```
<b,10/4x4^8t8cna
_str1:                  $5468$6973$2069$7320This is
_str1+$0008:            $6120$6368$6172$6163a charac
_str1+$0010:            $7465$7220$7374$7269ter stri
_str1+$0018:            $6e67$0000$0000$0001ng
_number:
_number:                $0000$01c8$0000$04d2HR
_fpt:
_fpt:                   $3fa0$0000$5468$6973? This
_str2+$0004:            $2069$7320$7468$6520 is the
_str2+$000c:            $7365$636f$6e64$2063second c
_str2+$0014:            $6861$7261$6374$6572haracter
_str2+$001c:            $2073$7472$696e$6700 string
__ctype_:
<b,10/2b8t^2cn
_str1:                  $0054$0068Th
                        $0069$0073is
                        $0020$0069 i
                        $0073$0020s
                        $0061$0020a
                        $0063$0068ch
                        $0061$0072ar
                        $0061$0063ac
                        $0074$0065te
                        $0072$0020r
$Q
```

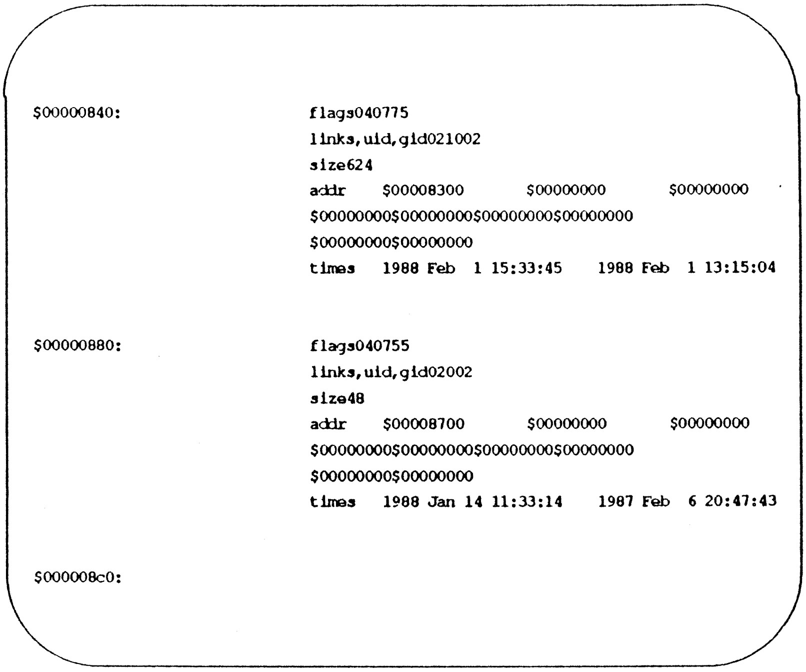
Figure 14-12: adb output illustrating fancy formats (Sheet 2 of 2)

```
adb . -    # dump '.', the current directory
=nt"Inode"t"Name"
0,6?ut14cn


                             Inode Name
$00000000:                  7959 .
                            6423 ..
                            7918 tut
                            6846 tut1
                            6852 ptr.script
                            7772 ptr.c

adb /dev/sw0.0 -
2048,37"flags"8ton"links,uid,gid"8t3on"size"8tDn"addr"8t10Xn"times"8t3Y2na
$00000800:                  flags0100000
                            links,uid,gid000
                            size0
                            addr    $00000000       $00000000       $00000000
                            $00000000$00000000$00000000$00000000
                            $00000000$00000000
                            times   1986 Oct 31 12:11:21    1986 Oct 31 12:11:21
```

Figure 14-13: Directory and inode dumps (Sheet 1 of 2)



```
$00000840:                  flags040775
                            links,uid,gid021002
                            size624
                            addr     $00008300         $00000000         $00000000
                            $00000000$00000000$00000000$00000000
                            $00000000$00000000
                            times    1988 Feb  1 15:33:45    1988 Feb  1 13:15:04


$00000880:                  flags040755
                            links,uid,gid02002
                            size48
                            addr     $00008700         $00000000         $00000000
                            $00000000$00000000$00000000$00000000
                            $00000000$00000000
                            times    1988 Jan 14 11:33:14    1987 Feb  6 20:47:43


$000008c0:
```

Figure 14-14: Directory and inode dumps (Sheet 2 of 2)

adb Summary

Command Summary

a) formatted printing

?format	print from *a.out* file according to *format*
/format	print from *core* file according to *format*
=format	print the value of *dot*
?w expr	write expression into *a.out* file
/w expr	write expression into *core* file
?l expr	locate expression in *a.out* file

b) breakpoint and program control

:b	set breakpoint at *dot*
:c	continue running program
:d	delete breakpoint
:k	kill the program being debugged
:r	run *a.out* file under ADB control
:s	single step

c) miscellaneous printing

$b	print current breakpoints
$c	C stack trace
$e	external variables
$f	floating registers (not yet)
$m	print ADB segment maps
$q	exit from ADB
$r	general registers
$s	set offset for symbol match
$v	print ADB variables
$w	set output line width

d) calling the shell

!	call *shell* to read rest of line

e) assignment to variables

> name	assign dot to variable or register *name*

Format Summary

a	the value of dot
b	one byte in octal
c	one byte as a character
d	one word in decimal
f	two words in floating point
i	68020 instruction
o	one word in octal
n	print a newline
r	print a blank space
s	a null terminated character string
nt	move to next *n* space tab
u	one word as unsigned integer
x	hexadecimal
Y	date
^	backup dot
"..."	print string

Expression Summary

a) expression components

decimal integer	e.g. 256
octal integer	e.g. 0277
hexadecimal	e.g. #ff
symbols	e.g. _main, main.argc
variables	e.g. <b
registers	e.g. <pc, <d0
(expression)	expression grouping

b) dyadic operators

+	add
–	subtract
*	multiply
%	integer division
&	bitwise and
|	bitwise or
#	round up to the next multiple

c) monadic operators

	not
*	contents of location
–	integer negate

Table of Contents

Chapter 15: **lint**

Introduction

The **lint** program examines C language source programs detecting a number of bugs and obscurities. It enforces the type rules of C language more strictly than the C compiler. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful or error prone constructions, which nevertheless are legal. **lint** accepts multiple input files and library specifications and checks them for consistency.

Usage

The **lint** command has the form:

lint [*options*] *files* ... *library-descriptors* ...

where *options* are optional flags to control **lint** checking and messages; *files* are the files to be checked which end with **.c** or **.ln**; and *library-descriptors* are the names of libraries to be used in checking the program.

The options that are currently supported by the **lint** command are:

–a	Suppress messages about assignments of long values to variables that are not long.
–b	Suppress messages about break statements that cannot be reached.
–c	Only check for intra-file bugs; leave external information in files suffixed with **.ln**.
–h	Do not apply heuristics (which attempt to detect bugs, improve style, and reduce waste).
–n	Do not check for compatibility with either the standard or the portable **lint** library.
–o *name*	Create a lint library from input files named **llib–l***name***.ln**.
–p	Attempt to check portability.
–u	Suppress messages about function and external variables used and not defined or defined and not used.
–v	Suppress messages about unused arguments in functions.
–x	Do not report variables referred to by external declarations but never used.

When more than one option is used, they should be combined into a single argument, such as **–ab** or **–xha**.

The names of files that contain C language programs should end with the suffix .c, which is mandatory for **lint** and the C compiler.

lint accepts certain arguments, such as:

–lm

These arguments specify libraries that contain functions used in the C language program. The source code is tested for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library arguments. These files all begin with the comment:

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED comments can be used to specify features of the library functions. The next section, "**lint** Message Types," describes how it is done.

lint library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined in a library file but are not used in a source file do not result in messages. **lint** does not simulate a full library search algorithm and will print messages if the source files contain a redefinition of a library routine.

By default, **lint** checks the programs it is given against a standard library file that contains descriptions of the programs that are normally loaded when a C language program is run. When the **–p** option is used, another file is checked containing descriptions of the standard library routines which are expected to be portable across various machines. The **–n** option can be used to suppress all library checking.

lint Message Types

The following paragraphs describe the major categories of messages printed by **lint**.

Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused. It is not uncommon for external variables or even entire functions to become unnecessary and yet not be removed from the source. These types of errors rarely cause working programs to fail, but are a source of inefficiency and make programs harder to understand and change. Also, information about such unused variables and functions can occasionally serve to discover bugs.

lint prints messages about variables and functions which are defined but not otherwise mentioned, unless the message is suppressed by means of the **–u** or **–x** option.

Certain styles of programming may permit a function to be written with an interface where some of the function's arguments are optional. Such a function can be designed to accomplish a variety of tasks depending on which arguments are used. Normally **lint** prints messages about unused arguments; however, the –v option is available to suppress the printing of these messages. When –v is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments. This can be considered an active (and preventable) waste of the register resources of the machine.

Messages about unused arguments can be suppressed for one function by adding the comment:

```
/* ARGSUSED */
```

to the source code before the function. This has the effect of the –v option for only one function. Also, the comment:

```
/* VARARGS */
```

can be used to suppress messages about variable number of arguments in calls to a function. The comment should be added before the function definition. In some cases, it is desirable to check the first several arguments and leave the later arguments unchecked. This can be done with a digit giving the number of arguments which should be checked. For example:

```
        /* VARARGS2 */
```

will cause only the first two arguments to be checked.

When **lint** is applied to some but not all files out of a collection that are to be loaded together, it issues complaints about unused or undefined variables. This information is, of course, more distracting than helpful. Functions and variables that are defined may not be used; conversely, functions and variables defined elsewhere may be used. The **–u** option suppresses the spurious messages.

Set/Used Information

lint attempts to detect cases where a variable is used before it is set. **lint** detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use" since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement since the true flow of control need not be discovered. It does mean that **lint** can print error messages about program fragments that are legal, but these programs would probably be considered bad on stylistic grounds. Because static and external variables are initialized to zero, no meaningful information can be discovered about their uses. The **lint** program does deal with initialized automatic variables.

The set/used information also permits recognition of those local variables that are set and never used. These form a frequent source of inefficiencies and may also be symptomatic of bugs.

Flow of Control

lint attempts to detect unreachable portions of a program. It will print messages about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. It attempts to detect loops that cannot be left at the bottom and to recognize the special cases **while(1)** and **for(;;)** as infinite loops. **lint** also prints messages about loops that cannot be entered at the top. Valid programs may have such loops, but they are considered to be bad style. If you do not want messages about unreached portions of the program, use the **–b** option.

lint has no way of detecting functions that are called and never return. Thus, a call to **exit** may cause unreachable code which **lint** does not detect. The most serious effects of this are in the determination of returned function values (see "Function Values"). If a particular place in the program is thought to be unreachable in a way that is not apparent to **lint**, the comment

```
/* NOTREACHED */
```

can be added to the source code at the appropriate place. This comment will inform **lint** that a portion of the program cannot be reached, and **lint** will not print a message about the unreachable portion.

Programs generated by **yacc** and especially **lex** may have hundreds of unreachable **break** statements, but messages about them are of little importance. There is typically nothing the user can do about them, and the resulting messages would clutter up the **lint** output. The recommendation is to invoke **lint** with the **–b** option when dealing with such input.

Function Values

Sometimes functions return values that are never used. Sometimes programs incorrectly use function values that have never been returned. **lint** addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
return(  expr );
```

and

```
return ;
```

statements is cause for alarm; **lint** will give the message

```
function name has return(e) and return
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f ( a ) {
            if ( a ) return ( 3 );
            g ();
            }
```

Notice that, if **a** tests false, **f** will call **g** and then return with no defined return value; this will trigger a message from **lint**. If **g**, like **exit**, never returns, the message will still be produced when in fact nothing is wrong. A comment

```
        /*NOTREACHED*/
```

in the source code will cause the message to be suppressed. In practice, some potentially serious bugs have been discovered by this feature.

On a global scale, **lint** detects cases where a function returns a value that is sometimes or never used. When the value is never used, it may constitute an inefficiency in the function definition that can be overcome by specifying the function as being of type (void). For example:

```
        (void) fprintf(stderr,"File busy. Try again later!\n");
```

When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The opposite problem, using a function value when the function does not return one, is also detected. This is a serious problem.

Type Checking

lint enforces the type checking rules of C language more strictly than the compilers do. The additional checking is in four major areas:

- across certain binary operators and implied assignments
- at the structure selection operators
- between the definition and uses of functions
- in the use of enumerations

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional (?:), and relational operators have this property. The argument of a **return** statement and expressions used in initialization suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly except that arrays of *x*s can, of course, be intermixed with pointers to *x*s.

The type checking rules also require that, in structure references, the left operand of the -> be a pointer to structure, the left operand of the . be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types or other enumerations and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

If it is desired to turn off strict type checking for an expression, the comment

```
/* NOSTRICT */
```

should be added to the source code immediately before the expression. This comment will prevent strict type checking for only the next line in the program.

Type Casts

The type cast feature in C language was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where **p** is a character pointer. **lint** will print a message as a result of detecting this. Consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this and has clearly signaled his intentions. Nevertheless, **lint** will continue to print messages about this.

Nonportable Character Use

On some systems, characters are signed quantities with a range from −128 to 127. On other C language implementations, characters take on only positive values. Thus, **lint** will print messages about certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
char c;
          . . .
if( (c = getchar( )) < 0 ) . . .
```

will work on one machine but will fail on machines where characters always take on positive values. The real solution is to declare *c* as an integer since **getchar** is actually returning integer values. In any case, **lint** will print the message

```
nonportable character comparison
```

A similar issue arises with bit fields. When assignments of constant values are made to bit fields, the field may be too small to hold the value. This is especially true because on some machines bit fields are considered as signed quantities. While it may seem logical to consider that a two-bit field declared of type **int** cannot hold the value 3, the problem disappears if the bit field is declared to have type **unsigned**

Assignments of longs to ints

Bugs may arise from the assignment of **long** to an **int**, which will truncate the contents. This may happen in programs which have been incompletely converted to use **typedefs**. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **ints**, which are truncated. The **–a** option can be used to suppress messages about the assignment of **longs** to **ints**.

Strange Constructions

Several perfectly legal, but somewhat strange, constructions are detected by **lint**. The messages hopefully encourage better code quality, clearer style, and may even point out bugs. The **–h** option is used to suppress these checks. For example, in the statement

```
*p++ ;
```

the * does nothing. This provokes the message

```
null effect
```

from **lint**. The following program fragment:

```
unsigned x ;
if( x < 0 ) . . .
```

results in a test that will never succeed. Similarly, the test

```
if( x > 0 ) . . .
```

is equivalent to

```
        if( x != 0 )
```

which may not be the intended action. **lint** will print the message

```
        degenerate unsigned comparison
```

in these cases. If a program contains something similar to

```
        if( 1 != 0 ) . . .
```

lint will print the message

```
        constant in conditional context
```

since the comparison of 1 with 0 gives a constant result.

Another construction detected by **lint** involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements

```
        if( x&077 == 0 ) . . .
```

and

```
        x<<2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and **lint** encourages this by an appropriate message.

Old Syntax

Several forms of older syntax are now illegal. These fall into two classes: assignment operators and initialization.

The older forms of assignment operators (e.g., =+, =–, ...) could cause ambiguous expressions, such as:

```
        a =-1 ;
```

which could be taken as either

```
        a =- 1 ;
```

or

```
        a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer and preferred operators (e.g., +=, –=, ...) have no such ambiguities. To encourage the abandonment of the older forms, **lint** prints messages about these old-fashioned operators.

A similar issue arises with initialization. The older language allowed

```
int x 1;
```

to initialize *x* to 1. This also caused syntactic difficulties. For example, the initialization

```
int x ( -1 ) ;
```

looks somewhat like the beginning of a function definition:

```
int x ( y ) { . . .
```

and the compiler must read past *x* in order to determine the correct meaning. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

```
int x = -1 ;
```

This is free of any possible syntactic ambiguity.

Pointer Alignment

Certain pointer assignments may be reasonable on some machines and illegal on others due entirely to alignment restrictions. lint tries to detect cases where pointers are assigned to other pointers and such alignment problems might arise. The message

```
possible pointer alignment problem
```

results from this situation.

Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines in which the stack runs backwards, function arguments will probably be best evaluated from right to left. On machines with a stack running forward, left to right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators that have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C language on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler. In fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect and also used elsewhere in the same expression, the result is explicitly undefined.

lint checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++];
```

will cause **lint** to print the message

```
warning: i evaluation order undefined
```

in order to call attention to this condition.

Table of Contents

Chapter 16: C Language

Introduction

This chapter contains a summary of the grammar and syntax rules of the C Programming Language. The implementation described is that found on the CADMUS line of 32 bit computers. A consistent attempt is made to point out where other implementations may differ.

Lexical Conventions

There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. Blanks, tabs, new-lines, and comments (collectively, "white space") as described below are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token.

Comments

The characters /* introduce a comment that terminates with the characters */. Comments do not nest.

Identifiers (Names)

An identifier is a sequence of letters and digits. The first character must be a letter. The underscore (_) counts as a letter. Uppercase and lowercase letters are different. There is no limit on the length of a name. Other implementations may collapse case distinctions for external names, and may reduce the number of significant characters for both external and non-external names.

Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

asm	**default**	**float**	**register**	**switch**
auto	**do**	**for**	**return**	**typedef**
break	**double**	**goto**	**short**	**union**
case	**else**	**if**	**sizeof**	**unsigned**
char	**enum**	**int**	**static**	**void**
continue	**external**	**long**	**struct**	**while**

Some implementations also reserve the word **fortran**.

Constants

There are several kinds of constants. Each has a type; an introduction to types is given in "Storage Class and Type."

Integer Constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with **0** (digit zero). An octal constant consists of the digits **0** through **7** only. A sequence of digits preceded by **0x** or **0X** (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include **a** or **A** through **f** or **F** with values 10 through 15. Otherwise, the integer constant is taken to be decimal. A decimal constant whose value exceeds the largest signed machine integer is taken to be **long**; an octal or hex constant that exceeds the largest unsigned machine integer is likewise taken to be **long**. Otherwise, integer constants are **int**.

Explicit Long Constants

A decimal, octal, or hexadecimal integer constant immediately followed by **l** (letter ell) or **L** is a long constant. As discussed below, on CADMUS computers integer and long values may be considered identical.

Character Constants

A character constant is a character enclosed in single quotes, as in **´x'**. The value of a character constant is the numerical value of the character in the machine's character set. Certain nongraphic characters, the single quote (') and the backslash (\), may be represented according to the table of escape sequences shown in Figure 17-1:

new-line NL (LF)	\n	
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
single quote	'	\'
bit pattern	*ddd*	\ddd

Figure 16-1: Escape Sequences for Nongraphic Characters

The escape \ddd consists of the backslash followed by 1, 2, or 3 octal digits that are taken to specify the value of the desired character. A special case of this construction is **\0** (not followed by a digit), which indicates the ASCII character NUL. If the character following a backslash is not one of those specified, the behavior is undefined. An explicit new-line character is illegal in a character constant. The type of a character constant is **int**.

Floating Constants

A floating constant consists of an integer part, a decimal point, a fraction part, an **e** or **E**, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing. Either the decimal point or the **e** and the exponent (not both) may be missing. Every floating constant has type **double**.

Enumeration Constants

Names declared as enumerators (see "Structure, Union, and Enumeration Declarations" under "Declarations") have type **int**.

String Literals

A string literal is a sequence of characters surrounded by double quotes, as in "...". A string literal has type "array of **char**" and storage class **static** (see "Storage Class and Type") and is initialized with the given characters. The compiler places a null byte (\0) at the end of each string literal so that programs that scan the string literal can find its end. In a string literal, the double quote character (") must be preceded by a \; in addition, the same escapes as described for character constants may be used.

A \ and the immediately following new-line are ignored. All string literals, even when written identically, are distinct.

Syntax Notation

Syntactic categories are indicated by *italic* type and literal words and characters by **bold** type. Alternative categories are listed on separate lines. An optional entry is indicated by the subscript "opt," so that

$\{\ \textit{expression}_{opt}\ \}$

indicates an optional expression enclosed in braces. The syntax is summarized in "Syntax Summary" at the end of the chapter.

Storage Class and Type

The C language bases the interpretation of an identifier upon two attributes of the identifier: its storage class and its type. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

Storage Class

There are four declarable storage classes:

- automatic
- static
- external
- register

Automatic variables are local to each invocation of a block (see "Compound Statement or Block" in "Statements") and are discarded upon exit from the block. Static variables are local to a block but retain their values upon reentry to a block even after control has left the block. External variables exist and retain their values throughout the execution of the entire program and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables, they are local to each block and disappear on exit from the block.

Type

The C language supports several fundamental types of objects. Objects declared as characters (char) are large enough to store any member of the implementation's character set. If a genuine character from that character set is stored in a char variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine dependent. In particular, char may be signed or unsigned by default. In this implementation the default is unsigned.

Up to three sizes of integer, declared **short int**, **int**, and **long int**, are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers or long integers, or both, equivalent to plain integers. Plain integers have the natural size suggested by the host machine architecture. The other sizes are provided to meet special needs. The sizes for the CADMUS 32 bit computers are shown in Figure 17-2.

CADMUS 32 bit (ASCII)	
char	8 bits
int	32
short	16
long	32
float	32
double	64
float range	$\pm 10^{\pm 38}$
double range	$\pm 10^{\pm 308}$

Figure 16-2: CADMUS 32 bit Computer Hardware Characteristics

The properties of **enum** types (see "Structure, Union, and Enumeration Declarations" under "Declarations") are identical to those of some integer types. The implementation may use the range of values to determine how to allot storage.

Unsigned integers, declared **unsigned**, obey the laws of arithmetic modulo 2^n where *n* is the number of bits in the representation.

Single-precision floating point (**float**) and double precision floating point (**double**) may be synonymous in some implementations.

Because objects of the foregoing types can usefully be interpreted as numbers, they will be referred to as arithmetic types. **Char**, **int** of all sizes whether **unsigned** or not, and **enum** will collectively be called integral types. The **float** and **double** types will collectively be called floating types.

The void type specifies an empty set of values. It is used as the type returned by functions that generate no value.

Besides the fundamental arithmetic types, there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

- arrays of objects of most types
- functions that return objects of a given type
- pointers to objects of a given type
- structures containing a sequence of objects of various types
- unions capable of containing any one of several objects of various types

In general these methods of constructing objects can be applied recursively.

Objects and lvalues

An object is a manipulatable region of storage. An lvalue is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators that yield lvalues: for example, if E is an expression of pointer type, then *E is an lvalue expression referring to the object to which E points. The name "lvalue" comes from the assignment expression E1 = E2 in which the left operand E1 must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

Operator Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This part explains the result to be expected from such conversions. The conversions demanded by most ordinary operators are summarized under "Arithmetic Conversions." The summary will be supplemented as required by the discussion of each operator.

Characters and Integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a shorter integer to a longer preserves sign. On the CADMUS computers sign extension of **char** variables occurs (but see the cc(1) option -u). It is guaranteed that a member of the standard character set is non-negative.

On machines that treat characters as signed, the characters of the ASCII set are all non-negative. However, a character constant specified with an octal escape suffers sign extension and may appear negative; for example, '\377' has the value –1.

When a longer integer is converted to a shorter integer or to a **char**, it is truncated on the left. Excess bits are simply discarded.

Float and Double

All floating arithmetic in C is carried out in double precision. Whenever a **float** appears in an expression it is lengthened to **double** by zero padding its fraction. When a **double** must be converted to **float**, for example by an assignment, the **double** is rounded before truncation to **float** length. This result is undefined if it cannot be represented as a float.

Floating and Integral

Conversions of floating values to integral type are rather machine dependent. In particular, the direction of truncation of negative numbers varies. The result is undefined if it will not fit in the space provided.

Conversions of integral values to floating type behave well. Some loss of accuracy occurs if the destination lacks sufficient bits.

Pointers and Integers

An expression of integral type may be added to or subtracted from a pointer; in such a case, the first is converted as specified in the discussion of the addition operator. Two pointers to objects of the same type may be subtracted; in this case, the result is converted to an integer as specified in the discussion of the subtraction operator.

Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value is the least unsigned integer congruent to the signed integer (modulo $2^{wordsize}$). In a 2's complement representation, this conversion is conceptual; and there is no actual change in the bit pattern.

When an unsigned **short** integer is converted to **long**, the value of the result is the same numerically as that of the unsigned integer. Thus, the conversion amounts to padding with zeros on the left.

Arithmetic Conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the "usual arithmetic conversions."

1. First, any operands of type **char** or **short** are converted to **int**, and any operands of type **unsigned char** or **unsigned short** are converted to **unsigned int**.
2. Then, if either operand is **double**, the other is converted to **double** and that is the type of the result.

3. Otherwise, if either operand is **unsigned long**, the other is converted to **unsigned long** and that is the type of the result.
4. Otherwise, if either operand is **long**, the other is converted to **long** and that is the type of the result.
5. Otherwise, if one operand is **long**, and the other is **unsigned int**, they are both converted to **unsigned long** and that is the type of the result.
6. Otherwise, if either operand is **unsigned**, the other is converted to **unsigned** and that is the type of the result.
7. Otherwise, both operands must be **int**, and that is the type of the result.

Void

The (nonexistent) value of a **void** object may not be used in any way, and neither explicit nor implicit conversion may be applied. Because a void expression denotes a nonexistent value, such an expression may be used only as an expression statement (see "Expression Statement" under "Statements") or as the left operand of a comma expression (see "Comma Operator" under "Expressions").

An expression may be converted to type **void** by use of a cast. For example, this makes explicit the discarding of the value of a function call used as an expression statement.

Expressions and Operators

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions referred to as the operands of + (see "Additive Operators") are those expressions defined under "Primary Expressions", "Unary Operators", and "Multiplicative Operators". Within each subpart, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators are summarized in the grammar of "Syntax Summary".

Otherwise, the order of evaluation of expressions is undefined. In particular, the compiler considers itself free to compute subexpressions in the order it believes most efficient even if the subexpressions involve side effects. Expressions involving a commutative and associative operator (*, +, &, |, ^) may be rearranged arbitrarily even in the presence of parentheses; to force a particular order of evaluation, an explicit temporary must be used.

The handling of overflow and divide check in expression evaluation is undefined. Most existing implementations of C ignore integer overflows; treatment of division by 0 and all floating-point exceptions varies between machines and is usually adjustable by a library function.

Primary Expressions

Primary expressions involving ., ->, subscripting, and function calls group left to right.

primary-expression:
- *identifier*
- *constant*
- *string literal*
- *(expression)*
- *primary-expression [expression]*
- *primary-expression (expression-list*$_{opt}$ *)*
- *primary-expression . identifier*
- *primary-expression -> identifier*

expression-list:
  expression
  expression-list , expression

An identifier is a primary expression provided it has been suitably declared as discussed below. Its type is specified by its declaration. If the type of the identifier is "array of . . .", then the value of the identifier expression is a pointer to the first object in the array; and the type of the expression is "pointer to . . .". Moreover, an array identifier is not an lvalue expression. Likewise, an identifier that is declared "function returning . . .", when used except in the function-name position of a call, is converted to "pointer to function returning . . .".

A constant is a primary expression. Its type may be **int**, **long**, or **double** depending on its form. Character constants have type **int** and floating constants have type **double**.

A string literal is a primary expression. Its type is originally "array of **char**", but following the same rule given above for identifiers, this is modified to "pointer to **char**" and the result is a pointer to the first character in the string literal. (There is an exception in certain initializers; see "Initialization" under "Declarations.")

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type "pointer to . . .", the subscript expression is **int**, and the type of the result is " . . . ". The expression **E1[E2]** is identical (by definition) to ***((E1)+(E2))**. All the clues needed to understand this notation are contained in this subpart together with the discussions in "Unary Operators" and "Additive Operators" on identifiers, * and +, respectively. The implications are summarized under "Arrays, Pointers, and Subscripting" under "Types Revisited."

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions that constitute the actual arguments to the function. The primary expression must be of type "function returning . . .", and the result of the function call is of type " . . . ". As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer.

Any actual arguments of type **float** are converted to **double** before the call. Any of type **char** or **short** are converted to **int**. Array names are converted to pointers. No other conversions are performed automatically; in particular, the compiler does not compare the types of actual arguments with those of formal arguments. If conversion is needed, use a cast; see "Unary Operators" and "Type Names" under "Declarations."

In preparing for the call to a function, a copy is made of each actual parameter. Thus, all argument passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. It is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. An array name is a pointer expression. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ. Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be a structure or a union, and the identifier must name a member of the structure or union. The value is the named member of the structure or union, and it is an lvalue if the first expression is an lvalue.

A primary expression followed by an arrow (built from – and >) followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points. Thus the expression **E1–>MOS** is the same as **(*E1).MOS**. Structures and unions are discussed in "Structure, Union, and Enumeration Declarations" under "Declarations."

Unary Operators

Expressions with unary operators group right to left.

unary-expression:
- * *expression*
- & *lvalue*
- – *expression*
- ! *expression*
- ~ *expression*
- ++ *lvalue*
- —*lvalue*
- *lvalue* ++
- *lvalue* —
- *(type-name) expression*
- sizeof *expression*
- sizeof *(type-name)*

The unary * operator means "indirection"; the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is "pointer to . . .," the type of the result is " . . . ".

The result of the unary & operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is " . . . ", the type of the result is "pointer to . . .".

The result of the unary – operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from 2^n where n is the number of bits in the corresponding signed type.

There is no unary + operator.

The result of the logical negation operator ! is one if the value of its operand is zero, zero if the value of its operand is nonzero. The type of the result is **int**. It is applicable to any arithmetic type or to pointers.

The ~ operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix ++ is incremented. The value is the new value of the operand but is not an lvalue. The expression ++x is equivalent to x += 1. See the discussions "Additive Operators" and "Assignment Operators" for information on conversions.

The lvalue operand of prefix — is decremented analogously to the prefix ++ operator.

When postfix ++ is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix ++ operator. The type of the result is the same as the type of the lvalue expression.

When postfix -- is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the prefix -- operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. This construction is called a cast. Type names are described in "Type Names" under "Declarations."

The **sizeof** operator yields the size in bytes of its operand. (A byte is undefined by the language except in terms of the value of **sizeof**. However, in all existing implementations, a byte is the space required to hold a **char**.) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an **unsigned** constant and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The **sizeof** operator may also be applied to a parenthesized type name. In that case it yields the size in bytes of an object of the indicated type.

The construction **sizeof(*type*)** is taken to be a unit, so the expression **sizeof(*type*)-2** is the same as **(sizeof(*type*))-2**.

Multiplicative Operators

The multiplicative operators *, /, and % group left to right. The usual arithmetic conversions are performed.

multiplicative expression:
  *expression * expression*
  expression / expression
  expression % expression

The binary * operator indicates multiplication. The * operator is associative, and expressions with several multiplications at the same level may be rearranged by the compiler. The binary / operator indicates division.

The binary % operator yields the remainder from the division of the first expression by the second. The operands must be integral.

When positive integers are divided, truncation is toward 0; but the form of truncation is machine-dependent if either operand is negative. On all machines covered by this manual, the remainder has the same sign as the dividend. It is always true that **(a/b)*b + a%b** is equal to **a** (if **b** is not 0).

Additive Operators

The additive operators + and – group left to right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

additive-expression:
  expression + expression
  expression – expression

The result of the + operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer that points to another object in the same array, appropriately offset from the original object. Thus if **P** is a pointer to an object in an array, the expression **P+1** is a pointer to the next object in the array. No further type combinations are allowed for pointers.

The + operator is associative, and expressions with several additions at the same level may be rearranged by the compiler.

The result of the – operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a pointer, and then the same conversions for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an int representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object length.

Shift Operators

The shift operators << and >> group left to right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to **int**; the type of the result is that of the left operand. The result is undefined if the right operand is negative or greater than or equal to the length of the object in bits.

shift-expression:
expression << expression
expression >> expression

The value of **E1<<E2** is **E1** (interpreted as a bit pattern) left-shifted **E2** bits. Vacated bits are 0 filled. The value of **E1>>E2** is **E1** right-shifted **E2** bit positions. The right shift is guaranteed to be logical (0 fill) if **E1** is **unsigned**; otherwise, it may be arithmetic.

Relational Operators

The relational operators group left to right.

relational-expression:
expression < expression
expression > expression
expression <= expression
expression >= expression

The operators < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is **int**. The usual arithmetic conversions are performed. Two pointers may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

Equality Operators

equality-expression:
expression == expression
expression != expression

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus **a<b == c<d** is 1

whenever **a<b** and **c<d** have the same truth value.)

A pointer may be compared to an integer only if the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object and will appear to be equal to 0. In conventional usage, such a pointer is considered to be null.

Bitwise AND Operator

and-expression:
　　expression & expression

The & operator is associative, and expressions involving & may be rearranged. The usual arithmetic conversions are performed. The result is the bitwise AND function of the operands. The operator applies only to integral operands.

Bitwise Exclusive OR Operator

exclusive-or-expression:
　　expression ^ expression

The ^ operator is associative, and expressions involving ^ may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

Bitwise Inclusive OR Operator

inclusive-or-expression:
　　expression | expression

The | operator is associative, and expressions involving | may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

Logical AND Operator

logical-and-expression:
expression && expression

The && operator groups left to right. It returns 1 if both its operands evaluate to nonzero, 0 otherwise. Unlike &, && guarantees left to right evaluation; moreover, the second operand is not evaluated if the first operand evaluates to 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

Logical OR Operator

logical-or-expression:
expression || expression

The || operator groups left to right. It returns 1 if either of its operands evaluates to nonzero, 0 otherwise. Unlike |, || guarantees left to right evaluation; moreover, the second operand is not evaluated if the value of the first operand evaluates to nonzero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

Conditional Operator

conditional-expression:
expression ? expression : expression

Conditional expressions group right to left. The first expression is evaluated; and if it is nonzero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type. If both are structures or unions of the same type, the result has the type of the structure or union. If both pointers are of the same type, the result has the common type. Otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

Assignment Operators

There are a number of assignment operators, all of which group right to left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are separate tokens.

assignment-expression:
lvalue = expression
lvalue += expression
lvalue −= expression
*lvalue *= expression*
lvalue /= expression
lvalue %= expression
lvalue >>= expression
lvalue <<= expression
lvalue &= expression
lvalue ^= expression
lvalue |= expression

In the simple assignment with =, the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment. Second, both operands may be structures or unions of the same type. Finally, if the left operand is a pointer, the right operand must in general be a pointer of the same type. However, the constant 0 may be assigned to a pointer; it is guaranteed that this value will produce a null pointer distinguishable from a pointer to any object.

The behavior of an expression of the form **E1** *op* = **E2** may be inferred by taking it as equivalent to **E1** = **E1** *op* (**E2**); however, **E1** is evaluated only once. In += and −=, the left operand may be a pointer, in which case the (integral) right operand is converted as explained in "Additive Operators." All right operands and all nonpointer left operands must have arithmetic type.

Comma Operator

comma-expression:
  expression , expression

A pair of expressions separated by a comma is evaluated left to right, and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left to right. In contexts where comma is given a special meaning, e.g., in lists of actual arguments to functions (see "Primary Expressions") and lists of initializers (see "Initialization" under "Declarations"), the comma operator as described in this subpart can only appear in parentheses. For example,

```
f(a, (t=3, t+2), c)
```

has three arguments, the second of which has the value 5.

Declarations

Declarations are used to specify the interpretation that C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

> *declaration:*
>
> *decl-specifiers declarator-list*$_{opt}$ *;*

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

> *decl-specifiers:*
>
> *type-specifier decl-specifiers*$_{opt}$
>
> *sc-specifier decl-specifiers*$_{opt}$

The list must be self-consistent in a way described below.

Storage Class Specifiers

The sc-specifiers are:

> *sc-specifier:*
>
> **auto**
>
> **static**
>
> **extern**
>
> **register**
>
> **typedef**

The **typedef** specifier does not reserve storage and is called a "storage class specifier" only for syntactic convenience. See "**typedef**" for more information. The meanings of the various storage classes were discussed in "Names."

The **auto, static,** and **register** declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the **extern** case, there must be an external definition (see "External Definitions") for the given identifiers somewhere outside the function in which they are declared.

A **register** declaration is best thought of as an **auto** declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few such declarations in each function are effective. Moreover, only variables of certain types will be stored in registers. One other restriction applies to variables declared using register storage clas: the address-of operator,

&, cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately.

At most, one sc-specifier may be given in a declaration. If the sc-specifier is missing from a declaration, it is taken to be **auto** inside a function, **extern** outside. Exception: functions are never automatic.

Type Specifiers

The type-specifiers are

> *type-specifier:*
>   *struct-or-union-specifier*
>   *typedef-name*
>   *enum-specifier*
> *basic-type-specifier:*
>   *basic-type*
>   *basic-type basic-type-specifiers*
> *basic-type:*
>   **char**
>   **short**
>   **int**
>   **long**
>   **unsigned**
>   **float**
>   **double**
>   **void**

At most one of the words **long** or **short** may be specified in conjunction with **int**; the meaning is the same as if **int** were not mentioned. The word **long** may be specified in conjunction with **float**; the meaning is the same as **double**. The word **unsigned** may be specified alone, or in conjunction with **int** or any of its short or long varieties, or with **char**.

Otherwise, at most on type-specifier may be given in a declaration. In particular, adjectival use of **long**, **short**, or **unsigned** is not permitted with **typedef** names. If the type-specifier is missing from a declaration, it is taken to be **int**.

Specifiers for structures, unions, and enumerations are discussed in "Structure, Union, and Enumeration Declarations." Declarations with **typedef** names are discussed in "**typedef**."

Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer:

declarator-list:
    init-declarator
    init-declarator , declarator-list

init-declarator:
    declarator initializer$_{opt}$

Initializers are discussed in "Initialization." The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

declarator:
    identifier
    (declarator)
    ** declarator*
    declarator ()
    declarator [constant-expression$_{opt}$ *]*

The grouping is the same as in expressions.

Meaning of Declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class.

Each declarator contains exactly one identifier; it is this identifier that is declared. If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

Now imagine a declaration

T D1

where **T** is a type-specifier (like **int**, etc.) and **D1** is a declarator. Suppose this declaration makes the identifier have type "... **T** ," where the "..." is empty if

D1 is just a plain identifier (so that the type of **x** in "**int x**" is just **int**). Then if **D1** has the form

```
*D
```

the type of the contained identifier is "... pointer to **T** ."

If **D1** has the form

```
D()
```

then the contained identifier has the type "... function returning **T**."

If **D1** has the form

D[*constant-expression*]

or

D[]

then the contained identifier has type "... array of T." In the first case, the constant expression is an expression whose value is determinable at compile time, whose type is **int**, and whose value is positive. (Constant expressions are defined precisely in "Constant Expressions.") When several "array of" specifications are adjacent, a multi-dimensional array is created; the constant expressions that specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful when the array is external and the actual definition, which allocates storage, is given elsewhere. The first constant expression may also be omitted when the declarator is followed by initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multi-dimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays or functions although they may return pointers; there are no arrays of functions although there may be arrays of pointers to functions. Likewise, a structure or union may not contain a function; but it may contain a pointer to a function.

As an example, the declaration

```
int i, *ip, f(), *fip(), (*pfi)();
```

declares an integer **i**, a pointer **ip** to an integer, a function **f** returning an integer, a function **fip** returning a pointer to an integer, and a pointer **pfi** to a function, which returns an integer. It is especially useful to compare the last two. The binding of ***fip()** is ***(fip())**. The declaration suggests, and the same construction in an expression requires, the calling of a function fip, and then using indirection through the (pointer) result to yield an integer. In the declarator **(*pfi)()**, the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called; it returns an integer.

As another example,

float fa[17], *afp[17];

declares an array of **float** numbers and an array of pointers to **float** numbers. Finally,

static int x3d[3][5][7];

declares a static 3-dimensional array of integers, with rank 3×5×7. In complete detail, **x3d** is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions **x3d**, **x3d[i]**, **x3d[i][j]**, **x3d[i][j][k]** may reasonably appear in an expression. The first three have type "array" and the last has type **int**.

Structure and Union Declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object that may, at a given time, contain any one of several members. Structure and union specifiers have the same form.

struct-or-union-specifier:
  struct-or-union { struct-decl-list }
  struct-or-union identifier { struct-decl-list }
  struct-or-union identifier

struct-or-union:
  struct
  union

The struct-decl-list is a sequence of declarations for the members of the structure or union:

struct-decl-list:
  struct-declaration
  struct-declaration struct-decl-list

struct-declaration:
  type-specifier struct-declarator-list ;

> *struct-declarator-list:*
> *struct-declarator*
> *struct-declarator , struct-declarator-list*

In the usual case, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a field; its length, a non-negative constant expression, is set off from the field name by a colon.

> *struct-declarator:*
> *declarator*
> *declarator : constant-expression*
> *: constant-expression*

Within a structure, the objects declared have addresses that increase as the declarations are read left to right. Each non-field member of a structure begins on an addressing boundary appropriate to its type; therefore, there may be unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field that does not fit into the space remaining in a word is put into the next word. No field may be wider than a word. (See Figure 17-2 for sizes of basic types on CADMUS computers.)

A struct-declarator with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, a field with a width of 0 specifies alignment of the next field at an implementation dependent boundary.

The language does not restrict the types of things that are declared as fields. Moreover, even **int** fields may be considered to be unsigned. For these reasons, it is strongly recommended that fields be declared as **unsigned** where that is the intent. There are no arrays of fields, and the address-of operator, &, may not be applied to them, so that there are no pointers to fields.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most, one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

> **struct** *identifier { struct-decl-list }*
> **union** *identifier { struct-decl-list }*

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

struct *identifier*
union *identifier*

Structure tags allow definition of self-referential structures. Structure tags also permit the long part of the declaration to be given once and used several times. It is illegal to declare a structure or union that contains an instance of itself, but a structure or union may contain a pointer to an instance of itself.

The third form of a structure or union specifier may be used prior to a declaration that gives the complete specification of the structure or union in situations in which the size of the structure or union is unnecessary. The size is unnecessary in two situations: when a pointer to a structure or union is being declared and when a **typedef** name is declared to be a synonym for a structure or union. This, for example, allows the declaration of a pair of structures that contain pointers to each other.

The names of members and tags do not conflict with each other or with ordinary variables. A particular name may not be used twice in the same structure, but the same name may be used in several different structures in the same scope.

A simple but important example of a structure declaration is the following binary tree structure:

```
struct tnode
{
        char tword[20];
        int count;
        struct tnode *left;
        struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the declaration

struct tnode s, *sp;

declares s to be a structure of the given sort and sp to be a pointer to a structure of the given sort. With these declarations, the expression

```
sp->count
```

refers to the **count** field of the structure to which sp points;

```
s.left
```

refers to the left subtree pointer of the structure **s**; and

```
s.right->tword[0]
```

refers to the first character of the **tword** member of the right subtree of **s**.

Enumeration Declarations

Enumeration variables and constants have integral type.

enum-specifier:
enum *{ enum-list }*
enum *identifier { enum-list }*
enum *identifier*

enum-list:
enumerator
enum-list , enumerator

enumerator:
identifier
identifier = constant-expression

The identifiers in an enum-list are declared as constants and may appear wherever constants are required. If no enumerators with = appear, then the values of the corresponding constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

The names of enumerators in the same scope must all be distinct from each other and from those of ordinary variables.

The role of the identifier in the enum-specifier is entirely analogous to that of the structure tag in a struct-specifier; it names a particular enumeration. For example,

```
enum color { chartreuse, burgundy, claret=20, winedark };
...
enum color *cp, col;
...
col = claret;
cp = &col;
...
if (*cp == burgundy) ...
```

makes color the enumeration-tag of a type describing various colors, and then declares cp as a pointer to an object of that type and col as an object of that type. The possible values are drawn from the set {0,1,20,21}.

Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by = and consists of an expression or a list of values nested in braces.

initializer:
  = *expression*
  = { *initializer-list* }
  = { *initializer-list* , }

initializer-list:
  expression
  initializer-list , *initializer-list*
  { *initializer-list* }
  { *initializer-list* , }

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in "Constant Expressions," or expressions that reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may be initialized by arbitrary expressions involving constants and previously declared variables and functions.

Static and external variables that are not initialized are guaranteed to start off as zero. Automatic and register variables that are not initialized are guaranteed to start off as garbage.

When an initializer applies to a scalar (a pointer or an object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

When the declared variable is an aggregate (a structure or array), the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with zeros. It is not permitted to initialize unions or automatic aggregates.

Braces may in some cases be omitted. If the initializer begins with a left brace, then the succeeding comma-separated list of initializers initializes the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

A final abbreviation allows a **char** array to be initialized by a string literal. In this case successive characters of the string literal initialize the members of the array.

For example,

```
int x[] = { 1, 3, 5 };
```

declares and initializes **x** as a one-dimensional array that has three members, since no size was specified and there are three initializers.

```
float y[4][3] =
{
     { 1, 3, 5 },
     { 2, 4, 6 },
     { 3, 5, 7 },
};
```

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array **y[0]**, namely **y[0][0]**, **y[0][1]**, and **y[0][2]**. Likewise, the next two lines initialize **y[1]** and **y[2]**. The initializer ends early and therefore **y[3]** is initialized with 0. Precisely, the same effect could have been achieved by

```
float y[4][3] =
{
     1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for y begins with a left brace but that for y[0] does not; therefore, three elements from the list are used. Likewise, the next three are taken successively for y[1] and y[2]. Also,

```
float y[4][3] =
{
     { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of y (regarded as a two-dimensional array) and leaves the rest 0.

Finally,

```
char msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string literal. The length of the string (or size of the array) includes the terminating NUL character, \0.

Type Names

In two contexts (to specify type conversions explicitly by means of a cast and as an argument of sizeof), it is desired to supply the name of a data type. This is accomplished using a "type name," which in essence is a declaration for an object of that type that omits the name of the object.

type-name:
  type-specifier abstract-declarator

abstract-declarator:
  empty
  (abstract-declarator)
  ** abstract-declarator*
  abstract-declarator ()
  abstract-declarator [*constant-expression*$_{opt}$]

To avoid ambiguity, in the construction

(abstract-declarator)

the abstract-declarator is required to be nonempty. Under this restriction, it is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int *[3]
int (*) [3]
int *()
int (*) ()
int (*[3]) ()
```

name respectively the types "integer," "pointer to integer," "array of three pointers to integers," "pointer to an array of three integers," "function returning pointer to integer," "pointer to function returning an integer," and "array of three pointers to functions returning an integer."

Implicit Declarations

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. The storage class is supplied by the context in external definitions and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be **int**; if a type but no storage class is indicated, the identifier is assumed to be **auto**. An exception to the latter rule is made for functions because **auto** functions do not exist. If the type of an identifier is "function returning . . . ," it is implicitly declared to be **extern**.

In an expression, an identifier followed by (and not already declared is contextually declared to be "function returning **int**."

typedef

Declarations whose "storage class" is **typedef** do not define storage but instead define identifiers that can be used later as if they were type keywords naming fundamental or derived types.

> *typedef-name:*
>   *identifier*

Within the scope of a declaration involving **typedef**, each identifier appearing as part of any declarator therein becomes syntactically equivalent to the type keyword naming the type associated with the identifier in the way described in "Meaning of Declarators." For example, after

```
typedef int MILES, *KLICKSP;
typedef struct { double re, im; } complex;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

are all legal declarations; the type of **distance** is **int**, that of **metricp** is "pointer to **int**," and that of **z** is the specified structure. The **zp** is a pointer to such a structure.

The **typedef** does not introduce brand-new types, only synonyms for types that could be specified in another way. Thus in the example above **distance** is considered to have exactly the same type as any other **int** object.

Statements

Except as indicated, statements are executed in sequence.

Expression Statement

Most statements are expression statements, which have the form

expression ;

Usually expression statements are assignments or function calls.

Compound Statement or Block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called "block") is provided:

compound-statement:
  { declaration-list$_{opt}$ statement-list$_{opt}$ }

declaration-list:
  declaration
  declaration declaration-list

statement-list:
  statement
  statement statement-list

If any of the identifiers in the declaration-list were previously declared, the outer declaration is pushed down for the duration of the block, after which it resumes its force.

Any initializations of **auto** or **register** variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of **static** variables are performed only once when the program begins execution. Inside a block, **extern** declarations do not reserve storage so initialization is not permitted.

Conditional Statement

The two forms of the conditional statement are

> **if** (*expression*) *statement*
> **if** (*expression*) *statement* **else** *statement*

In both cases, the expression is evaluated; if it is nonzero, the first substatement is executed. In the second case, the second substatement is executed if the expression is 0. The **else** ambiguity is resolved by connecting an else with the last encountered **else**-less **if**.

while Statement

The **while** statement has the form

> **while** (*expression*) *statement*

The substatement is executed repeatedly so long as the value of the expression remains nonzero. The test takes place before each execution of the statement.

do Statement

The **do** statement has the form

> **do** *statement* **while** (*expression*) ;

The substatement is executed repeatedly until the value of the expression becomes 0. The test takes place after each execution of the statement.

for Statement

The **for** statement has the form:

> **for** *(exp–1*$_{opt}$ *; exp–2*$_{opt}$ *; exp–3*$_{opt}$ *) statement*

Except for the behavior of **continue**, this statement is equivalent to

```
exp-1 ;
while ( exp-2 )
{
     statement
     exp-3 ;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0. The third expression often specifies an incrementing that is performed after each iteration.

Any or all of the expressions may be dropped. A missing *exp-2* makes the implied **while** clause equivalent to **while(1)**; other missing expressions are simply dropped from the expansion above.

switch Statement

The **switch** statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

switch (*expression*) *statement*

The usual arithmetic conversion is performed on the expression, but the result must be **int**. The statement is typically compound. Any statement within the statement may be labeled with one or more case prefixes as follows:

case *constant-expression* :

where the constant expression must be **int**. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in "Constant Expressions."

There may also be at most one statement prefix of the form

default :

which properly goes at the end of the case constants.

When the **switch** statement is executed, its expression is evaluated and compared in turn with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression and if there is a **default** prefix, control passes to the statement prefixed by **default**. If no case matches and if there is no **default**, then none of the statements in the switch is executed.

The prefixes **case** and **default** do not alter the flow of control, which continues unimpeded across such prefixes. That is, once a case constant is matched, all **case** statements (and the **default**) from there to the end of the **switch** are executed. To exit from a switch, see "**break** Statement."

Usually, the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, but initializations of automatic or register variables are ineffective. A simple example of a complete **switch** statement is:

```
switch (c) {
        case 'o':
                oflag = TRUE;
                break;
        case 'p':
                pflag = TRUE;
                break;
        case 'r':
                rflag = TRUE;
                break;
        default :
                (void) fprintf(stderr, "Unknown option\n");
                exit(2);
        }
```

break Statement

The statement **break ;** causes termination of the smallest enclosing **while**, **do**, **for**, or **switch** statement; control passes to the statement following the terminated statement.

continue Statement

The statement **continue ;** causes control to pass to the loop-continuation portion of the smallest enclosing **while, do,** or **for** statement; that is to the end of the loop. More precisely, in each of the statements

```
while (...)         do                  for (...)
{                   {                   {
        ...                 ...                 ...
contin: ;           contin: ;           contin: ;
}                   } while (...);      }
```

a **continue** is equivalent to **goto contin.** (Following the **contin:** is a null statement; see "Null Statement.")

return Statement

A function returns to its caller by means of the **return** statement, which has one of the forms

> **return ;**
> **return** *expression* ;

In the first case, the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

goto Statement

Control may be transferred unconditionally by means of the statement

goto *identifier* ;

The identifier must be a label (see "Labeled Statement") located in the current function.

Labeled Statement

Any statement may be preceded by label prefixes of the form

identifier :

which serve to declare the identifier as a label. The only use of a label is as a target of a **goto**. The scope of a label is the current function, excluding any sub-blocks in which the same identifier has been redeclared. See "Scope Rules."

Null Statement

The null statement has the form

;

A null statement is useful to carry a label just before the } of a compound statement or to supply a null body to a looping statement such as **while**.

External Definitions

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class **extern** (by default) or perhaps **static**, and a specified type. The type-specifier (see "Type Specifiers" in "Declarations") may also be empty, in which case the type is taken to be **int**. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations except that only at this level may the code for functions be given.

External Function Definitions

Function definitions have the form

function-definition:
  decl-specifiers$_{opt}$ *function-declarator function-body*

The only sc-specifiers allowed among the decl-specifiers are **extern** or **static**; see "Scope of Externals" in "Scope Rules" for the distinction between them. A function declarator is similar to a declarator for a "function returning . . . " except that it lists the formal parameters of the function being defined.

function-declarator:
  declarator (parameter-list$_{opt}$ *)*

parameter-list:
  identifier
  identifier , parameter-list

The function-body has the form

function-body:
  declaration-list$_{opt}$ *compound-statement*

The identifiers in the parameter list, and only those identifiers, may be declared in the declaration list. Any identifiers whose type is not given are taken to be **int**. The only storage class that may be specified is **register**; if it is specified, the corresponding actual parameter will be copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition is

```
int max(a, b, c)
        int a, b, c;
{
        int m;

        m = (a > b) ? a : b;
        return((m > c) ? m : c);
}
```

Here **int** is the type-specifier; **max(a, b, c)** is the function-declarator; **int a, b, c;** is the declaration-list for the formal parameters; **{ ... }** is the block giving the code for the statement.

The C program converts all **float** actual parameters to **double**, so formal parameters declared **float** have their declaration adjusted to read **double**. All **char** and **short** formal parameter declarations are similarly adjusted to read **int**. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared "array of . . . " are adjusted to read "pointer to . . . ."

External Data Definitions

An external data definition has the form

data-definition:
    declaration

The storage class of such data may be **extern** (which is the default) or **static**, but not **auto** or **register**.

Scope Rules

A C program need not all be compiled at the same time. The source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scopes to consider: first, what may be called the lexical scope of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

Lexical Scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the source file in which they appear. The lexical scope of identifiers that are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of a block persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also (see "Structure, Union, and Enumeration Declarations" in "Declarations") that tags, identifiers associated with ordinary variables, and identities associated with structure and union members form three disjoint classes which do not conflict. Members and tags follow the same scope rules as other identifiers. The **enum** constants are in the same class as ordinary variables and follow the same scope rules. The **typedef** names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
...
{
     int distance;
     ...
```

The **int** must be present in the second declaration, or it would be taken to be a declaration with no declarators and type **distance**.

Scope of Externals

If a function refers to an identifier declared to be **extern**, then somewhere among the files or libraries constituting the complete program there must be at least one external definition for the identifier. All functions in a given program that refer to the same external identifier refer to the same object, so care must be taken that the type and size specified in the definition are compatible with those specified by each function that references the data.

It is illegal to explicitly initialize any external identifier more than once in the set of files and libraries comprising a multi-file program. It is legal to have more than one data definition for any external non-function identifier; explicit use of **extern** does not change the meaning of an external declaration.

In restricted environments, the use of the **extern** storage class takes on an additional meaning. In these environments, the explicit appearance of the **extern** keyword in external data declarations of identities without initialization indicates that the storage for the identifiers is allocated elsewhere, either in this file or another file. It is required that there be exactly one definition of each external identifier (without **extern**) in the set of files and libraries comprising a multi-file program.

Identifiers declared **static** at the top level in external definitions are not visible in other files. Functions may be declared **static**.

Compiler Control Lines

The C compilation system contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with # communicate with this preprocessor. There may be any number of blanks and horizontal tabs between the # and the directive, but no additional material (such as comments) is permitted. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect that lasts (independent of scope) until the end of the source program file.

Token Replacement

A control line of the form

#define *identifier token-string*$_{opt}$

causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. Semicolons in or at the end of the token-string are part of that string. A line of the form

#define *identifier(identifier, ...) token-string*$_{opt}$

where there is no space between the first identifier and the (, is a macro definition with arguments. There may be zero or more formal parameters. Subsequent instances of the first identifier followed by a (, a sequence of tokens delimited by commas, and a) are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however, commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Strings and character constants in the token-string are scanned for formal parameters, but strings and character constants in the rest of the program are not scanned for defined identifiers to replace.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing \ at the end of the line to be continued. This facility is most valuable for definition of "manifest constants," as in

```
#define TABSIZE 100

int table[TABSIZE];
```

A control line of the form

#undef *identifier*

causes the identifier's preprocessor definition (if any) to be forgotten.

If a **#defined** identifier is the subject of a subsequent **#define** with no intervening **#undef**, then the two token-strings are compared textually. If the two token-strings are not identical (all white space is considered as equivalent), then the identifier is considered to be redefined.

File Inclusion

A control line of the form

#include *"filename"*

causes the replacement of that line by the entire contents of the file *filename*. The named file is searched for first in the directory of the file containing the **#include**, and then in a sequence of specified or standard places. Alternatively, a control line of the form

#include *<filename>*

searches only the specified or standard places and not the directory of the **#include**. (How the places are specified is not part of the language. See cpp(1) for a description of how to specify additional libraries.)

#includes may be nested.

Conditional Compilation

A compiler control line of the form

#if *restricted-constant-expression*

checks whether the restricted-constant expression evaluates to nonzero. (Constant expressions are discussed in "Constant Expressions"; the following additional restrictions apply here: the constant expression may not contain **sizeof**, casts, or an enumeration constant.)

A restricted-constant expression may also contain the additional unary expression

defined *identifier*

or

defined (*identifier*)

which evaluates to one if the identifier is currently defined in the preprocessor and zero if it is not.

All currently defined identifiers in restricted-constant-expressions are replaced by their token-strings (except those identifiers modified by **defined**) just as in normal text. The restricted-constant expression will be evaluated only after all expressions have finished. During this evaluation, all undefined (to the procedure) identifiers evaluate to zero.

A control line of the form

#ifdef *identifier*

checks whether the identifier is currently defined in the preprocessor; i.e., whether it has been the subject of a **#define** control line. It is equivalent to **#if defined** (*identifier*).

A control line of the form

#ifndef *identifier*

checks whether the identifier is currently undefined in the preprocessor. It is equivalent to **#if !defined** (*identifier*).

All three forms are followed by an arbitrary number of lines, possibly containing a control line

#else

and then by a control line

#endif

If the checked condition is true, then any lines between **#else** and **#endif** are ignored. If the checked condition is false, then any lines between the test and a **#else** or, lacking a **#else**, the **#endif** are ignored.

Another control directive is

#elif *restricted-constant-expression*

An arbitrary number of **#elif** directives can be included between **#if**, **#ifdef**, or **#ifndef** and **#else**, or **#endif** directives. These constructions may be nested.

Line Control

For the benefit of other preprocessors that generate C programs, a line of the form

#line *constant "filename"*

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the constant and the current input file is named by *"filename"*. If *"filename"* is absent, the remembered file name does not change.

Version Control

This capability, known as *S-lists*, helps administer version control information. A line of the form

#ident *"version"*

puts any arbitrary string in the **.comment** section of the **a.out** file. It is usually used for version control. It is worth remembering that **.comment** sections are not loaded into memory when the **a.out** file is executed.

On CADMUS systems **#ident** currently only works, if the environment variable DOIDENT is set to "y" or "yes", otherwise **#ident** directives are ignored.

Types Revisited

This part summarizes the operations that can be performed on objects of certain types.

Structures and Unions

Structures and unions may be assigned, passed as arguments to functions, and returned by functions. Other plausible operators, such as equality comparison and structure casts, are not implemented.

In a reference to a structure or union member, the name on the right of the –> or the . must specify a member of the aggregate named or pointed to by the expression on the left. In general, a member of a union may not be inspected unless the value of the union has been assigned using that same member. However, one special guarantee is made by the language in order to simplify the use of unions: if a union contains several structures that share a common initial sequence and if the union currently contains one of these structures, it is permitted to inspect the common initial part of any of the contained structures. For example, the following is a legal fragment:

```
union
{
        struct
        {
                int         type;
        } n;
        struct
        {
                int         type;
                int         intnode;
        } ni;
        struct
        {
                int         type;
                float       floatnode;
        } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
        ... sin(u.nf.floatnode) ...
```

Functions

There are only two things that can be done with a function: call it or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```
int f();
...
g(f);
```

Then the definition of **g** might read

```
g(funcp)
        int (*funcp)();
{
        ...
        (*funcp)();
        ...
}
```

Notice that **f** must be declared explicitly in the calling routine since its appearance in **g(f)** was not followed by **(**.

Arrays, Pointers, and Subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator **[]** is interpreted in such a way that **E1[E2]** is identical to ***((E1)+(E2))**. Because of the conversion rules that apply to **+**, if **E1** is an array and **E2** an integer, then **E1[E2]** refers to the **E2** -th member of **E1**. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multidimensional arrays. If **E** is an *n*-dimensional array of rank ixjx...xk, then **E** appearing in an expression is converted to a pointer to an (n–1)-dimensional array with rank jx...xk. If the ***** operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to (n–1)-dimensional array, which itself is immediately converted into a pointer.

For example, consider **int x[3][5];** Here **x** is a 3×5 array of integers. When **x** appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression **x[i]**, which is equivalent to ***(x+i)**, **x** is first converted to a pointer as described; then **i** is converted to the type of **x**, which involves multiplying **i** by the length the object to which the pointer points, namely 5-integer objects. The results are added and indirection applied to yield an array (of five integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript, the same argument applies again; this time the result is an integer.

Arrays in C are stored row-wise (last subscript varies fastest) and the first subscript in the declaration helps determine the amount of storage consumed by an array. Arrays play no other part in subscript calculations.

Explicit Pointer Conversions

Certain conversions involving pointers are permitted but have implementation-dependent aspects. They are all specified by means of an explicit type-conversion operator, see "Unary Operators" under "Expressions" and "Type Names" under "Declarations."

A pointer may be converted to any of the integral types large enough to hold it. Whether an **int** or **long** is required is machine dependent. The mapping function is also machine dependent but is intended to be unsurprising to those who know the addressing structure of the machine.

An object of integral type may be explicitly converted to a pointer. The mapping always carries an integer converted from a pointer back to the same pointer but is otherwise machine dependent.

A pointer to one type may be converted to a pointer to another type. The resulting pointer may cause addressing exceptions upon use if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given size may be converted to a pointer to an object of a smaller size and back again without change.

For example, a storage-allocation routine might accept a size (in bytes) of an object to allocate, and return a **char** pointer; it might be used in this way.

```
extern char *alloc();
double *dp;

dp = (double *) alloc(sizeof(double));
*dp = 22.0 / 7.0;
```

The **alloc** must ensure (in a machine-dependent way) that its return value is suitable for conversion to a pointer to **double**; then the use of the function is portable.

Constant Expressions

In several places C requires expressions that evaluate to a constant: after **case**, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, casts to integral types, enumeration constants, and **sizeof** expressions, possibly connected by the binary operators

```
+ - * / % & | ^ << >> == != < > <= >= && ||
```

or by the unary operators

```
- ~
```

or by the ternary operator

```
?:
```

Parentheses can be used for grouping but not for function calls.

More latitude is permitted for initializers; besides constant expressions as discussed above, one can also use floating constants and arbitrary casts and can also apply the unary & operator to external or static objects and to external or static arrays subscripted with a constant expression. The unary & can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

Portability Considerations

Certain parts of C are inherently machine dependent. The following list of potential trouble spots is not meant to be all-inclusive but to point out the main ones.

Purely hardware issues like word size and the properties of floating point arithmetic and integer division have proven in practice to be not much of a problem. Other facets of the hardware are reflected in differing implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are nuisances that must be carefully watched. Most of the others are only minor problems.

The number of **register** variables that can actually be placed in registers varies from machine to machine as does the set of valid types. Nonetheless, the compilers all do things properly for their own machine; excess or invalid **register** declarations are ignored.

The order of evaluation of function arguments is not specified by the language. The order in which side effects take place is also unspecified.

Since character constants are really objects of type **int**, multicharacter character constants may be permitted. The specific implementation is very machine dependent because the order in which characters are assigned to a word varies from one machine to another.

Fields are assigned to words and characters to integers right to left on some machines and left to right on other machines. These differences are invisible to isolated programs that do not indulge in type punning (e.g., by converting an **int** pointer to a **char** pointer and inspecting the pointed-to storage) but must be accounted for when conforming to externally-imposed storage layouts.

Syntax Summary

This summary of C syntax is intended more for aiding comprehension than as an exact statement of the language.

Expressions

The basic expressions are:

expression:

primary
* *expression*
& *lvalue*
– *expression*
! *expression*
~ *expression*
++ *lvalue*
— *lvalue*
lvalue ++
lvalue —
sizeof *expression*
sizeof (*type-name*)
(*type-name*) *expression*
expression binop expression
expression ? expression : expression
lvalue asgnop expression
expression , expression

primary:

identifier
constant
string literal
(*expression*)
primary (*expression-list*$_{opt}$)
primary [*expression*]
primary . *identifier*
primary –> *identifier*

lvalue:
    identifier
    primary [expression]
    lvalue . identifier
    primary -> identifier
    * *expression*
    (lvalue)

The primary-expression operators

```
() [] . ->
```

have highest priority and group left to right. The unary operators

```
* & - ! ~ ++ -- sizeof ( type-name )
```

have priority below the primary operators but higher than any binary operator and group right to left. Binary operators group left to right; they have priority decreasing as indicated below.

binop:

```
* / %
+ -
>> <<
< > <= >=
== !=
&
^
|
&&
||
```

The conditional operator groups right to left.

Assignment operators all have the same priority and all group right to left.

asgnop:

```
= += -= *= /= %= >>= <<= &= ^= |=
```

The comma operator has the lowest priority and groups left to right.

Declarations

declaration:
  decl-specifiers init-declarator-list$_{opt}$;

decl-specifiers:
  type-specifier decl-specifiers$_{opt}$
  sc-specifier decl-specifiers$_{opt}$

sc-specifier:
  auto
  static
  extern
  register
  typedef

type-specifier:
  struct-or-union-specifier
  typedef-name
  enum-specifier

basic-type-specifier:
  basic-type
  basic-type basic-type-specifiers

basic-type:
  char
  short
  int
  long
  unsigned
  float
  double
  void

enum-specifier:
  enum *{ enum-list }*
  enum *identifier { enum-list }*
  enum *identifier*

enum-list:
  enumerator
  enum-list , enumerator

enumerator:
  identifier
  identifier = constant-expression

init-declarator-list:
  init-declarator
  init-declarator , init-declarator-list

init-declarator:
  declarator initializer$_{opt}$

declarator:
  identifier
  (declarator)
  ** declarator*
  declarator ()
  declarator [constant-expression$_{opt}$ *]*

struct-or-union-specifier:
  struct *{ struct-decl-list }*
  struct *identifier { struct-decl-list }*
  struct *identifier*
  union *{ struct-decl-list }*
  union *identifier { struct-decl-list }*
  union *identifier*

struct-decl-list:
  struct-declaration
  struct-declaration struct-decl-list

struct-declaration:
  type-specifier struct-declarator-list ;

struct-declarator-list:
  struct-declarator
  struct-declarator , struct-declarator-list

struct-declarator:
    declarator
    declarator : constant-expression
    : constant-expression

initializer:
    = expression
    = { initializer-list }
    = { initializer-list , }

initializer-list:
    expression
    initializer-list , initializer-list
    { initializer-list }
    { initializer-list , }

type-name:
    type-specifier abstract-declarator

abstract-declarator:
    empty
    (abstract-declarator)
    ** abstract-declarator*
    abstract-declarator ()
    abstract-declarator [constant-expression$_{opt}$ *]*

typedef-name:
    identifier

Statements

compound-statement:
    { declaration-list$_{opt}$ *statement-list*$_{opt}$ *}*

declaration-list:
    declaration
    declaration declaration-list

statement-list:
  statement
  statement statement-list

statement:
  compound-statement
  expression ;
  if *(expression) statement*
  if *(expression) statement* **else** *statement*
  while *(expression) statement*
  do *statement* **while** *(expression) ;*
  for *(*exp_{opt}*;*exp_{opt}*;*exp_{opt}*) statement*
  switch *(expression) statement*
  case *constant-expression : statement*
  default *: statement*
  break ;
  continue ;
  return ;
  return *expression ;*
  goto *identifier ;*
  identifier : statement
  ;

External Definitions

program:
  external-definition
  external-definition program

external-definition:
  function-definition
  data-definition

function-definition:
  decl-specifier$_{opt}$ *function-declarator function-body*

function-declarator:
  declarator (parameter-list$_{opt}$ *)*

parameter-list:
identifier
identifier , parameter-list

function-body:
declaration-list$_{opt}$ *compound-statement*

data-definition:
extern *declaration* ;
static *declaration* ;

Preprocessor

#define *identifier token-string*$_{opt}$
#define *identifier*(*identifier*,...)*token-string*$_{opt}$
#undef *identifier*
#include "*filename*"
#include <*filename*>
#if *restricted-constant-expression*
#ifdef *identifier*
#ifndef *identifier*
#elif *restricted-constant-expression*
#else
#endif
#line *constant* "*filename*"
#ident "*version*"

Table of Contents

Chapter 17: A68 Assembler

Introduction

This chapter describes the syntax and usage of the **a68** assembler for the Motorola 68000 and 68020 microprocessors. Originally written for the MC68000 the assembler was later upgraded to handle the new addressing modes and instruction set extensions of MC68020. The **a68** remains completely upward compatible. The basic format of **a68** is loosely based on the Digital Equipment Corp Macro-11 assembler described in DEC's publication DEC-11-0MACA-A-D, but also contains elements of the UNIX *as* assembler. The instruction mnemonics and effective address format are derived from a Motorola publication on the 68000, *the MACSS MC68000 Design Specification Intruction Set Processor* dated June 30, 1979. All infomation required in the upgrading was obtained from *the MC68020 32-Bit Microprocessor User's Manual* dated 1984, also a Motorola Publication.

Sections 1-3 of this chapter describe the general form of **a68** programs, section 4 describes the instruction mnemonics and addressing modes, section 5 describes the pseudo-ops supported by the assembler, section 6 describes the coprocessor instructions and section 7 describes the error codes generated. For instructions on how to operate the assembler from MUNIX, consult the corresponding manual page **a68(1)**.

NOTATION

The notation used in this document is a modified BNF similar to that used in the MULTICS PL/I Language Manual. The operators of the BNF in order of decreasing precedence are:

x ... *Repetition: Denotes one or more occurrences of* **x**. **xy** *Juxtaposition: Denotes an occurrence of* **x** *followed by an occurrence of* **y**. **x|y** *Alternation: Denotes an occurrence of* **x** *or* **y** *but not both.*

Brackets and braces define the order of expression interpretation. The subexpression enclosed in brackets is optional. That is,

[x] *denotes zero or one occurrence of x.* **{x|y}z** *denotes an x or a y, followed by a z.*

Brackets or braces which appear in **a68** syntax will be boldfaced to distinguish them from the meta-brackets and braces.

SOURCE PROGRAM FORMAT

An **a68** program consists of a series of statements, each of which occupies exactly one line, i.e., a sequence of characters followed by the *newline* character. Form feed, ascii ^L, also serves as a line terminator. Neither multiple statements on a single line nor continuation lines are allowed.

The format of an **a68** assembly language statement is:

[LabelField :] op-code [OperandField] [|comment]

There are three exceptions to this rule:

- Blank lines are permitted.
- A statement may contain only a *LabelField*. The label defined in this field has the same value as if it were defined in the *LabelField* of the next statement in the program. For example, the two statements

  ```
  sea:
  movw d1,d2
  ```

  are equivalent to the single statement

  ```
  sea: movw d1,d2
  ```

- A line may consist of only the comment field. For example, the two statements below are allowed:

  ```
  | This is a comment field.
  | So is this
  ```

In general, blanks or tabs are allowed anywhere in a statement. For example, multiple blanks are allowed in the *OperandField* to separate symbols from operators. Blanks are meaningful only when they occur in a character string (e.g. as the operand of an *.ascii* pseudo-op). At least one blank or tab must appear between the op-code and the *OperandField* of a statement.

Label Fields

A label is a user-defined symbol which is assigned the value of the current location counter and entered into the assembler's symbol table. The value of the label may be either absolute or relocatable; in the latter case, the absolute value of the symbol is assigned when the program is linked via *ld*.

A label is a symbolic means of referring to a specific location within a program. If present, a label **always** occurs first in a statement and **must** be terminated by a colon. The collection of label definitions in a statement is called the *LabelField.*

The format of a *LabelField* is:

symbol: [symbol:] . . .

Examples:

```
start:
sea: bar:    / Multiple symbols
7$:          / A local symbol, defined below
```

Op-code Fields

The *OpcodeField* of an assembly language statement identifies the statement as either a machine instruction, or an assembler directive. One or more blanks (or tabs) must separate the *OpcodeField* from the *OperandField* in a statement. No blanks are necessary between *LabelFields* and *OpcodeFields*, but they are recommended to improve readability of the program.

A machine instruction is indicated by an instruction mnemonic. The assembly language statement is intended to produce a single executable machine instruction. The operation of each instruction is described in the manufacturer's user manual. Some conventions used in **a68** for instruction mnemonics are described in section 4 and a complete list of the instructions is presented in the appendix.

An assembler directive, or pseudo-op, performs some function during the assembly process. It does not produce any executable code, but it may assign space in a program for data.

Operand Fields

A distinction is made between *OperandField* and *operand* in **a68**. Several machine instructions and assembler directives require two or more arguments, and each of these is referred to as an *operand*. In general, an *OperandField* consists of zero or more operands, **and in all cases, operands are separated by a comma.** In other words, the format for an *OperandField* is:

[operand [, operand] . . .]

The format of the *OperandField* for machine instruction statements is the same for all instructions, and is described in section 4. The format of the *OperandField* for assembler directives depends on the directive itself, and is included in the directive's description in section 5 of this manual.

Comment Field

The comment character in **a68** is the vertical bar, (|), not the semicolon, (;). Use of the semicolon as a comment character will result in an "Invalid Operator" error.

The comment field consists of all characters on a source line following and including the comment character. These characters are ignored by the assembler. Any character may appear in the comment field, with the obvious exception of the *newline* character, which starts a new line.

A|- line switches listing off.
A|+ line switches listing on.

SYMBOLS AND EXPRESSIONS

This section describes the various components of a68 expressions: symbols, numbers, terms, and expressions.

Symbols

A symbol consists of a sequence of characters, with the following restrictions:

- Valid characters include A-Z, a-z, 0-9, period (.), underscore (_), and dollar sign ($).
- The first character must not be numeric.

All characters are significant and are checked in comparisons with other symbols. Upper and lower cases are distinct, ("One" and "one" are separate symbols).

A symbol is declared when the assembler recognizes it as a symbol of the program. A symbol is defined when a value is associated with it. With the exception of symbols declared by a *.globl* directive, all symbols are defined when they are declared. A label symbol (which represents an address of the program) may not be redefined; all other symbols are allowed to receive a new value.

There are several ways to declare a symbol:

- As the label of a statement (See section 2.1).
- In a *direct assignment* statement.
- As an *external* symbol via the *.globl* directive.
- As a *common* symbol via the *.comm* directive.
- As a *local* symbol.

Direct Assignment Statements

A *direct assignment* statement assigns the value of an arbitrary expression to a specified symbol. The format of a *direct assignment* statement is:

```
symbol = expression
```

Examples of valid *direct assignments* are:

```
vect_size = 4
vectora = 0xFFFE
vectorb = vectora - vect_size
CRLF = 0x0D0A
```

Only one symbol may be assigned in a single statement.

Any symbol defined by *direct assignment* may be redefined later in the program, in which case its value is the result of the last such statement. A *local* symbol may be defined by *direct assignment*, though this doesn't make much sense. Label or register symbols may not be redefined.

If the *expression* is absolute, then the symbol is also absolute, and may be treated as a constant in subsequent expressions (see section 3.4). If the *expression* is relocatable, however, then the symbol is also relocatable, and it is considered to be declared in the same program section as the *expression*. See section 3.11 for an explanation of absolute and relocatable expressions.

If the *expression* contains an *external* symbol, then the symbol defined by the = statement will also be considered *external*. For example:

```
.globl x | x is declared as external symbol
sum = x  | sum becomes an external symbol
```

assigns the value of x (zero if it is undefined) to sum and makes sum an *external* symbol. *External* symbols may be defined by *direct assignment*.

Register Symbols

Register symbols are symbols used to represent registers in the machine. Register symbols are defined in the source descriptor file for a machine in the pre-assembly code portion of the file. This portion consists of source statements that are assembled before every source program for the machine.

The following are register symbols for the MC68000.

d0	d1	d2	d3	d4	d5	d6	d7
a0	a1	a2	a3	a4	a5	a6	a7
sp	pc	cc	sr	usp			

The following are register symbols for the MC68020.

d0	d1	d2	d3	d4	d5	d6	d7
a0	a1	a2	a3	a4	a5	a6	a7
sp	pc	cc	sr	usp	sfc	cacr	dfc
vbr	caar	msp	isp				

For the meaning of the register symbols see section 4.2 .

External Symbols

A program may be assembled in separate modules, and then linked together to form a single program (see *ld* (I)). *External* symbols may be defined in each of these separate modules. A symbol which is declared (given a value) in one module may be referenced in another module by declaring the symbol to be *external* in both modules. There are two forms of *external* symbols: those defined with the *.globl* and those defined with the *.comm* directive.

External symbols are declared with the *.globl* assembler directive. The format is:

```
.globl symbol [, symbol] . . .
```

For example, the following statements declare the array TABLE and the routine SRCH to be *external* symbols:

```
.globl TABLE,SRCH
TABLE: .=.+20
SRCH: movl #TABLE,a0
. . .
```

External symbols are only **declared** to the assembler. *External* symbols must be **defined** (i.e. given a value) in some other statement by one of the methods mentioned above. They need not be defined in the current program; in this case they are flagged as "undefined" in the symbol table. If they are undefined, they are considered to have a value of zero in expressions.

The other form of *external* symbol is defined with the *.comm* directive. These statement reserve storage in the bss section similar to FORTRAN common areas. The format of the statement is:

.comm name, ConstantExpression

which causes **a68** to declare the *name* as a *common* symbol with a value equal to the *ConstantExpression.* For the rest of the assembly this symbol will be treated as though it was an undefined global. **a68** does not allocate storage for *common* symbols; this task is left to the loader. The loader will compute the maximum size of for each *common* symbol which may appear in several load modules, allocates storage for it in the final *bss* section and resolves linkages.

Local Symbols

Local symbols provide a convenient means of generating labels for branch instructions, etc. Use of *local* symbols reduces the possibility of multiply defined symbols in a program, and separates entry point symbols from local references, such as the top of a loop. Local symbols cannot be referenced by other object modules.

Local symbols are of the form **n$** where **n** is any integer. The following are valid *local* symbols

1$
27$
394$

A *local* symbol is defined and referenced only within a single "local symbol block" (lsb). A new local symbol block is entered when:

- a label is declared; or,
- a new program section is entered.

There is no conflict between *local* symbols with the same name which appear in different local symbol blocks.

Assembly Location Counter

The assembly location counter is the period character, '.'; hence its name "dot". When used in the operand field of any statement, dot represents the address of the first byte of the statement. Even in assembly directives, it represents the address of the start of the directive. A dot appearing as the third argument in a *.byte* instruction has the value of the address where the first byte was loaded; this address is not updated "during" the pseudo-op.

For example,

```
Ralph: movl .,a0 |load value of this instruction into a0
```

At the beginning of each assembly pass, the assembler clears the location counter. Normally, consecutive memory locations are assigned to each byte of generate code. However, the location where the code is stored may be changed by a *direct assignment* altering the location counter:

```
. = expression
```

This *expression* must not contain any forward references, and must not change from one pass to another. Storage area may also be reserved by advancing dot. For example, if the current value of dot is 1000, the *direct assignment* statement:

```
Table: .=.+100
```

would reserve 100 (decimal) bytes of storage, with the address of the first byte as the value of Table. The next instruction would be stored at address 1100.

Program Sections

As in UNIX, programs to **a68** are divided into sections: *text*, *data*, *bss* and *dummy*. The normal interpretation of these sections is: instruction space, initialized data space and uninitialized data space, respectively. These three sections are equivalent as far as **a68** is concerned with the exception that no instructions or data will be output for the *bss* section although its size will be computed and its symbol values will be output.

In the first pass of the assembly, **a68** maintains a separate location counter for each section, thus for code like:

```
 .text
sum: movw d1,d2
 .data
hello: .long 27
 .text
jnk: addw d2,d1
 .data
myst: .byte 4
```

in the output, sum will immediately precede jnk and hello will immediately precede **myst**. At the end of the first pass, **a68** rearranges all the addresses so that the sections will be output in the following order: *text*, *data* and *bss*. The resulting output file is an executable image file with all addresses correctly resolved, with the exception of undefined *.globls* and *.comms*. For more information on the format of the output file, consult the UNIX *man* entry on *a.out* files.

The MC68020 version of this assembler produces standard UNIX COFF object files (common object file format). The MC68000 version of this assembler has both a COFF and a Version 7 object file form.

The following code:

```
 .dorg
 .=0x400
lab1: .=.+4
lab2: word 0
```

will assign **0x400** to **lab1**, **0x404** to **lab2**. Both are absolute; no code is generated.

Constants

All constants are considered absolute quantities when appearing in an expression.

Numeric Constants

An **a68** numeric constant is a sequence of digits. **a68** interprets integers as octal, hex, or decimal according to the following conventions.

octal	octal numbers begin with 0
hex	hex numbers begin with 0x or 0X
decimal	all other numbers

Operators

Unary Operators

There are two unary operators in **a68**:

-	unary minus.
~	logical negation.

Binary Operators

Binary operators in **a68** include:

+	Addition; e.g. "3+4" evaluates to 7.
-	Subtraction; e.g. "3-4" evaluates to -1., or 0xFFFFFFFF
*	Multiplication; e.g. "4*3" evaluates to 12.
/	division; e.g. "12/4" evaluates to 3.
%	modulo; e.g. "7%3" evaluates to 1.
&	binary/logical "and"; e.g. "5&3" evaluates to 1.
!	binary/logical "or"; e.g. "5!3" evaluates to 7.

Each operator is assumed to work on a 32-bit number. Furthermore "[" and "]" can be used as parenthesies.

Terms

A term is a component of an expression. A term may be one of the following:

- A number.
- A symbol.
- A term preceded by a unary operator. For example, both "sum" and "~sum" may be considered to be terms. Multiple unary operators are allowed; e.g. " -- A" has the same value as "A".

Expressions

Expressions are combinations of terms joined together by binary operators. An expression is always evaluated to a 32-bit value. If the instruction calls for only one byte, (e.g. *.byte*), then the low-order byte is used.

Expressions are evaluated left to right with operator precedence. Thus "1+2*3" evaluates to 7. Unary operators have precedence over binary operators since they are considered part of a term, and both terms of a binary operator must be evaluated before the binary operator can be applied. Parenthesies (i.e. "[" and "]") have the highest priority, "*" and "/" follows, and "+" and "-" have the lowest priority. Parenthesis may be nested. An expression may have up to 10 operands and the sum of operands and operators must not exceed 30.

A missing expression or term is interpeted as having a value of zero. In this case, an "Invalid expression" error will be generated. An "Invalid Operator" error means that a valid end-of-line character or binary operator was not detected after the assembler processed a term. In particular, this error will be generated if an expression contains a symbol with an illegal character, or if an incorrect comment character was used.

Any expression, when evaluated, is either absolute, relocatable, or external:

- An expression is absolute if its value is fixed. An expression whose terms are constants, or symbols whose values are constants via a *direct assignment* statement, is absolute. A relocatable expression minus a relocatable term, where both items belong to the same program section is also absolute.
- An expression is relocatable if its value is fixed relative to a base address, but will have an offset value when it is linked, or loaded into core. All labels of a program defined in relocatable sections are relocatable terms, and any expression which contains them must only **add or subtract constants to their value.** For example, assume the symbol "sum" was defined in a relocatable section of the program. Then the following demonstrates the use of relocatable expressions:

sum	relocatable
sum+5	relocatable
sum*2	Not relocatable (error)
2-sum	Not relocatable (error), since the expression cannot be linked by adding sum's offset to it.
sum-jnk	Absolute, since the offsets added to sum and jnk cancel each other out.

- An expression is *external* (or global) if it contains an *external* symbol not defined in the current program. The same restrictions on expressions containing relocatable symbols apply to expressions containing *external* symbols. Exception: the expression **sum-jnk** where both **sum** and **jnk** are *external* symbols is not allowed.

INSTRUCTIONS AND ADDRESSING MODES

This section describes the conventions used in **a68** to specify instruction mnemonics and addressing modes.

Instruction Mnemonics

The instruction mnemonics used by **a68** are described in the previously mentioned Motorola manuals with a few variations. The 68020 instructions can apply to byte, word, long or even quad word operands (68000 instructions have no quad word operands), so in **a68** the normal instruction mnemonic is suffixed with **b**, **w**, **l**,or **q** to indicate which length operand was intended (presently only **divu** and **divs** on the 68020 support quad word operands). For example, there are three mnemonics for the *or* instruction: *orb*, *orw* and *orl*. Op-codes for instructions with unusual opcodes may have additional suffixes, thus in addition to the normal add variations, there also exist: *addqb*, *addqw* and *addql* for the add quick instruction. The assembler will use the *addq*, *subq* and *movq* instead of *add*, *sub* and *mov* wherever possible. However, if the operand is a difference of two labels of the same section (i.e. an immediate operand), this optimization is not possible and the assembler will use the long instruction. In any case, the instructions *movq*, ... are still available.

Branch instructions come in three flavors, byte, word and long. In **a68**, the byte (i.e., short) version is specified by appending the suffix s to the basic mnemonic as in *beq* and *beqs*. Similarily the long and word versions are specified by appending the suffixes **l** and **w**. In the absence of a suffix, **w** is assumed to allow for upwards compatibility between the two versions of the **a68**.

In addition to the instructions which explicitly specify the instruction length, **a68** supports extended branch instructions, whose names are generally constructed by replacing the **b** with **j**. If the operand of the extended branch instruction is a simple address in the current segment, and the offset to that address is sufficiently small, **a68** will automatically generate the corresponding short branch instruction. If the offset is too large for a short branch, but small enough for a word branch, then the corresponding word branch instruction is generated and so on. If the operand references an external address or is complex, then the extended branch instruction is implemented either by a *jmp* or *jsr* (for *jra* or *jbsr*), or by a conditional branch (with the sense of the conditional inverted) around a *jmp* for the extended conditional branches. In this context, a complex address is either an address which specifies other than normal mode addressing, or relocatable expressions containing more than one relocatable symbol. i.e. if **a**, **b** and **c** are symbols in the current segment, then the expression **a+b-c** is relocatable, but not simple.

Note that **a68** is not optimal for extended branch instructions whose operand addresses the next instruction. The optimal code is no instruction at all, but **a68** currently retains insufficient information to make this optimization. The difficulty is that if **a68** decides to just eliminate the instruction, the address of the next instruction will be the same as the address of the (nonexistent) extended branch instruction. This instruction will then look like a branch to the current location, which would require an instruction to be generated. The code that **a68** actually generates for this case is a nop (recall that an offset of zero in a branch instruction indicates a long offset). Although this problem may arise in compiler code generators, it can easily be handled by a peephole optimizer.

The algorithm used by **a68** for deciding how to implement extended branch instructions is described in ''Assembling Code for Machines with Span Dependent Instruction,'' by Thomas G. Szymanski in "Communications of the ACM", Volume 21, Number 4, pp300-308, April 1978.

Consult the appendix for a complete list of the instruction op-codes.

Addressing Modes

The following table describes the addressing modes recognized by **a68**. In this table are the following symbols:

an	-address register
dn	-data register
Ri	-data register or address register
d	-displacement (constant expression, 8, 16, or 32 bits)
bd	-base displacement (constant expression, 0, 16, or 32 bits)
od	-outer displacement (constant expression, 0, 16, or 32 bits)
Sc	-scale factor (valid values 1, 2, 4, 8)
xxx	-constant expression.
off	-bit field offset (constant expression, 0 <= off <= 31)
wid	-bit field width (constant expression, 0 <= wid <= 31)
<ea>	-effective address.

Certain instructions, particularly *move* accept a variety of special registers including **sp**, the stack pointer which is equivalent to **a7**, **sr** †, the status register, † The access to **sr** is restricted to specific instructions. **cc**, the condition codes of the status register, **usp**, the user mode stack pointer, **pc**, the program counter, **sfc**, the source function code register, **dfc**, destination function code register, **cacr**, the cache control register, **vbr**, the vector base register, **caar**, the central address register, **msp**, the master stack pointer, and **isp**, the interrupt stack pointer.

Addressing Modes for the MC68000:

Mode	Notation	Motorola Syntax
Register	an,dn,sp,pc,cc,sr, usp	an,dn,sp,pc,cc,sr, usp
Reg Deferred	an@	(an)
Postincrement	an@+	(an)+
Predecrement	an@-	-(an)
Displacement	an@(d)	(d,an)
Word Index	an@(bd,Ri:W)	(bd,an,Ri.w)
Long Index	an@(bd,Ri:L)	(bd,an,Ri.l)
Absolute Short	xxx:W‡	xxx
Absolute Long	xxx:L	xxxxxx
PC Displacement	pc@(d)	(d,pc)
PC Word Index	pc@(bd,Ri:W)	(bd,pc,Ri.w)
PC Long Index	pc@(bd,Ri:L)	(bd,pc,Ri.l)
Normal	sun	sun
Immediate	#xxx	#xxx

Addressing Modes extensions of the MC68020 over the MC68000:

Mode	Notation	Motorola Syntax
Register	an,dn,sp,pc,cc,sr, usp,sfc,dfc,cacr vbr,caar,msp,isp	an,dn,sp,pc,cc,sr, usp,sfc,dfc,cacr vbr,caar,msp,isp
Scaled Long Index	an@(bd,Ri:L*Sc)	(bd,an,Ri.l*Sc)
Scaled Word Index	an@(bd,Ri:W*Sc)	(bd,an,Ri.w*Sc)
Memory Indirect Post-Indexed(Word)	an@(bd)@(od,Ri:W*Sc)	([bd,an],Ri.w*Sc,od)
Memory Indirect Pre-Indexed(Long)	an@(bd,Ri:L*Sc)@(od)	([bd,an,Ri.l*Sc],od)
Memory Indirect No Indexing	an@(bd)@(od)	([bd,an,],od)
PC Scaled Long Index	pc@(bd,Ri:L*Sc)	(bd,pc,Ri.l*Sc)
PC Scaled Word Index	pc@(bd,Ri:W*Sc)	(bd,pc,Ri.w*Sc)
PC Memory Indirect Post-Indexed(Word)	pc@(bd)@(od,Ri:W*Sc)	([bd,pc],Ri.w*Sc,od)
PC Memory Indirect Pre-Indexed(Long)	pc@(bd,Ri:L*Sc)@(od)	([bd,pc,Ri.l*Sc],od)
PC Memory Indirect No Indexing	pc@(bd)@(od)	([bd,pc],od)
Bit Field	<ea>{off:wid}	<ea>{off:wid}

‡ MUNIX LD(1) does not support short external addresses

All displacements can be different sized. For example **d** will be implemented as either 8, 16, or 32 bits depending on its value. Also, both **od** and **bd** will be implemented as either 0, 16, or 32 bits. If the value of **od** is zero then the outer displacement will be suppressed and will not be included in the opcode (i.e. 0 bits). Specifying a zero outer displacement is effectively omitting it. Similarily for base displacement.

All addressing modes that include indexing may be of either long (Ri:L) or word (Ri:W) index type. They may also omit the scale factor which will then automatically take on the default value of 1 (i.e. no scaling). Please note that the scale factor itself Sc, if present, must be either 1, 2, 4, or 8, and is multiplied with the index register to produce the index value.

Memory indirect addressing basically uses the base register (an or pc) along with the first set of brackets to calculate an intermediate memory address. Then using an address operand fetched from that intermediate memory address along with the second set of brackets the final effective address is calculated.

The Motorola manual presents different opcodes for instructions that use the effective address as data rather than the contents of the effective address such as *adda* for add address. **a68** does not make this distinction because it can determine the type of the operand from its form. Thus an instruction of the form:

```
sun: .word 0
   ...
   addl #sun,a0
```

will assemble to the *add address* instruction because **sun** is known to be an address.

The MC68000 family tends to be very restrictive in that most instructions accept only a limited subset of the address modes above. For example, the add address instruction does not accept a data register as a destination. **a68** tries to check all these restrictions and will generate the illegal operand error code for instructions that do not satisfy the address mode restrictions.

ASSEMBLER DIRECTIVES

The following pseudo-ops are available in **a68**:

.ascii	stores character strings
.asciz	"
.byte	stores 8-bit bytes
.word	stores 16-bit words
.long	stores 32-bit longwords
.zerol	long zeroes
.text	Text csect
.data	Data csect
.bss	Bss csect
.globl	declares external symbols
.extern	same as **.globl**
.comm	declares common symbols
.dorg	Dummy segment

.ascii .asciz

The *.ascii* directive translates character strings into their 7-bit ascii (represented as 8-bit bytes) equivalents for use in the source program. The format of the ascii directive is as follows:

```
        .ascii  "character string"
```

The syntax of C-strings is used (\n is newline, etc.). Obviously, a *newline* must not appear within the character string.

The *.asciz* directive is equivalent to the *.ascii* directive with a zero byte automatically inserted as the final character of the string. Thus, when a list or text string is to be printed, a search for the null character can terminate the string.

.byte .word .long

The *.byte*, *.word* and *.long* directives are used to reserve bytes and words, and to initialize them with certain values.

The format is:

```
[label:]     .byte [expression] [,expression]. .
                                    .
[label:]     .word [expression]  [,expression]..
                                    .
[label:]     .long [expression]  [,expression]..
                                    .
```

For example, the first statement reserves one byte for each expression in the operand field, and initializes the value of the byte to be the low-order byte of the corresponding expression. Note that multiple expressions must be separated by commas. A blank expression is interpreted as zero, and no error is generated.

The syntax and semantics for *.word* is identical, except that 16-bit words are reserved and initialized, of course, and *.long* uses 32-bit quantities.

.text .data .bss .dorg

These statements change the "program section" where assembled code will be loaded.

.globl .comm

See section 3.4.

.even

This directive advances the location counter if its current value is odd. This is useful for forcing storage allocation like *.word* directives to be on word boundaries.

COPROCESSOR INSTRUCTIONS

The following coprocessor instructions are available in **a68** (MC68020 version) :

cpbcc	Specified coprocessor condition is tested and if met, program execution continues at address given.
cpdbcc	Specified coprocessor condition is tested and if met, program execution continues at next instruction. If not met, then low order word of count register is decremented, and if result is not -1, then program execution continues at address given.
cpgen	Pass the command word to the coprocessor. This instruction is used by coprocessors to specify general data processing and movement operations. The specific coprocessor operation is determined by the command word (second word of the first operand) of the instruction.
cprestore	Restore the internal state of the coprocessor.
cpsave	Save the internal state of the coprocessor.
cpscc	Set the byte specified by the effective address to all ones if the coprocessor condition is met. Otherwise set it to all zeros.
cptrapcc	If the coprocessor condition is true, then the processor initiates exception processing. The immediate data, if present, is to be used by the trap handler routine.

The **a68** uses the operands of the coprocessor instructions to produce the final opcode. All the necessary information must be present in these operands, and it must be in the correct form. The **a68** does very little error checking on the operands of coprocessor instructions.

The first operand of a coprocessor instruction is either a word or long word of immediate data representing the opcode of the instruction, minus any addressing or coprocessor parameter information. The second operand, if present, is the effective address, or, for a cptrapcc instruction, it is a parameter for the exception handling routine.

Therefore when using coprocessor instructions the **a68** expects a certain amount of preparation. This preparation can be done manually, or via some pre-processor. All information required for this preparation can be found in the *MC68020 32-Bit Microprocessor User's Manual* and the corresponding coprocessor user's manual (e.g. the *MC68881 Floating-Point Coprocessor User's Manual*).

ERROR CODES

If an error is detected during assembly, a message of the form:

line_no .error_code

is output to the standard error stream.

The following *.error_codes*, and their probable cause, appear below:

- **2**
  Invalid Character. An invalid character for a character constant or character string was encountered.
- **3**
  Multiply defined symbol. A symbol appears twice as a label, or an attempt to redefine a label using an = statement.
- **5**
  Invalid offset.
- **7**
  Undefined symbol. A symbol not declared by one of the methods mentioned above in 'Symbols' was encountered. This happens when an invalid instruction mnemonic is used. This also occurs when an invalid or non-printing character occurs in the statement.
- **8**
  Invalid Constant. An invalid digit was encountered in a number.
- **9**
  Invalid Term. The expression evaluator could not find a valid term: symbol, constant or expression. An invalid prefix to a number or a bad symbol name in an operand will generate this.
- **10**
  Invalid Operator. Check the operand field for a bad operator.
- **11**
  Non-relocatable expression. If an expression contains a relocatable symbol (e.g. label) then the only operations that can be applied to it are the addition of absolute expressions or the subtraction of another relocatable symbol (which produces an absolute result).

- 13
  Invalid operand. This is a catch-all *.error*. It appears notably when an attempt is made to assign an undefined value to dot during pass 1.
- 14
  Invalid symbol. If the first token on the source line is not a valid symbol (or the beginning of a comment), this is generated. Might happen if you try and implied *.word*.
- 15
  Invalid assignment. An attempt was made to redefine a label with an = statement.
- 17
  Invalid op-code. An op-code mnemonic was not recognized by the assembler.
- 25
  Wrong number of operands. This is usually a warning. Check the manufacturer's assembly manual for the correct number of operands for the current instruction.
- 27
  Invalid register expression. Any expression inside parentheses should be absolute and have the value of a register code (register symbols). This may occur if you use parentheses for anything other than the register portion of an operand.
- 43
  Odd address. The address where an op-code will be put is odd. Take care that instructions will be put in even addresses.
- 44
  Operand relocation may not be handled by the loader.
- 45
  Unbalanced parenthesis. Take care that each "[" in an expression has a corresponding "]".

Appendix

Mnemonic	Description
abcd	add decimal with extend.
addL	add.
addqL	add quick.
addxL	add extended.
andL	logical and.
aslL	arithmetic shift left.
asrL	arithmetic shift right.
bCCS	branch on condition.
bchg	test bit & change.
bclr	test bit & clear.
bfchg	test bit field & change (MC68020).
bfclr	test bit field & clear (MC68020).
bfexts	signed bit field extract (MC68020).
bfextu	unsigned bit field extract (MC68020).
bfffo	bit field find first one (MC68020).
bfins	bit field insert (MC68020).
bfset	test bit field & set (MC68020).
bftst	test bit field (MC68020).
braS	branch.
bset	test bit & set.
bsrS	branch to subroutine.
btst	test bit.
callm	call module (MC68020).
casL	compare & swap operands (MC68020). "casL Dc,Du,<ea>"†
cas2L	compare & swap dual operands (MC68020). "cas2L Dc1:Dc2,Du1:Du2,<ea>"†
chkw	check register against word bound (chk).
chkl	check register against long bound.
chk2L	check register against upper & lower bounds (MC68020).
clrL	clear an operand.

† c: compare operand, u: update operand

Mnemonic	Description
cmpL	compare.
cmpmL	compare memory to memory.
cmp2L	compare register against upper & lower bounds (MC68020).
cpbcc	coprocessor branch conditionally (MC68020). "cpbcc #word, label"
cpdbcc	test coprocessor condition and count (MC68020). "cpdbcc #lword, label"
cpgen	coprocessor general function (MC68020). "cpgen #lword" "cpgen #lword, <ea>"
cprestore	coprocessor restore function (MC68020). "cprestore #word, <ea>"
cpsave	coprocessor save function (MC68020). "cpsave #word, <ea>"
cpscc	coprocessor set according to condition (MC68020). "cpscc #lword, <ea>"
cptrapcc	trap on coprocessor condition (MC68020). "cptrapcc #lword" "cptrapcc #lword, #word" "cptrapcc #lword, #lword"
dbCC	test condition, decrement & branch.
divs	signed divide
divsw	signed divide, long word dividend, word divisor, 32/16->16r:16q ‡ (MC68020) "divsw <ea>,Dn"
divsl	signed divide, long word dividend, long word divisor, 32/32->32q‡ (MC68020) "divsl <ea>,Dq"†
divsl	signed divide, long word dividend, long word divisor, 32/32->32r:32q ‡ (MC68020) "divsl <ea>,Dr:Dq"†

‡ q: quotient of operation, r: remainder of operation † Dq: quotient data register, Dr: remainder data register

Mnemonic	Description
divsq	signed divide, quad word dividend, long word divisor, 64/32->32r:32q ‡ (MC68020) "divsq <ea>,Dr:Dq"†
divu	unsigned divide
divuw	unsigned divide, long word dividend, word divisor, 32/16->16r:16q ‡ (MC68020) "divuw <ea>,Dn"
divul	unsigned divide, long word dividend, long word divisor 32/32->32q, "divul <ea>,Dq" ‡ (MC68020) 32/32->32r:32q, "divul <ea>,Dr:Dq"†
divuq	unsigned divide, quad word dividend, long word divisor, 64/32->32r:32q ‡ (MC68020) "divuq <ea>,Dr:Dq"†
eorL	logical exclusive or.
exg	exchange registers.
extb	sign extend byte to long word (MC68020).
extl	sign extend word to long word.
extw	sign extend byte to word.
jbsr	jump to subroutine.
jCC	jump on condition.
jra	jump.
jsr	jump to subroutine.
lea	load effective address.
link	link and allocate.
lslL	logical shift left.
lsrL	logical shift right.
movL	move.
movec	move control register (MC68020).
moveml	move multiple registers.
movemw	move multiple registers.
movepl	move peripheral.
movepw	move peripheral.

† s: source operand, d: destination operand ‡ Dh: data register (high 32 bits), Dl: data register (low 32 bits)

Mnemonic	Description
moveq	move quick.
muls	signed multiply
mulsl	signed multiply(MC68020)
	32s*32d->32d, "mulsl <ea>,Dl" †
	32s*32d->64d, "mulsl <ea>,Dh:Dl"‡
mulsw	signed multiply(MC68020)
	16s*16d->32d ,"mulsw <ea>,Dn"†
mulu	unsigned multiply
mulul	unsigned multiply(MC68020)
	32s*32d->32d, "mulul <ea>,Dl" †
	32s*32d->64d, "mulul <ea>,Dh:Dl" ‡
muluw	unsigned multiply(MC68020)
	16s*16d->32d, "muluw <ea>,Dn" †
nbcd	negate decimal with extend.
negL	negate.
negxL	negate with extend.
nop	no operation.
notL	logical compliment.
orL	logical inclusive or.
pack	pack BCD (MC68020).
pea	push effective address.
reset	reset external devices.
rolL	rotate left.
rorL	rotate right.
roxlL	rotate left with extend.
roxrL	rotate right with extend.
rte	return from exception.
rtm	return from module (MC68020).
rtr	return and restore condition codes.
rts	return from subroutine.
sbcd	subtract decimal with extend.

Mnemonic	Description
sCC	set on condition.
sf	set all zeros.
st	set all ones.
stop	halt machine.
subL	subtract.
subqL	subtract quick.
subxL	subtract with extend.
swap	swap register halves.
tas	test operand and set.
trap	trap.
trapCC	conditional trap (MC68020).
trapv	trap on overflow.
tstL	test operand.
unlk	unlink.
unpk	unpack BCD (MC68020).

	S Size of branch
s	short
w	word branch
l	long branch

CC Condition Code	
cs	carry set
eq	equal
ra	inconditional
ge	greater or equal
gt	greater than
hi	high
le	less or equal
ls	lower or same
lt	less than
mi	minus
ne	not equal
pl	positive
vc	no overflow
vs	overflow set

L Length	
b	byte
w	word
l	long

Glossary

Ada	Named after the Countess of Lovelace, the nineteenth century mathematician and computer pioneer, Ada is a high-level general-purpose programming language developed under the sponsorship of the U.S. Department of Defense. Ada was developed to provide consistency among programs originating in different branches of the military. Ada features include packages that make data objects visible only to the modules that need them, task objects that facilitate parallel processing, and an exception handling mechanism that encourages well-structured error processing.
ANSI standard	ANSI is the acronym for the American National Standards Institute. ANSI establishes guidelines in the computing industry, from the definition of ASCII to the determination of overall datacom system performance. ANSI standards have been established for both the Ada and FORTRAN programming languages, and a standard for C has been proposed.
a.out file	**a.out** is the default file name used by the link editor when it outputs a successfully compiled, executable file. **a.out** contains object files that are combined to create a complete working program. Object file format is described in Chapter 11, "The Common Object File Format," and in **a.out**(4).
application program	An application program is a working program in a system. Such programs are usually unique to one type of users' work, although some application programs can be used in a variety of business situations. An accounting application, for example, may well be applicable to many different businesses.
archive	An archive file or archive library is a collection of data gathered from several files. Each of the files within an archive is called a member. The command ar(1) collects data for use as a library.

argument	An argument is additional information that is passed to a command or a function. On a command line, an argument is a character string or number that follows the command name and is separated from it by a space. There are two types of command-line arguments: options and operands. Options are immediately preceded by a minus sign (–) and change the execution or output of the command. Some options can themselves take arguments. Operands are preceded by a space and specify files or directories that will be operated on by the command. For example, in the command **pr –t –h Heading file** all of the elements after the **pr** are arguments. **–t** and **–h** are options, **Heading** is an argument to the **–h** option, and **file** is an operand. For a function, arguments are enclosed within a pair of parentheses immediately following the function name. The number of arguments can be zero or more; if more than two are present they are separated by commas and the whole list enclosed by the parentheses. The formal definition of a function, such as might be found on a manual page in Section 3, describes the number and data type of argument(s) expected by the function.
ASCII	ASCII is an acronym for American Standard Code for Information Interchange, a standard for data representation that is followed in the UNIX system. ASCII code represents alphanumeric characters as binary numbers. The code includes 128 upper- and lower-case letters, numerals, and special characters. Each alphanumeric and special character has an ASCII code (binary) equivalent that is one byte long.
assembler	The assembler is a translating program that accepts instructions written in the assembly language of the computer and translates them into the binary representation of machine instructions. In many cases, the assembly language instructions map 1 to 1 with the binary machine instructions.

assembly language	A programming language that uses the instruction set that applies to a particular computer.
BASIC	BASIC is a high-level conversational programming language that allows a computer to be used much like a complex electronic calculating machine. The name is an acronym for Beginner's All-purpose Symbolic Instruction Code.
branch table	A branch table is an implementation technique for fixing the addresses of text symbols, without forfeiting the ability to update code. Instead of being directly associated with function code, text symbols label jump instructions that transfer control to the real code. Branch table addresses do not change, even when one changes the code of a routine. Jump table is another name for branch table.
buffer	A buffer is a storage space in computer memory where data are stored temporarily into convenient units for system operations. Buffers are often used by programs, such as editors, that access and alter text or data frequently. When you edit a file, a copy of its contents are read into a buffer where you make changes to the text. For the changes to become part of the permanent file, you must write the buffer contents back into the permanent file. This replaces the contents of the file with the contents of the buffer. When you quit the editor, the contents of the buffer are flushed.
byte	A byte is a unit of storage in the computer. On many UNIX systems, a byte is eight bits (binary digits), the equivalent of one character of text.
byte order	Byte order refers to the order in which data are stored in computer memory.
C	The C programming language is a general-purpose programming language that features economy of expression, control flow, data structures, and a variety of operators. It can be used to perform both high-level and low-level tasks. Although it has been called a system programming language, because it is useful for writing operating systems, it has been used equally effectively to write major numerical, text-

	processing, and data base programs. The C programming language was designed for and implemented on the UNIX system; however, the language is not limited to any one operating system or machine.
C compiler	The C compiler converts C programs into assembly language programs that are eventually translated into object files by the assembler.
C preprocessor	The C preprocessor is a component of the C Compilation System. In C source code, statements preceded with a pound sign (#) are directives to the preprocessor. Command line options of the cc(1) command may also be used to control the actions of the preprocessor. The main work of the preprocessor is to perform file inclusions and macro substitution.
CCS	CCS is an acronym for C Compilation System, which is a set of programming language utilities used to produce object code from C source code. The major components of a C Compilation System are a C preprocessor, C compiler, assembler, and link editor. The C preprocessor accepts C source code as input, performs any preprocessing required, then passes the processed code to the C compiler, which produces assembly language code that it passes to the assembler. The assembler in turn produces object code that can be linked to other object files by the link editor. The object files produced are in the Common Object File Format (COFF). Other components of CCS include a symbolic debugger, an optimizer that makes the code produced as efficient as possible, productivity tools, tools used to read and manipulate object files, and libraries that provide runtime support, access to system calls, input/output, string manipulation, mathematical functions, and other code processing functions.
COBOL	COBOL is an acronym for COmmon Business Oriented Language. COBOL is a high-level programming language designed for business and commercial applications. The English-language statements of COBOL provide a relatively machine-independent method of expressing a business-oriented problem to the computer.

COFF	COFF is an acronym for Common Object File Format. COFF refers to the format of the output file produced on some UNIX systems by the assembler and the link editor. This format is also used by other operating systems. The following are some of its key features: ☐ Applications may add system-dependent information to the object file without causing access utilities to become obsolete. ☐ Space is provided for symbolic information used by debuggers and other applications. ☐ Users may make some modifications in the object file construction at compile time.
command	A command is the term commonly used to refer to an instruction that a user types at a computer terminal keyboard. It can be the name of a file that contains an executable program or a shell script that can be processed or executed by the computer on request. A command is composed of a word or string of letters and/or special characters that can continue for several (terminal) lines, up to 256 characters. A command name is sometimes used interchangeably with a program name.
command line	A command line is composed of the command name followed by any argument(s) required by the command or optionally included by the user. The manual page for a command includes a command line synopsis in a notation designed to show the correct way to type in a command, with or without options and arguments.
compiler	A compiler transforms the high-level language instructions in a program (the source code) into object code or assembly language. Assembly language code may then be passed to the assembler for further translation into machine instructions.
core	Core is a (mostly archaic) synonym for primary memory.

core file	A core file is an image of a terminated process saved for debugging. A core file is created under the name "core" in the current directory of the process when an abnormal event occurs resulting in the process' termination. A list of these events is found in the **signal**(2) manual page.
core image	Core image is a copy of all the segments of a running or terminated program. The copy may exist in main storage, in the swap area. or in a core file.
curses	**curses**(3X) is a library of C routines that are designed to handle input, output, and other operations in screen management programs. The name **curses** comes from the cursor optimization that the routines provide. When a screen management program is run, cursor optimization minimizes the amount of time a cursor has to move about a screen to update its contents. The program refers to the **terminfo**(4) data base at run time to obtain the information that it needs about the screen (terminal) being used.
data symbol	A data symbol names a variable that may or may not be initialized. Normally, these variables reside in read/write memory during execution. See text symbol.
data base	A data base is a bank of information on a particular subject or subjects. On-line data bases are designed so that by using subject headings, key words, or key phrases you can search for, analyze, update, and print out data.
debug	Debugging is the process of locating and correcting errors in computer programs.
default	A default is the way a computer will perform a task in the absence of other instructions.
delimiter	A delimiter is an initial character that identifies the next character or character string as a particular kind of argument. Delimiters are typically used for option names on a command line; they identify the associated word as an option (or as a string of several options if the options are bundled). In the UNIX system command syntax, a minus sign (–) is most often

	the delimiter for option names, for example, **–s** or **–n**, although some commands also use a plus sign (+).
directory	A directory is a type of file used to group and organize other files or directories. A directory consists of entries that specify further files (including directories) and constitutes a node of the file system. A subdirectory is a directory that is pointed to by a directory one level above it in the file system organization. The **ls**(1) command is used to list the contents of a directory. When you first log onto the system, you are in your home directory ($HOME). You can move to another directory by using the **cd**(1) command and you can print the name of the current directory by using the **pwd**(1) command. You can also create new directories with the **mkdir**(1) command and remove empty directories with **rmdir**(1). A directory name is a string of characters that identifies a directory. It can be a simple directory name, the relative path name or the full path name of a directory.
dynamic linking	Dynamic linking refers to the ability to resolve symbolic references at run time. Systems that use dynamic linking can execute processes without resolving unused references. See static linking.
environment	An environment is a collection of resources used to support a function. In the UNIX system, the shell environment is composed of variables whose values define the way you interact with the system. For example, your environment includes your shell prompt string, specifics for backspace and erase characters, and commands for sending output from your terminal to the computer. An environment variable is a shell variable such as $HOME (which stands for your login directory) or $PATH (which is a list of directories the shell will search through for executable commands) that is part of your environment. When you log in, the system executes programs that create most of the environmental variables that you need for the commands to

	work. These variables come from /etc/profile, a file that defines a general working environment for all users when they log onto a system. In addition, you can define and set variables in your personal .profile file, which you create in your login directory to tailor your own working environment. You can also temporarily set variables at the shell level.
executable file	An executable file is a file that can be processed or executed by the computer without any further translation. That is, when you type in the file name, the commands in the file are executed. An object file that is ready to run (ready to be copied into the address space of a process to run as the code of that process) is an executable file. Files containing shell commands are also executable. A file may be given execute permission by using the chmod(1) command. In addition to being ready to run, a file in the UNIX system needs to have execute permission.
exit	A specific system call that causes the termination of a process. The **exit**(2) call will close any open files and clean up most other information and memory which was used by the process.
exit status: return code	An exit status or return code is a code number returned to the shell when a command is terminated that indicates the cause of termination.
exported symbol	A symbol that a shared library defines and makes available outside the library. See imported symbol.
expression	An expression is a mathematical or logical symbol or meaningful combination of symbols. See regular expression.
file	A file is an identifiable collection of information that, in the UNIX system, is a member of a file system. A file is known to the UNIX system as an inode plus the information the inode contains that tells whether the file is a plain file, a special file, or a directory. A plain file may contain text, data, programs or other information that forms a coherent unit. A special file is a hardware device or portion thereof, such as a disk partition. A directory is a type of file that contains

	the names and inode addresses of other plain, special or directory files.
file and record locking	The phrase "file and record locking" refers to software that protects records in a data file against the possibility of being changed by two users at the same time. Records (or the entire file) may be locked by one authorized user while changes are made. Other users are thus prevented from working with the same record until the changes are completed.
file descriptor	A file descriptor is a number assigned by the operating system to a file when the file is opened by a process. File descriptors 0, 1, and 2 are reserved; file descriptor 0 is reserved for standard input (**stdin**), 1 is reserved for standard output (**stdout**), and 2 is reserved for standard error output (**stderr**).
file system	A UNIX file system is a hierarchical collection of directories and other files that are organized in a tree structure. The base of the structure is the root (/) directory; other directories, all subordinate to the root, are branches. The collection of files can be mounted on a block special file. Each file of a file system appears exactly once in the inode list of the file system and is accessible via a single, unique path from the root directory of the file system.
filter	A filter is a program that reads information from standard input, acts on it in some way, and sends its results to standard output. It is called a filter because it can be used as a data transformer in a pipeline. Filters are different from editors and other commands because filters do not change the contents of a file. Examples of filters are **grep**(1) and **tail**(1), which select and output part of the input; **sort**(1), which sorts the input; and **wc**(1), which counts the number of words, characters, and lines in the input. **sed**(1) and **awk**(1) are also filters but they are called programmable filters or data transformers because a program must be supplied as input in addition to the data to be transformed.

flag	A flag or option is used on a command line to signal a specific condition to a command or to request particular processing. UNIX system flags are usually indicated by a leading hyphen (–). The word option is sometimes used interchangeably with flag. Flag is also used as a verb to mean to point out or to draw attention to. See option.
fork	**fork**(2) is a system call that divides a new process into two, the parent and child processes, with separate, but initially identical, text, data, and stack segments. After the duplication, the child (created) process is given a return code of 0 and the parent is given the process id of the newly created child as the return code.
FORTRAN	FORTRAN is an acronym for FORmula TRANslator. FORTRAN is a high-level programming language originally designed for scientific and engineering calculations but now also widely adapted for many business uses.
function	A function is a task done by a computer. In most modern programming languages, programs are made up of functions and procedures which perform small parts of the total job to be done.
header file	A header file is used in programming and in document formatting. In a programming context, a header file is a file that usually contains shared data declarations that are to be copied into source programs as they are compiled. A header file includes symbolic names for constants, macro definitions, external variable references and inclusion of other header files. The name of a header file customarily ends with '.h' (dot-h). Similarly, in a document formatting context, header files contain general formatting macros that describe a common document type and can be used with many different document bodies.
high-level language	A high-level language is a computer programming language such as C, FORTRAN, COBOL, or PASCAL that uses symbols and command statements representing actions the computer is to perform, the exact steps for a machine to follow. A high-level

	language must be translated into machine language by a compilation system before a computer can execute it. A characteristic of a high-level language is that each statement usually translates into a series of machine language instructions. The low-level details of the computer's internal organization are left to the compilation system.
host machine	A host machine is the machine on which an a.out file is built.
imported symbol	A symbol used but not defined by a shared library. See exported symbol.
interpreted language	An interpreted language is a high-level language that is not translated by a compilation system and stored in an executable object file. The statements of a program in an interpreted language are translated each time the program is executed.
Interprocess Communication	Interprocess Communication describes software that enables independent processes running at the same time, to exchange information through messages, semaphores, or shared memory.
interrupt	An interrupt is a break in the normal flow of a system or program. Interrupts are initiated by signals that are generated by a hardware condition or a peripheral device indicating that a certain event has happened. When the interrupt is recognized by the hardware, an interrupt handling routine is executed. An interrupt character is a character (normally ASCII) that, when typed on a terminal, causes an interrupt. You can usually interrupt UNIX programs by pressing the delete or break keys, by typing Control-d, or by using the **kill**(1) command.
I/O (Input/Output)	I/O is the process by which information enters (input) and leaves (output) the computer system.
kernel	The kernel (comprising 5 to 10 percent of the operating system software) is the basic resident software on which the UNIX system relies. It is responsible for most operating system functions. It schedules and manages the work done by the computer and maintains the file system. The kernel has its own text,

	data, and stack areas.
lexical analysis	Lexical analysis is the process by which a stream of characters (often comprising a source program) is subdivided into its elementary words and symbols (called tokens). The tokens include the reserved words of the language, its identifiers and constants, and special symbols such as =, :=, and ;. Lexical analysis enables you to recognize, for example, that the stream of characters 'print("hello, universe")' is to be analyzed into a series of tokens beginning with the word 'print' and not with, say, the string 'print("h.' In compilers, a lexical analyzer is often called by the compiler's syntactic analyzer or parser, which determines the statements of the program (that is, the proper arrangements of its tokens).
library	A library is an archive file that contains object code and/or files for programs that perform common tasks. The library provides a common source for object code, thus saving space by providing one copy of the code instead of requiring every program that wants to incorporate the functions in the code to have its own copy. The link editor may select functions and data as needed.
link editor	A link editor, or loader, collects and merges separately compiled object files by linking together object files and the libraries that are referenced into executable load modules. The result is an **a.out** file. Link editing may be done automatically when you use the compilation system to process your programs on the UNIX system, but you can also link edit previously compiled files by using the **ld(1)** command.
magic number	The magic number is contained in the header of an **a.out** file. It indicates what the type of the file is, whether shared or non-shared text, and on which processor the file is executable.
makefile	A makefile is a file that lists dependencies among the source code files of a software product and methods for updating them, usually by recompilation. The **make(1)** command uses the makefile to maintain self-consistent software.

manual page	A manual page, or "man page" in UNIX system jargon, is the repository for the detailed description of a command, a system call, subroutine or other UNIX system component.
null pointer	A null pointer is a C pointer with a value of 0.
object code	Object code is executable machine-language code produced from source code or from other object files by an assembler or a compilation system. An object file is a file of object code and associated data. An object file that is ready to run is an executable file.
optimizer	An optimizer, an optional step in the compilation process, improves the efficiency of the assembly language code. The optimizer reduces the space used by and speeds the execution time of the code.
option	An option is an argument used in a command line to modify program output by modifying the execution of a command. An option is usually one character preceded by a hyphen (–). When you do not specify any options, the command will execute according to its default options. For example, in the command line **ls –a –l directory** **–a** and **–l** are the options that modify the **ls**(1) command to list all **directory** entries, including entries whose names begin with a period (.), in the long format (including permissions, size, and date).
parent process	A parent process occurs when a process is split into two, a parent process and a child process, with separate, but initially identical text, data, and stack segments.
parse	To parse is to analyze a sentence in order identify its components and to determine their grammatical relationship. In computer terminology the word has a similar meaning, but instead of sentences, program statements or commands are analyzed.

PASCAL	PASCAL is a multipurpose high-level programming language often used to teach programming. It is based on the ALGOL programming language and emphasizes structured programming.
path name	A path name is a way of designating the exact location of a file in a file system. It is made up of a series of directory names that proceed down the hierarchical path of the file system. The directory names are separated by a slash character (/). The last name in the path is either a file or another directory. If the path name begins with a slash, it is called a full path name; the initial slash means that the path begins at the **root** directory. A path name that does not begin with a slash is known as a relative path name, meaning relative to the present working directory. A relative path name may begin either with a directory name or with two dots followed by a slash (../). One that begins with a directory name indicates that the ultimate file or directory is below the present working directory in the hierarchy. One that begins with ../ indicates that the path first proceeds up the hierarchy; ../ is the parent of the present working directory.
permissions	Permissions are a means of defining a right to access a file or directory in the UNIX file system. Permissions are granted separately to you, the owner of the file or directory, your group, and all others. There are three basic permissions: ☐ Read permission (r) includes permission to cat, pg, lp, and cp a file. ☐ Write permission (w) is the permission to change a file. ☐ Execute permission (x) is the permission to run an executable file. Permissions can be changed with the UNIX system **chmod(1)** command.

pipe	A pipe causes the output of one command to be used as the input for the next command so that the two run in sequence. You can do this by preceding each command after the first command with the pipe symbol (|), which indicates that the output from the process on the left should be routed to the process on the right. For example, in the command **who | wc -l**, the output from the **who(1)** command, which lists the users who are logged on to the system, is used as input for the word-count command, wc(1), with the **l** option. The result of this pipeline (succession of commands connected by pipes) is the number of people who are currently logged on to the system.
portable	Portability describes the degree of ease with which a program or a library can be moved or ported from one system to another. Portability is desirable because once a program is developed it is used on many systems. If the program writer must change the program in many different ways before it can be distributed to the other systems, time is wasted, and each modification increases the chances for an error.
preprocessor	Preprocessor is a generic name for a program that prepares an input file for another program. For example, **neqn(1)** and **tbl(1)** are preprocessors for **nroff(1)**. **grap(1)** is a preprocessor for **pic(1)**. **cpp(1)** is a preprocessor for the C compiler.
process	A process is a program that is at some stage of execution. In the UNIX system, it also refers to the execution of a computer environment, including contents of memory, register values, name of the current working directory, status of files, information recorded at login time, etc. Every time you type the name of a file that contains an executable program, you initiate a new process. Shell programs can cause the initiation of many processes because they can contain many command lines. The process id is a unique system-wide identification

	number that identifies an active process. The process status command, **ps(1)**, prints the process ids of the processes that belong to you.
program	A program is a sequence of instructions or commands that cause the computer to perform a specific task, for example, changing text, making a calculation, or reporting on the status of the system. A subprogram is part of a larger program and can be compiled independently.
regular expression	A regular expression is a string of alphanumeric characters and special characters that describe a character string. It is a shorthand way of describing a pattern to be searched for in a file. The pattern-matching functions of **ed(1)** and **grep(1)**, for example, use regular expressions.
routine	A routine is a discrete section of a program to accomplish a set of related tasks
semaphore	In the UNIX system, a semaphore is a sharable short unsigned integer maintained through a family of system calls which include calls for increasing the value of the semaphore, setting its value, and for blocking waiting for its value to reach some value. Semaphores are part of the UNIX system IPC facility.
shared library	Shared libraries include object modules that may be shared among several processes at execution time.
shared memory	Shared memory is an IPC (interprocess communication) facility in which two or more processes can share the same data space.
shell	The shell is the UNIX system program—sh(1)—responsible for handling all interaction between you and the system. It is a command language interpreter that understands your commands and causes the computer to act on them. The shell also establishes the environment at your terminal. A shell normally is started for you as part of the login process. Three shells, the Bourne shell, the Korn shell and the C shell, are popular. The shell can also be used as a programming language to write procedures for a variety of tasks.

signal: signal number	A signal is a message that you send to processes or processes send to one another. The most common signals you might send to a process are ones that would cause the process to stop: for example, interrupt, quit, or kill. A signal sent by a running process is usually a sign of an an exceptional occurrence that has caused the process to terminate or divert from the normal flow of control.
source code	Source code is the programming-language version of a program. Before the computer can execute the program, the source code must be translated to machine language by a compilation system or an interpreter.
standard error	Standard error is an output stream from a program. It is normally used to convey error messages. In the UNIX system, the default case is to associate standard error with the user's terminal.
standard input	Standard input is an input stream to a program. In the UNIX system, the default case is to associate standard input with the user's terminal.
standard output	Standard output is an output stream from a program. In the UNIX system, the default case is to associate standard output with the user's terminal.
stdio: standard input-output	**stdio**(3S) is a collection of functions for formatted and character-by-character input-output at a higher level than the basic read, write, and open operations.
static linking	Static linking refers to the requirement that symbolic references be resolved before run time. See dynamic linking.
stream	☐ A stream is an open file with buffering provided by the stdio package. ☐ A stream is a full duplex, processing and data transfer path in the kernel. It implements a connection between a driver in kernel space and a process in user space, providing a general character input/output interface for the user

processes.

string	A string is a contiguous sequence of characters treated as a unit. Strings are normally bounded by white space(s), tab(s), or a character designated as a separator. A string value is a specified group of characters symbolized to the shell by a variable.
strip	**strip**(1) is a command that removes the symbol table and relocation bits from an executable file.
subroutine	A subroutine is a program that defines desired operations and may be used in another program to produce the desired operations. A subroutine can be arranged so that control may be transferred to it from a master routine and so that, at the conclusion of the subroutine, control reverts to the master routine. Such a subroutine is usually called a closed subroutine. A single routine may be simultaneously a subroutine with respect to another routine and a master routine with respect to a third.
symbol table	A symbol table describes information in an object file about the names and functions in that file. The symbol table and relocation bits are used by the link editor and by the debuggers.
symbol value	The value of a symbol, typically its virtual address, used to resolve references.
syntax	☐ Command syntax is the order in which command names, options, option arguments, and operands are put together to form a command on the command line. The command name is first, followed by options and operands. The order of the options and the operands varies from command to command. ☐ Language syntax is the set of rules that describe how the elements of a programming language may legally be used.
system call	A system call is a request by an active process for a service performed by the UNIX system kernel, such as I/O, process creation, etc. All system operations are

allocated, initiated, monitored, manipulated, and terminated through system calls. System calls allow you to request the operating system to do some work that the program would not normally be able to do. For example, the **getuid**(2) system call allows you to inspect information that is not normally available since it resides in the operating system's address space.

target machine — A target machine is the machine on which an **a.out** file is run. While it may be the same machine on which the **a.out** file was produced, the term implies that it may be a different machine.

TCP/IP (Transmission Control Protocol/Internetwork Protocol) — TCP/IP is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols that support multi-network applications. It is the Department of Defense standard in packet networks.

terminal definition — A terminal definition is an entry in the **terminfo**(4) data base that describes the characteristics of a terminal. See **terminfo**(4) and **curses**(3X).

terminfo

- ☐ a group of routines within the curses library that handle certain terminal capabilities. For example, if your terminal has programmable function keys, you can use these routines to program the keys.
- ☐ a data base containing the compiled descriptions of many terminals that can be used with **curses**(3X) screen management programs. These descriptions specify the capabilities of a terminal and how it performs various operations — for example, how many lines and columns it has and how its control characters are interpreted. A **curses**(3X) program refers to the data base at run time to obtain the information that it needs about the terminal being used.

	See **curses**(3X). **terminfo**(4) routines can be used in shell programs, as well as C programs.
text symbol	A text symbol is a symbol, usually a function name, that is defined in the **.text** portion of an **a.out** file.
tool	A tool is a program, or package of programs, that performs a given task.
trap	A trap is a condition caused by an error where a process state transition occurs and a signal is sent to the currently running process.

UNIX operating system

The UNIX operating system is a general-purpose, multiuser, interactive, time-sharing operating system. An operating system is the software on the computer under which all other software runs. The UNIX operating system has two basic parts:

- ☐ The kernel is the program that is responsible for most operating system functions. It schedules and manages all the work done by the computer and maintains the file system. It is always running and is invisible to users.
- ☐ The shell is the program responsible for handling all interaction between users and the computer. It includes a powerful command language called shell language.

The utility programs or UNIX system commands are executed using the shell, and allow users to communicate with each other, edit and manipulate files, and write and execute programs in several programming languages.

userid	A userid is an integer value, usually associated with a login name, used by the system to identify owners of files and directories. The userid of a process becomes the owner of files created by the process and descendent (forked) processes.
utility	A utility is a standard, permanently available program used to perform routine functions or to assist a programmer in the diagnosis of hardware and software

	errors, for example, a loader, editor, debugging, or diagnostics package.
variable	☐ A variable in a computer program is an object whose value may change during the execution of the program, or from one execution to the next. ☐ A variable in the shell is a name representing a string of characters (a string value). ☐ A variable normally set only on a command line is called a parameter (positional parameter and keyword parameter). ☐ A variable may be simply a name to which the user (user-defined variable) or the shell itself may assign string values.
white space	White space is one or more spaces, tabs, or newline characters. White space is normally used to separate strings of characters, and is required to separate the command from its arguments on a command line.
window	A window is a screen within your terminal screen that is set off from the rest of the screen. If you have two windows on your screen, they are independent of each other and the rest of the screen. The most common way to create windows on a UNIX system is by using the layers capability of the TELETYPE 5620 Dot-Mapped Display. Each window you create with this program has a separate shell running it. Each one of these shells is called a layer. If you do not have this facility, the shl(1) command, which stands for shell layer, offers a function similar to the layers program. You cannot create windows using shl(1), but you can start different shells that are independent of each other. Each of the shells you create with shl(1) is called a layer.
word	A word is a unit of storage in a computer that is composed of bytes of information. The number of bytes in a word depends on the computer you are using.

The CADMUS 32bit computers, for example, have 32 bits or 4 bytes per word, and 16 bits or 2 bytes per half word.

Index

Contents	Chapter	Page
scanw()	10	35
scatter	10	88
Scope Rules	16	44
Scope of Externals	16	45
sdb	2	65
	14	2
Section Definition Directives	12	8
Section Header Declaration	11	10
Section Headers	11	8
Section Number Field	11	27
Sections	11	3 12 35
	12	2
Selecting Library Contents	8	17 50
Selectively Loading Imported Symbols	8	34
Semaphores	9	38
semctl	9	53
semget	9	44
semop	9	66
Sending a Message	9	25
Separate Source Files	8	25
Shared Library	3	29
	8	55
Shared Memory	9	74
Shared Memory Segments	9	79
Shell Facility	4	49
Shell as a Prototyping Tool	1	5
Shift Operators	16	18
shmat	9	100
shmctl	9	87
shmdt	9	101
shmget	9	79
shmop	9	98
show	10	91
Signals and Interrupts	2	42
Single-User Programmer	1	7
size	2	65
Soft Labels	10	58
Source File Display and Manipulation	14	6
Source Program Format	17	3