68NW9209H08C

SYSTEM V/68 Release 3 Programmer's Guide Volume 2

MOTOROLA

MOTOROLA IN

R

Motorola welcomes your commen

Manual Title ____________

Part Number ____________

Your Name ____________

Your Title ____________

Company ____________

Address ____________

____

____

General Information:

Do you read this manual in orc

☐ Install the product ☐ Us

☐ Reference information

In general, how do you locate

☐ Index ☐ Table of Conte

Completeness: ☐ Excellent

What topic would you like mo

____

____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 2565 PHOENIX, ARIZONA

POSTAGE WILL BE PAID BY ADDRESSEE

MOTOROLA INC.
Computer Group, Microcomputer Division, DW164
2900 South Diablo Way
Tempe, AZ 85282-9741

Presentation: ☐ Excellent ☐ Very Good ☐ Good ☐ Fair ☐ Poor

What features of the manual are most useful (tables, figures, appendixes, index, etc.)?

Is the information easy to understand? ☐ Yes ☐ No If you checked no, please explain:

Is the information easy to find? ☐ Yes ☐ No If you checked no, please explain:

Technical Accuracy: ☐ Excellent ☐ Very Good ☐ Good ☐ Fair ☐ Poor

If you have found technical or typographical errors, please list them here.

Page Number	Description of Error

SYSTEM V/68 RELEASE 3

PROGRAMMER'S GUIDE

Part Number 68NW9209H08C

Volume 2

January 1989

PREFACE

The *Programmer's Guide* (Part Number 68NW9209H08C) provides information about programming in a SYSTEM V/68 environment. This basic document covers Motorola's Release 3 Basic Operating System through Version 5. Information beyond this version will be provided in supplemental documents.

The following changes have been made to this version of the document:

- Chapter 8 (Shared Libraries) of the /D1 and /D2 versions has been removed. The software does not support shared libraries.
- The original document has been divided into two volumes.
- The original Chapters 1 through 10, 19 and 20 (renumbered 1 through 11), Appendices A, B, and C, and the Glossary are contained in Volume 1.
- The original Chapters 11 through 18 (renumbered 12 through 19), and the Index are in Volume 2.

CONTENTS

FIGURES

CHAPTER 12
COMMON OBJECT FILE FORMAT (COFF)

The Common Object File Format (COFF)

This section describes the Common Object File Format (COFF), 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 12-1 shows the overall structure.

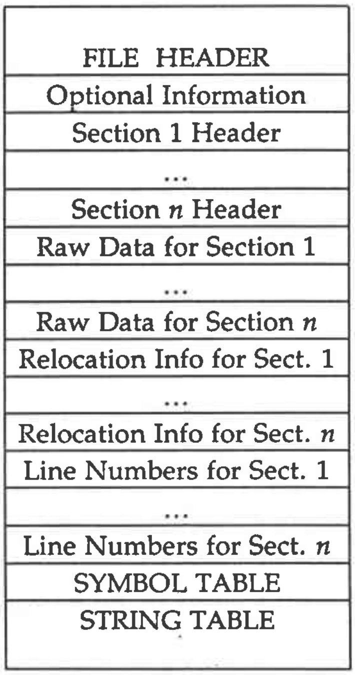

Figure 12-1. Object File Format

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 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 specific 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, 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. Usually, the target machine is the same computer on which the object file is being created.

File Header

The file header contains the 20 bytes of information shown in Figure 12-2. The last two 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 12-3)

Figure 12-2. File Header Contents

Magic Numbers

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

Flags

The last two 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 12-3.

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_MINMAL	00020	Not used by SYSTEM V/68
F_UPDATE	00040	Not used by SYSTEM V/68
F_SWABD	00100	Not used by SYSTEM V/68
F_AR16WR	00200	File has the byte ordering used by the PDP-11/70 processor
F_AR32WR	00400	File has the byte ordering used by the VAX-11/780 (i.e., 32 bits per word, least significant byte first)
F_AR32W	01000	File has the byte ordering used by the M68K computers (i.e., 32 bits per word, most significant byte first)
F_PATCH	02000	Not used by SYSTEM V/68

Figure 12-3. File Header Flags

File Header Declaration

The C structure declaration for the file header is given in Figure 12-4. 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 12-4. 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 Operating System a.out Header

By default, files produced by the link editor for the operating system always have a standard operating system **a.out** header in the optional header field. The operating system **a.out** header is 28 bytes. The fields of the optional header are described in Figure 12-5.

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 12-5. Optional Header Contents

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 operating system are given in Figure 12-6.

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 12-6. Operating System Magic Numbers

12

Optional Header Declaration

The C language structure declaration currently used for the operating system **a.out** file header is given in Figure 12-7. 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 12-7. 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 12-8.

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 12-9)

Figure 12-8. 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 operating system function **fseek**(3S).

Flags

The lower 2 bytes of the flag field indicate a section type. The flags are described in Figure 12-9.

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 12-9. Section Header Flags

Section Header Declaration

The C structure declaration for the section headers is described in Figure 12-10. 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 12-10. 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 12-1 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 four-byte boundary in the file.

Link editor SECTIONS directives (see Chapter 13) 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 several 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 12-11.

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 12-11. Relocation Section Contents

The first four 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 12-12.

Mnemonic	Flag	Meaning
R_ABS	0	Reference is absolute; no relocation is necessary. The entry will be ignored.
R_RELBYTE	017	Direct 8-bit reference to the symbol's virtual address.
R_RELWORD	020	Direct 16-bit reference to the symbol's virtual address.
R_RELLONG	021	Direct 32-bit reference to the symbol's virtual address.
R_PCRBYTE	022	A "PC-relative" 8-bit reference to the symbol's virtual address.
R_PCRWORD	023	A "PC-relative" 16-bit reference to the symbol's virtual address.
R_PCRLONG	024	A "PC-relative" 32-bit reference to the symbol's virtual address.

Figure 12-12. Relocation Types

Relocation Entry Declaration

The structure declaration for relocation entries is given in Figure 12-13. 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 12-13. 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 12-14.

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

Figure 12-14. 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 12-15.

```
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 12-15. 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 12-16.

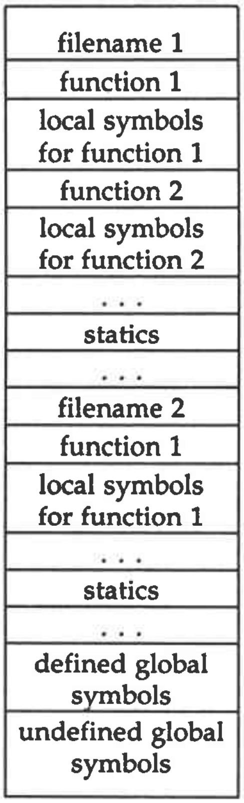

Figure 12-16. COFF Symbol Table

The word statics in Figure 12-16 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**. It contains other tools as well. These symbols are given in Figure 12-17.

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
.*x*fake	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 12-17. 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 **.*x*fake** 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 12-18.

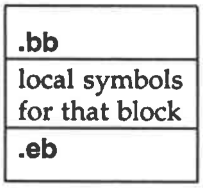

Figure 12-18. Special Symbols (**.bb** and **.eb**)

Because inner blocks can be nested by several levels, the **.bb-.eb** pairs and associated symbols may also be nested. See Figure 12-19.

```
  {                           /* 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 12-19. Nested blocks

The symbol table would look like Figure 12-20.

.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 12-20. 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 12-21.

function name
.bf
local symbol
.ef

Figure 12-21. 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 12-22. Note 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 12-22. Symbol Table Entry Format

Symbol Names

The first eight 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 eight 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 four bytes serve to distinguish a symbol table entry with an offset from one with a name in the first eight bytes as shown in Figure 12-23.

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 12-23. Name Field

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

Storage Classes

The storage class field has one of the values described in Figure 12-24. 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
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 12-24. Storage Classes

All these storage classes except for C_ALIAS and C_HIDDEN are generated by the **cc** or **as** commands. The compress utility, **cprs**, generates the C_ALIAS mnemonic. This utility (described in the *User's Reference Manual*) removes duplicated structure, union, and enumeration definitions and puts alias entries in their places. The storage class C_HIDDEN is not used by any operating 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 12-25.

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

Figure 12-25. Storage Class by Special Symbols

Also some storage classes are used only for certain special symbols. They are summarized in Figure 12-26.

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

Figure 12-26. 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 12-27.

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 12-27. 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 12-28.

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 12-28. 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 12-29.

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 12-29. Section Number and Storage Class

Type Entry

The type field in the symbol table entry contains information 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 12-30.

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 12-30. 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 12-31.

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

Figure 12-31. 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 12-32 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
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 12-32. Type Entries by Storage Class

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 12-33. 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 12-33. 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 12-34.

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_STRUCT, T_UNION, T_ENUM	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_NON	T_STRUCT, T_UNION, T_ENUM	name related to structure, union, enumeration

Notes to Figure 12-34:
1. Any except T_MOE.
2. C_AUTO, C_STAT, C_MOS, C_MOU, C_TPDEF.

Figure 12-34. Auxiliary Symbol Table Entries

In Figure 12-34, *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 12-34 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 12-35.

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 12-35. Format for Auxiliary Table Entries for Sections

Tag Names

The auxiliary table entries for tag names have the format shown in Figure 12-36.

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 12-36. Tag Names Table Entries

End of Structures

The auxiliary table entries for the end of structures have the format shown in Figure 12-37.

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 12-37. Table Entries for End of Structures

Functions

The auxiliary table entries for functions have the format shown in Figure 12-38.

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 by the operating system)

Figure 12-38. Table Entries for Functions

Arrays

The auxiliary table entries for arrays have the format shown in Figure 12-39. 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 12-39. 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 12-40.

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 12-40. 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 12-41.

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 12-41. 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 12-42.

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 12-42. 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 12-43. 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
 .
 .
 .
 .
 .
 .
                     {
                            long     x_lnnoptr;
                            long     x_endndx;
                     } x_fcn;
                     struct
                     {
                            unsigned short   x_dimen[DIMNUM];
                     } x_ary;
                } x_fcnary;
                unsigned short   x_tvndx;
        } x_sym;
```

Figure 12-43. Auxiliary Symbol Table Entry (Sheet 1 of 2)

```
        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 12-43. 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 12-44.

'l'	'o'	'n'	'g'
'_'	'n'	'a'	'm'
'e'	'_'	'l'	'\0'
'a'	'n'	'o'	't'
'h'	'e'	'r'	'_'
'o'	'n'	'e'	'\0'

Figure 12-44. String Table

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

Access Routines

Operating 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 the *Programmer's Reference Manual*. A summary of what is available can be found in the *Programmer's Reference Manual* under **ldfcn**(4).

CHAPTER 13
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 their location in memory. When you do need to be precise in controlling the link editor output, you do it via 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 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 contains 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 13-2, "Syntax Diagram for Input Directives.") Constants are, as in C, recognized as decimal numbers unless preceded with 0 for octal or 0x for hexadecimal. All numbers are treated as long integers. 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 13-1.

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

Figure 13-1. 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 (except 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 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 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 on the target machine.

Using 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.
}
```

13

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 way 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
        {
                outsec1:
                {
                        file1.o (sec1)
                        file2.o
                        file3.o (sec1, sec2)
                }
        }
```

According to this directive, the order in which the input sections appear in the output section **outsec1** would be

1. section **sec1** from file **file1.o**
2. all sections from **file2.o**, in the order they appear in the file
3. section **sec1** 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 **outsec1**, these sections are linked following the last section named in the definition of **outsec1**. If there are any other input sections in **file1.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

An output section is bonded to a specific virtual address 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) does the following:

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 0x2000 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 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 this way 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) : {
```

Note 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 operating 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:

a. Explicit initialization options must be used to initialize all **.bss** sections in the output section.

b. **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 **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):

a. The value of the symbol specified with the **–e** option, if present, is used.

b. The value of the symbol **_start**, if present, is used.

c. The value of the symbol **main**, if present, is used.

d. 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

```
        _start   =   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:

a. There exists a reference to a symbol defined in that member.

b. The reference is found by **ld**(1) before the 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 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) 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 external reference to the members. For example, the command:

```
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.
}
```

13

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 comprise zero or more input sections. 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

Figure 13-2 summarizes the system requirements for input directives. Note that two punctuation symbols, square brackets and curly braces, do double duty in this diagram.

Where the 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
<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 13-2. Syntax Diagram for Input Directives (Sheet 1 of 4)

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 13-2. 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 13-2. 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 operating system
<sectionname>	Any valid section name,
<path_name>	Any valid operating system

Figure 13-2. Syntax Diagram for Input Directives (Sheet 4 of 4)

CHAPTER 14
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 such things as:

- file-to-file dependencies
- files that were modified and the impact that 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 enough to regenerate the target regardless of the number files edited since the last **make**. Usually, the description file is not difficult to write and changes infrequently. Even if only a single file has been edited, typing the **make** command rather than typing all the commands to regenerate the target 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 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 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, a "save" entry might be included to copy a certain set of files, or a "clean" entry 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.

The **$***, **$@**, **$?**, and **$<** 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 operating 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 14-2 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 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 14-1 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.

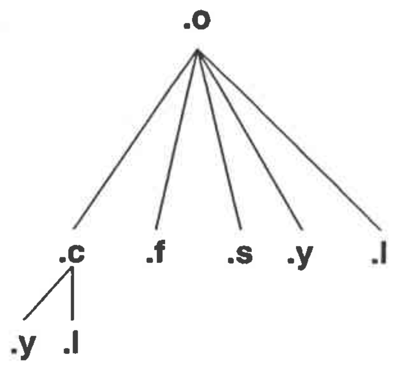

Figure 14-1. Summary of Default Transformation Path

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 14-1.

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
```

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, the command line:

```
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 as follows:

```
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 **.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 **make**ing **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. **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
```

Note 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.

Source Code Control System Filenames: the Tilde

The syntax of **make** does not directly permit referencing of prefixes. For most types of files, this is acceptable since nearly everyone uses a suffix to distinguish different types of files. The SCCS files are the exception. Here, **s.** precedes the filename part of the complete path name.

To allow **make** easy access to the prefix **s.** the tilde, ~, is used as an identifier of SCCS files. Hence, **.c~.o** refers to the rule which transforms an SCCS C language source file into an object file. Specifically, the internal rule is:

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

Thus, the tilde appended to any suffix transforms the file search into an SCCS filename search with the suffix named by the dot and all characters up to (but not including) the tilde.

The following SCCS suffixes are internally defined:

.c~
.f~
.y~
.l~
.s~
.sh~
.h~

The following rules involving SCCS transformations are internally defined:

```
.c~:
.f~:
.sh~:
.c~.a:
.c~.c:
.c~.o:
.f~.a:
.f~.f:
.f~.o:
.s~.a:
.s~.s:
.s~.o:
.y~.c:
.y~.o:
.l~.l:
.l~.o:
.h~.h:
```

Obviously, the user can define other rules and suffixes, which may prove useful. The tilde provides a handle on the SCCS filename format so that this is possible.

The Null Suffix

There are many programs that consist of a single source file. **make** handles this case by the null suffix rule. Thus, to maintain the operating system program **cat,** a rule in the **makefile** of the following form is needed:

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

In fact, this **.c:** rule is internally defined so no **makefile** is necessary at all. The user only needs to type:

```
make cat dd echo date
```

(these are all operating system single-file programs) and all four C language source files are passed through the above shell command line associated with the **.c:** rule.

The internally defined single suffix rules are:

.c:
.c~:
.f:
.f~:
.sh:
.sh~:

Others may be added in the **makefile** by the user.

include Files

The **make** program has a capability similar to the **#include** directive of the C preprocessor. If the string **include** appears as the first seven letters of a line in a **makefile** and is followed by a blank or a tab, the rest of the line is assumed to be a filename, which the current invocation of **make** will read. Macros may be used in filenames. The file descriptors are stacked for reading **include** files so that no more than 16 levels of nested **includes** are supported.

SCCS Makefiles

Makefiles under SCCS control are accessible to **make**. That is, if **make** is typed and only a file named **s.makefile** or **s.Makefile** exists, **make** will do a **get** on the file, then read and remove the file.

Dynamic Dependency Parameters

The parameter has meaning only on the dependency line in a makefile. The **$$@** refers to the current "thing" to the left of the colon (which is **$@**). Also the form **$$(@F)** exists, which allows access to the file part of **$@**. Thus, in the following:

```
cat:     $$@.c
```

the dependency is translated at execution time to the string **cat.c**. This is useful for building a large number of executable files, each of which has only one source file.

For instance, the operating system command directory could have a **makefile** like:

```
CMDS = cat dd echo date cmp comm chown

$(CMDS):        $$@.c
        $(CC) -O $? -o $@
```

Obviously, this is a subset of all the single file programs. For multiple file programs, a directory is usually allocated and a separate **makefile** is made. For any particular file that has a peculiar compilation procedure, a specific entry must be made in the **makefile**.

The second useful form of the dependency parameter is **$$(@F)**. It represents the filename part of **$$@**. Again, it is evaluated at execution time. Its usefulness becomes evident when trying to maintain the **/usr/include** directory from a makefile in the **/usr/src/head** directory. Thus, the **/usr/src/head/makefile** would look like:

```
INCDIR = /usr/include

INCLUDES = \
        $(INCDIR)/stdio.h \
        $(INCDIR)/pwd.h \
        $(INCDIR)/dir.h \
        $(INCDIR)/a.out.h

$(INCLUDES): $$(@F)
        cp $? $@
        chmod 0444 $@
```

This would completely maintain the **/usr/include** directory whenever one of the above files in **/usr/src/head** was updated.

Command Usage

The **make** command description is found under **make**(1) in the *Programmer's Reference Manual*.

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 **makefile**s.

–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 way 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**.

14

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 **makefile**s 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 line that specifies:

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

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 14-2. 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 14-2. 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 14-2. 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 14-2. 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
```

Figure 14-2. make Internal Rules (Sheet 5 of 5)

CHAPTER 15
SOURCE CODE CONTROL SYSTEM (SCCS)

Introduction

The Source Code Control System (SCCS) is a maintenance and enhancement tracking tool that runs under the operating system. SCCS takes custody of a file and, when changes are made, identifies and stores them in the file with the original source code and/or documentation. As other changes are made, they too are identified and retained in the file.

Retrieval of the original or any set of changes is possible. Any version of the file as it develops can be reconstructed for inspection or additional modification. History data can be stored with each version: why the changes were made, who made them, when they were made.

This guide covers the following:

- SCCS for Beginners: how to make, retrieve, and update an SCCS file
- Delta Numbering: how versions of an SCCS file are named
- SCCS Command Conventions: what rules apply to SCCS commands
- SCCS Commands: the fourteen SCCS commands and their more useful arguments
- SCCS Files: protection, format, and auditing of SCCS files

Neither the implementation of SCCS nor the installation procedure for SCCS is described in this guide.

SCCS For Beginners

Several terminal session fragments are presented in this section. Try them all. The best way to learn SCCS is to use it.

Terminology

A delta is a set of changes made to a file under SCCS custody. To identify and keep track of a delta, it is assigned an SID (SCCS IDentification) number. The SID for any original file turned over to SCCS is composed of release number 1 and level number 1, stated as 1.1. The SID for the first set of changes made to that file, that is, its first delta is release 1 version 2, or 1.2. The next delta would be 1.3, the next 1.4, and so on. More on delta numbering later. At this point, it is enough to know that by default SCCS assigns SIDs automatically.

Creating an SCCS File via admin

Suppose, for example, you have a file called **lang** that is simply a list of five programming language names. Use a text editor to create file **lang** containing the following list.

```
C
PL/1
FORTRAN
COBOL
ALGOL
```

Custody of your **lang** file can be given to SCCS using the **admin** command (i.e., administer SCCS file). The following creates an SCCS file from the **lang** file:

```
admin –ilang s.lang
```

All SCCS files must have names that begin with **s.**, hence **s.lang**. The **–i** keyletter, together with its value **lang**, means **admin** is to create an SCCS file and initialize it with the contents of the file **lang**.

The **admin** command replies

```
No id keywords (cm7)
```

This is a warning message that may also be issued by other SCCS commands. Ignore it for now. Its significance is described later with the **get** command under "SCCS Commands." In the following examples, this warning message is not shown although it may be issued.

The **lang** file is no longer needed because it exists now under SCCS as **s.lang**. Remove the **lang** file:

```
rm lang
```

Retrieving a File via get

Use the **get** command as follows:

```
get s.lang
```

This retrieves **s.lang** and prints:

```
1.1
5 lines
```

This tells you that **get** retrieved version 1.1 of the file, which is made up of five lines of text.

The retrieved text has been placed in a new file known as a "g.file." SCCS forms the g.file name by deleting the prefix **s.** from the name of the SCCS file. Thus, the original **lang** file has been recreated.

If you list, **ls**(1), the contents of your directory, you will see both **lang** and **s.lang**. SCCS retains **s.lang** for use by other users.

The **get s.lang** command creates **lang** as read-only and keeps no information regarding its creation. Because you are going to make changes to it, **get** must be informed of your intention to do so. This is done as follows:

```
get –e s.lang
```

get –e causes SCCS to create **lang** for both reading and writing (editing). It also places certain information about **lang** in another new file, called the "p.file" (**p.lang** here), which is needed later by the **delta** command.

get –e prints the same messages as **get**, except that now the SID for the first delta you will create is issued:

```
1.1
new delta  1.2
5 lines
```

Change **lang** by adding two more programming languages:

```
SNOBOL
ADA
```

Recording Changes via delta

Next, use the **delta** command as follows:

```
delta s.lang
```

delta then prompts with:

```
comments?
```

Your response should be an explanation of why the changes were made. For example:

```
added more languages
```

delta now reads the p.file, **p.lang**, and determines what changes you made to **lang**. It does this by doing its own **get** to retrieve the original version and applying the **diff**(1) command to the original version and the edited version. Next, **delta** stores the changes in **s.lang** and destroys the no longer needed **p.lang** and **lang** files.

When this process is complete, **delta** outputs:

```
1.2
2 inserted
0 deleted
5 unchanged
```

The number 1.2 is the SID of the delta you just created, and the next three lines summarize what was done to **s.lang**.

Additional Information about get

The command:

```
get  s.lang
```

retrieves the latest version of the file **s.lang**, now 1.2. SCCS does this by starting with the original version of the file and applying the delta you made. If you use the **get** command now, you can retrieve version 1.2. with any of the following:

```
get  s.lang
get  –r1  s.lang
get  –r1.2  s.lang
```

The numbers following **–r** are SIDs. When you omit the level number of the SID (as in **get –r1 s.lang**), the default is the highest level number that exists within the specified release. Thus, the second command requests the retrieval of the latest version in release 1, namely 1.2. The third command specifically requests the retrieval of a particular version, in this case also 1.2.

Whenever a major change is made to a file, you may want to signify it by changing the release number, the first number of the SID. This, too, is done with the **get** command:

```
get  –e  –r2  s.lang
```

Because release 2 does not exist, **get** retrieves the latest version before release 2. **get** also interprets this as a request to change the release number of the new delta to 2, thereby naming it 2.1 rather than 1.3. The output is:

```
1.2
new delta 2.1
7 lines
```

which means version 1.2 has been retrieved, and 2.1 is the version **delta** will create. If the file is now edited (for example, by deleting COBOL from the list of languages), and **delta** is executed:

```
delta s.lang
 comments? deleted cobol from list of languages
```

you will see by **delta**'s output that version 2.1 is indeed created:

```
2.1
0 inserted
1 deleted
6 unchanged
```

Deltas can now be created in release 2 (deltas 2.2, 2.3, etc.), or another new release can be created in a similar manner.

The help Command

If the command:

```
get lang
```

is now executed, the following message will be output:

```
ERROR [lang]: not an SCCS file (co1)
```

The code **co1** can be used with **help** to print a fuller explanation of the message:

```
help co1
```

This gives the following explanation of why **get lang** produced an error message:

```
co1:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s.".
```

help is useful whenever there is doubt about the meaning of almost any SCCS message.

Delta Numbering

Think of deltas as the nodes of a tree in which the root node is the original version of the file. The root is normally named 1.1 and deltas (nodes) are named 1.2, 1.3, etc. The components of these SIDs are called release and level numbers, respectively. Thus, normal naming of new deltas proceeds by incrementing the level number. This is done automatically by SCCS whenever a delta is made.

Because the user may change the release number to indicate a major change, the release number then applies to all new deltas unless specifically changed again. Thus, the evolution of a particular file could be represented by Figure 15-1.

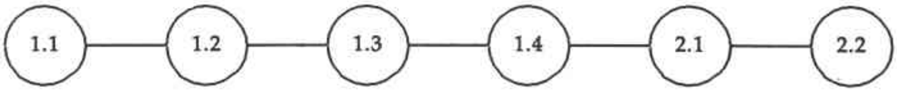

Figure 15-1. Evolution of an SCCS File

This is the normal sequential development of an SCCS file, with each delta dependent on the preceding deltas. Such a structure is called the trunk of an SCCS tree.

There are situations that require branching an SCCS tree. That is, changes are planned to a given delta that will not be dependent on all previous deltas. For example, consider a program in production use at version 1.3 and for which development work on release 2 is already in progress. Release 2 may already have a delta in progress as shown in Figure 15-1. Assume that a production user reports a problem in version 1.3 that cannot wait to be repaired in release 2. The changes necessary to repair the trouble will be applied as a delta to version 1.3 (the version in production use). This creates a new version that will then be released to the user but will not affect the changes being applied for release 2 (i.e., deltas 1.4, 2.1, 2.2, etc.). This new delta is the first node of a new branch of the tree.

Branch delta names always have four SID components: the same release number and level number as the trunk delta, plus a branch number and sequence number. The format is as follows:

release.level.branch.sequence

The branch number of the first delta branching off any trunk delta is always 1, and its sequence number is also 1. For example, the full SID for a delta branching off trunk delta 1.3 will be 1.3.1.1. As other deltas on that same branch are created, only the sequence number changes: 1.3.1.2, 1.3.1.3, etc. This is shown in Figure 15-2.

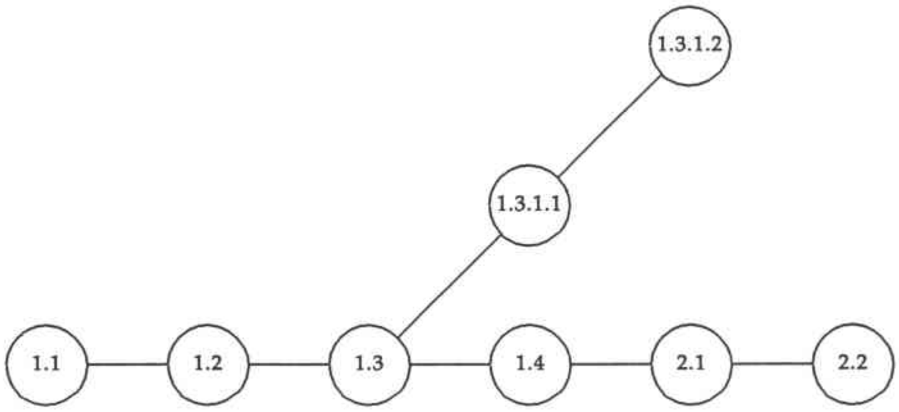

Figure 15-2. Tree Structure with Branch Deltas

The branch number is incremented only when a delta is created that starts a new branch off an existing branch, as shown in Figure 15-3. As this secondary branch develops, the sequence numbers of its deltas are incremented (1.3.2.1, 1.3.2.2, etc.), but the secondary branch number remains the same.

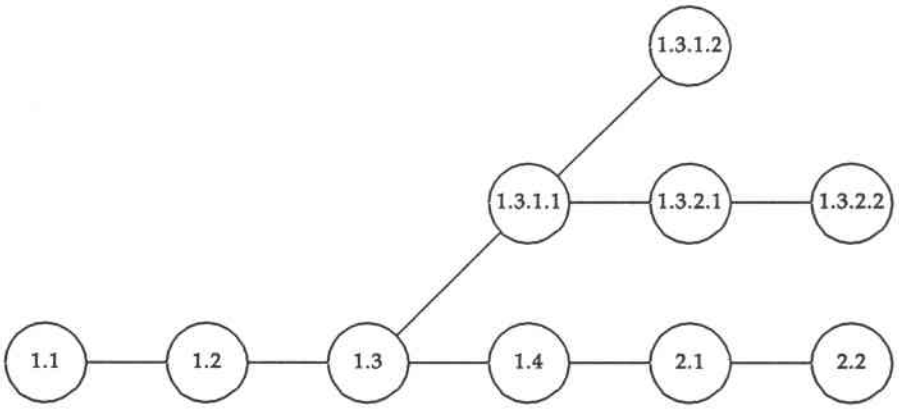

Figure 15-3. Extended Branching Concept

The concept of branching may be extended to any delta in the tree, and the numbering of the resulting deltas proceeds as shown above. SCCS allows the generation of complex tree structures. Although this capability has been provided for certain specialized uses, the SCCS tree should be kept as simple as possible. Comprehension of its structure becomes difficult as the tree becomes complex.

SCCS Command Conventions

SCCS commands accept two types of arguments:

- keyletters
- filenames

Keyletters are options that begin with a minus sign, –, followed by a lowercase letter and, in some cases, a value.

File and/or directory names specify the file(s) the command is to process. Naming a directory is equivalent to naming all the SCCS files within the directory. Non-SCCS files and unreadable files (because of permission modes via **chmod**(1)) in the named directories are silently ignored.

In general, filename arguments may not begin with a minus sign. If a filename of – (a lone minus sign) is specified, the command will read the standard input (usually your terminal) for lines and take each line as the name of an SCCS file to be processed. The standard input is read until end-of-file. This feature is often used in pipelines with, for example, the commands **find**(1) or **ls**(1).

Keyletters are processed before filenames. Therefore, the placement of keyletters is arbitrary—that is, they may be interspersed with filenames. Filenames, however, are processed left to right. Somewhat different conventions apply to **help**(1), **what**(1), **sccsdiff**(1), and **val**(1), detailed later under "SCCS Commands."

Certain actions of various SCCS commands are controlled by flags appearing in SCCS files. Some of these flags will be discussed, but for a complete description see **admin**(1) in the *Programmer's Reference Manual.*

The distinction between real user (see **passwd**(1)) and effective user will be of concern in discussing various actions of SCCS commands. For now, assume that the real and effective users are the same—the person logged into the operating system.

x.files and z.files

All SCCS commands that modify an SCCS file do so by writing a copy called the "x.file." This is done to ensure that the SCCS file is not damaged if processing terminates abnormally. SCCS names the x.file by replacing the **s.** of the SCCS filename with **x.**. The x.file is created in the same directory as the SCCS file, given the same mode (see **chmod**(1)), and is owned by the effective user. When processing is complete, the old SCCS file is destroyed and the modified x.file is renamed (**x.** is replaced by **s.**) and becomes the new SCCS file.

To prevent simultaneous updates to an SCCS file, the same modifying commands also create a lock-file called the "z.file." SCCS forms its name by replacing the **s.** of the SCCS filename with a **z.** prefix. The z.file contains the process number of the command that creates it, and its existence prevents other commands from processing the SCCS file. The z.file is created with access permission mode 444 (read only) in the same directory as the SCCS file and is owned by the effective user. It exists only for the duration of the execution of the command that creates it.

In general, users can ignore x.files and z.files. They are useful only in the event of system crashes or similar situations.

Error Messages

SCCS commands produce error messages on the diagnostic output in this format:

```
ERROR [name-of-file-being-processed]: message text (code)
```

The code in parentheses can be used as an argument to the **help** command to obtain a further explanation of the message. Detection of a fatal error during the processing of a file causes the SCCS command to stop processing that file and proceed with the next file specified.

SCCS Commands

This section describes the major features of the fourteen SCCS commands and their most common arguments. Full descriptions with details of all arguments are in the *Programmer's Reference Manual*.

Here is a quick-reference overview of the commands:

get	retrieves versions of SCCS files
unget	undoes the effect of a **get –e** prior to the file being **delta**ed
delta	applies deltas (changes) to SCCS files and creates new versions
admin	initializes SCCS files, manipulates their descriptive text, and controls delta creation rights
prs	prints portions of an SCCS file in user specified format

sact	prints information about files that are currently out for edit
help	gives explanations of error messages
rmdel	removes a delta from an SCCS file allows removal of deltas created by mistake
cdc	changes the commentary associated with a delta
what	searches any operating system file(s) for all occurrences of a special pattern and prints out what follows it useful in finding identifying information inserted by the **get** command
sccsdiff	shows differences between any two versions of an SCCS file
comb	combines consecutive deltas into one to reduce the size of an SCCS file
val	validates an SCCS file
vc	a filter that may be used for version control

The get Command

The **get**(1) command creates a file that contains a specified version of an SCCS file. The version is retrieved by beginning with the initial version and then applying deltas, in order, until the desired version is obtained. The resulting file is called the "g.file." It is created in the current directory and is owned by the real user. The mode assigned to the g.file depends on how the **get** command is used.

The most common use of **get** is:

```
get s.abc
```

which normally retrieves the latest version of file **abc** from the SCCS file tree trunk and produces (for example) on the standard output:

```
1.3
67 lines
No id keywords (cm7)
```

meaning version 1.3 of file **s.abc** was retrieved (assuming 1.3 is the latest trunk delta), it has 67 lines of text, and no ID keywords were substituted in the file.

The generated g.file (file **abc**) is given access permission mode 444 (read only). This particular way of using **get** is intended to produce g.files only for inspection, compilation, etc. It is not intended for editing (making deltas).

When several files are specified, the same information is output for each one. For example, the command:

```
get s.abc s.xyz
```

produces:

```
s.abc:
1.3
67 lines
No id keywords (cm7)

s.xyz:
1.7
85 lines
No id keywords (cm7)
```

ID Keywords

In generating a g.file for compilation, it is useful to record the date and time of creation, the version retrieved, the module's name, etc. within the g.file. This information appears in a load module when one is eventually created. SCCS provides a convenient mechanism for doing this automatically. Identification (ID) keywords appearing anywhere in the generated file are replaced by appropriate values according to the definitions of those ID keywords. The format of an ID keyword is an uppercase letter enclosed by percent signs, %. For example, the ID keyword replaced by the SID of the retrieved version of a file is:

```
%I%
```

Similarly, **%H%** and **%M%** are the names of the g.file. Thus, executing **get** on an SCCS file that contains the PL/I declaration,

```
DCL ID CHAR(100) VAR INIT('%M% %I% %H%');
```

gives (for example) the following:

```
DCL ID CHAR(100) VAR INIT('MODNAME 2.3 07/18/85');
```

When no ID keywords are substituted by **get**, the following message is issued:

```
No id keywords (cm7)
```

This message is normally treated as a warning by **get** although the presence of the **i** flag in the SCCS file causes it to be treated as an error. For a complete list of the approximately twenty ID keywords provided, see **get**(1) in the *Programmer's Reference Manual*.

Retrieval of Different Versions

The version of an SCCS file that **get** retrieves is the most recently created delta of the highest numbered trunk release. However, any other version can be retrieved with **get –r** by specifying the version's SID. Thus, the command:

get –r1.3 s.abc

retrieves version 1.3 of file **s.abc** and produces (for example) on the standard output:

```
1.3
64 lines
```

A branch delta may be retrieved similarly, with a command such as:

get –r1.5.2.3 s.abc

which produces (for example) on the standard output:

```
1.5.2.3
234 lines
```

When a SID is specified and the particular version does not exist in the SCCS file, an error message results.

Omitting the level number, as in:

get –r3 s.abc

causes retrieval of the trunk delta with the highest level number within the given release. Thus, the above command might output:

```
3.7
213 lines
```

If the given release does not exist, **get** retrieves the trunk delta with the highest level number within the highest-numbered existing release that is lower than the given release. For example, assume release 9 does not exist in file **s.abc** and release 7 is the highest-numbered release below 9. Executing the command:

get –r9 s.abc

might produce:

```
7.6
420 lines
```

which indicates that trunk delta 7.6 is the latest version of file **s.abc** below release 9. Similarly, omitting the sequence number, as in:

```
get -r4.3.2 s.abc
```

results in the retrieval of the branch delta with the highest sequence number on the given branch. (If the given branch does not exist, an error message results.) This might result in the following output:

```
4.3.2.8
89 lines
```

get -t will retrieve the latest (top) version of a particular release when no **-r** is used or when its value is simply a release number. The latest version is the delta produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is 3.5, the command:

```
get -r3 -t s.abc
```

might produce:

```
3.5
59 lines
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command might produce:

```
3.2.1.5
46 lines
```

Retrieval With Intent to Make a Delta

get -e indicates an intent to make a delta. First, **get** checks the following items:

1. The user list to determine whether the login name or group ID of the person executing **get** is present. The login name or group ID must be present for the user to be allowed to make deltas. (See "The **admin** Command" for a discussion of making user lists.)
2. Whether the release number (R) of the version being retrieved satisfies the relation:

   floor is less than or equal to R, which is
   less than or equal to ceiling

   to determine if the release being accessed is a protected release. The floor and ceiling are flags in the SCCS file representing start and end of range.
3. That the R is not locked against editing. The lock is a flag in the SCCS file.

4. Whether multiple concurrent edits are allowed for the SCCS file by the **j** flag in the SCCS file.

A failure of any of the first three conditions causes the processing of the corresponding SCCS file to terminate.

If the above checks succeed, **get –e** causes the creation of a g.file in the current directory with mode 644 (readable by everyone, writable only by the owner) owned by the real user. If a writable g.file already exists, **get** terminates with an error. This is to prevent inadvertent destruction of a g.file being edited to make a delta.

Any ID keywords appearing in the g.file are not substituted by **get –e** because the generated g.file is subsequently used to create another delta. Replacement of ID keywords causes them to be permanently changed in the SCCS file. Because of this, **get** does not need to check for their presence in the g.file. Thus, the message:

```
No id keywords (cm7)
```

is never output when **get –e** is used.

In addition, **get –e** causes the creation (or updating) of a p.file that is used to pass information to the **delta** command.

The command:

```
get –e s.abc
```

produces (for example) on the standard output:

```
1.3
new delta 1.4
67 lines
```

Undoing a get –e

There may be times when a file is retrieved for editing in error; there is really no editing that needs to be done at this time. In such cases, the **unget** command can be used to cancel the delta reservation that was set up.

Additional get Options

If **get –r** and/or **–t** are used together with **–e**, the version retrieved for editing is the one specified with **–r** and/or **–t**.

get –i and **–x** are used to specify a list (see **get**(1) in the *Programmer's Reference Manual* for the syntax of such a list) of deltas to be included and excluded, respectively. Including a delta means forcing its changes to be included in the retrieved version. This is useful in applying the same changes to more than one version of the SCCS file. Excluding a delta means forcing it not to be applied. This may be used to undo the effects of a previous delta in the version to be created.

Whenever deltas are included or excluded, **get** checks for possible interference with other deltas. Two deltas can interfere, for example, when each one changes the same line of the retrieved g.file. A warning shows the range of lines within the retrieved g.file where the problem may exist. The user should examine the g.file to determine what the problem is and take corrective steps (e.g., edit the file).

CAUTION

get –i and **get –x** should be used with extreme care.

get –k is used either to regenerate a g.file that may have been accidentally removed or ruined after **get –e**, or simply to generate a g.file in which the replacement of ID keywords has been suppressed. A g.file generated by **get –k** is identical to one produced by **get –e**, but no processing related to the p.file takes place.

Concurrent Edits of Different SID

The ability to retrieve different versions of an SCCS file allows several deltas to be in progress at any given time. This means that several **get –e** commands may be executed on the same file as long as no two executions retrieve the same version (unless multiple concurrent edits are allowed).

The p.file created by **get –e** is named by automatic replacement of the SCCS filename's prefix **s.** with **p.**. It is created in the same directory as the SCCS file, given mode 644 (readable by everyone, writable only by the owner), and owned by the effective user. The p.file contains the following information for each delta that is still in progress:

- the SID of the retrieved version
- the SID given to the new delta when it is created
- the login name of the real user executing **get**

The first execution of **get –e** causes the creation of a p.file for the corresponding SCCS file. Subsequent executions only update the p.file with a line containing the above information. Before updating, however, **get** checks to assure that no entry already in the p.file specifies that the SID of the version to be retrieved is already retrieved (unless multiple concurrent edits are allowed). If the check succeeds, the user is informed that other deltas are in progress and processing continues. If the check fails, an error message results.

Note that concurrent executions of **get** must be carried out from different directories. Subsequent executions from the same directory will attempt to overwrite the g.file, which is an SCCS error condition. In practice, this problem does not arise since each user normally has a different working directory. See "Protection" under "SCCS Files" for a discussion of how different users are permitted to use SCCS commands on the same files.

Figure 15-4 shows the possible SID components a user can specify with **get** (left-most column), the version that will then be retrieved by **get**, and the resulting SID for the delta, which **delta** will create (right-most column).

SID Specified in get*	–b Key-Letter Used†	Other Conditions	SID Retrieved by get	SID of Delta To be Created by delta
none‡	no	R defaults to mR	mR.mL	mR.(mL+1)
none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB+1)
R	no	R > mR	mR.mL	R.1§
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R = mR	mR.mL	mR.mL.(mB+1).1

Figure 15-4. Determination of New SID (sheet 1 of 2)

15

SID Specified in get*	−b Key-Letter Used†	Other Conditions	SID Retrieved by get	SID of Delta To be Created by delta
R	–	R< mR and R does not exist	hR.mL**	hR.mL.(mB+1).1
R	–	Trunk successor number in release > R and R exists	R.mL	R.mL.(mB+1).1
R.L.	no	No trunk successor	R.L	R.(L+1)
R.L.	yes	No trunk successor	R.L	R.L.(mB+1).1
R.L	–	Trunk successor in release ≥ R	R.L	R.L.(mS+1).1
R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB+1).1
R.L.B.S	–	Branch successor	R.L.B.S	R.L.(mB+1).1

Figure 15-4. Determination of New SID (sheet 2 of 2)

Footnotes to Figure 15-4:

* R, L, B, and S mean release, level, branch, and sequence numbers in the SID, and m means maximum. Thus, for example, R.mL means the maximum level number within release R. R.L.(mB+1).1 means the first

sequence number on the new branch (i.e., maximum branch number plus 1) of level L within release R. Note that if the SID specified is R.L, R.L.B, or R.L.B.S, each of these specified SID numbers must exist.

† The **–b** keyletter is effective only if the **b** flag (see **admin(1)**) is present in the file. An entry of – means irrelevant.

‡ This case applies if the **d** (default SID) flag is not present. If the **d** flag is present in the file, the SID is interpreted as if specified on the command line. Thus, one of the other cases in this figure applies.

§ This is used to force the creation of the first delta in a new release.

** hR is the highest existing release that is lower than the specified, nonexistent release R.

Concurrent Edits of Same SID

Under normal conditions, more than one **get –e** for the same SID is not permitted. That is, **delta** must be executed before a subsequent **get –e** is executed on the same SID.

Multiple concurrent edits are allowed if the **j** flag is set in the SCCS file. Thus, the command and output:

```
get –e s.abc
1.1
new delta 1.2
5 lines
```

may be immediately followed by:

```
get –e s.abc
1.1
new delta 1.1.1.1
5 lines
```

without an intervening **delta**. In this case, a **delta** after the first **get** will produce delta 1.2 (assuming 1.1 is the most recent trunk delta), and a **delta** after the second **get** will produce delta 1.1.1.1.

Keyletters That Affect Output

get –p causes the retrieved text to be written to the standard output rather than to a g.file. In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the diagnostic output. **get –p** is used, for example, to create a g.file with an arbitrary name, as in:

```
get -p s.abc > arbitrary-file-name
```

get –s suppresses output normally directed to the standard output, such as the SID of the retrieved version and the number of lines retrieved, but it does not affect messages normally directed to the diagnostic output. **get –s** is used to prevent nondiagnostic messages from appearing on the user's terminal and is often used with **–p** to pipe the output, as in:

```
get -p -s s.abc | pg
```

get –g suppresses the retrieval of the text of an SCCS file. This is useful in several ways. For example, to verify a particular SID in an SCCS file, the command:

```
get -g -r4.3 s.abc
```

outputs the SID 4.3 if it exists in the SCCS file **s.abc** or an error message if it does not. Another use of **get –g** is in regenerating a p.file that may have been accidentally destroyed, as in:

```
get -e -g s.abc
```

get –l causes SCCS to create an "l.file." It is named by replacing the **s.** of the SCCS filename with **l.**, created in the current directory with mode 444 (read only) and owned by the real user. The l.file contains a table (whose format is described under **get**(1) in the *Programmer's Reference Manual*) showing the deltas used in constructing a particular version of the SCCS file. For example, the command:

```
get -r2.3 -l s.abc
```

generates an l.file showing the deltas applied to retrieve version 2.3 of file **s.abc**. Specifying **p** with **–l**, as in:

```
get -lp -r2.3 s.abc
```

causes the output to be written to the standard output rather than to the l.file. **get –g** can be used with **–l** to suppress the retrieval of the text.

get –m identifies the changes applied to an SCCS file. Each line of the g.file is preceded by the SID of the delta that caused the line to be inserted. The SID is separated from the text of the line by a tab character.

get –n causes each line of a g.file to be preceded by the value of the ID keyword and a tab character. This is most often used in a pipeline with **grep**(1). For example, to find all lines that match a given pattern in the latest version of each SCCS file in a directory, the following may be executed:

get –p –n –s *directory* **| grep** *pattern*

If both **–m** and **–n** are specified, each line of the generated g.file is preceded by the value of the **chap3.13** ID keyword and a tab (this is the effect of **–n**) and is followed by the line in the format produced by **–m**. Because use of **–m** and/or **–n** causes the contents of the g.file to be modified, such a g.file must not be used for creating a delta. Therefore, neither **–m** nor **–n** may be specified together with **get –e**.

NOTE

See **get**(1) in the *Programmer's Reference Manual* for a full description of additional keyletters.

The delta Command

The **delta**(1) command is used to incorporate changes made to a g.file into the corresponding SCCS file—that is, to create a delta and, therefore, a new version of the file.

The **delta** command requires the existence of a p.file (created via **get –e**). It examines the p.file to verify the presence of an entry containing the user's login name. If none is found, an error message results.

get –e performs. If all checks are successful, **delta** determines what has been changed in the g.file by comparing it via **diff**(1) with its own temporary copy of the g.file as it was before editing. This temporary copy of the g.file is called the d.file and is obtained by performing an internal **get** on the SID specified in the p.file entry.

The required p.file entry is the one containing the login name of the user executing **delta**, because the user who retrieved the g.file must be the one who creates the delta. However, if the login name of the user appears in more than one entry, the same user has executed **get –e** more than once on the same SCCS file. Then, **delta –r** must be used to specify the SID that uniquely identifies the p.file entry. This entry is then the one used to obtain the SID of the delta to be created.

In practice, the most common use of **delta** is:

```
delta s.abc
```

which prompts:

```
comments?
```

to which the user replies with a description of why the delta is being made, ending the reply with a newline character. The user's response may be up to 512 characters long with newlines (not intended to terminate the response) escaped by backslashes, \.

If the SCCS file has a **v** flag, **delta** first prompts with:

```
MRs?
```

(Modification Requests), on the standard output. The standard input is then read for MR numbers, separated by blanks and/or tabs, ended with a newline character. A Modification Request is a formal way of asking for a correction or enhancement to the file. In some controlled environments where changes to source files are tracked, deltas are permitted only when initiated by a trouble report, change request, trouble ticket, etc., collectively called MRs. Recording MR numbers within deltas is a way of enforcing the rules of the change management process.

delta –y and/or **–m** can be used to enter comments and MR numbers on the command line rather than through the standard input, as in:

delta –y"*descriptive comment*" **–m**"*mrnum1 mrnum2*" **s.abc**

In this case, the prompts for comments and MRs are not printed, and the standard input is not read. These two keyletters are useful when **delta** is executed from within a shell procedure (see **sh**(1) in the *Programmer's Reference Manual*).

NOTE

delta –m is allowed only if the SCCS file has a **v** flag.

No matter how comments and MR numbers are entered with **delta**, they are recorded as part of the entry for the delta being created. Also, they apply to all SCCS files specified with the **delta**.

If **delta** is used with more than one file argument and the first file named has a **v** flag, all files named must have this flag. Similarly, if the first file named does not have the flag, none of the files named may have it.

When **delta** processing is complete, the standard output displays the SID of the new delta (from the p.file) and the number of lines inserted, deleted, and left unchanged. For example:

```
1.4
14 inserted
7 deleted
345 unchanged
```

If line counts do not agree with the user's perception of the changes made to a g.file, it may be because there are various ways to describe a set of changes, especially if lines are moved around in the g.file. However, the total number of lines of the new delta (the number inserted plus the number left unchanged) should always agree with the number of lines in the edited g.file.

If you are in the process of making a delta, the **delta** command finds no ID keywords in the edited g.file, the message:

```
No id keywords (cm7)
```

is issued after the prompts for commentary but before any other output. This means that any ID keywords that may have existed in the SCCS file have been replaced by their values or deleted during the editing process. This could be caused by making a delta from a g.file that was created by a **get** without **–e** (ID keywords are replaced by **get** in such a case). It could also be caused by accidentally deleting or changing ID keywords while editing the g.file. Or, it is possible that the file had no ID keywords. In any case, the delta will be created unless there is an **i** flag in the SCCS file (meaning the error should be treated as fatal), in which case the delta will not be created.

After the processing of an SCCS file is complete, the corresponding p.file entry is removed from the p.file. All updates to the p.file are made to a temporary copy, the "q.file," whose use is similar to the use of the x.file described earlier under "SCCS Command Conventions." If there is only one entry in the p.file, then the p.file itself is removed.

In addition, **delta** removes the edited g.file unless **–n** is specified. For example, the command:

```
delta –n s.abc
```

will keep the g.file after processing.

delta –s suppresses all output normally directed to the standard output, other than `comments?` and `MRs?`. Thus, use of **–s** with **–y** (and/or **–m**) causes **delta** to neither read the standard input nor write the standard output.

The differences between the g.file and the d.file constitute the delta and may be printed on the standard output by using **delta -p**. The format of this output is similar to that produced by **diff**(1).

The admin Command

The **admin**(1) command is used to administer SCCS files—that is, to create new SCCS files and change the parameters of existing ones. When an SCCS file is created, its parameters are initialized by use of keyletters with **admin** or are assigned default values if no keyletters are supplied. The same keyletters are used to change the parameters of existing SCCS files.

Two keyletters are used in detecting and correcting corrupted SCCS files (see "Auditing" under "SCCS Files").

Newly created SCCS files are given access permission mode 444 (read only) and are owned by the effective user. Only a user with write permission in the directory containing the SCCS file may use the **admin** command on that file.

Creation of SCCS Files

An SCCS file can be created by executing the command:

```
admin -ifirst s.abc
```

in which the value **first** with **-i** is the name of a file from which the text of the initial delta of the SCCS file **s.abc** is to be taken. Omission of a value with **-i** means **admin** is to read the standard input for the text of the initial delta.

The command:

```
admin -i s.abc < first
```

is equivalent to the previous example.

If the text of the initial delta does not contain ID keywords, the message:

```
No id keywords (cm7)
```

is issued by **admin** as a warning. However, if the command also sets the **i** flag (not to be confused with the **-i** keyletter), the message is treated as an error and the SCCS file is not created. Only one SCCS file may be created at a time using **admin -i**.

admin –r is used to specify a release number for the first delta. Thus, the command:

```
admin -ifirst -r3 s.abc
```

means the first delta should be named 3.1 rather than the normal 1.1. Because **–r** has meaning only when creating the first delta, its use is permitted only with **–i**.

Inserting Commentary for the Initial Delta

When an SCCS file is created, the user may want to record why this was done. Comments (**admin –y**) and/or MR numbers (**–m**) can be entered in exactly the same way as a **delta**.

If **–y** is omitted, a comment line of the form:

```
date and time created YY/MM/DD HH:MM:SS by logname
```

is automatically generated.

If it is desired to supply MR numbers (**admin –m**), the **v** flag must be set via **–f**. The **v** flag simply determines whether MR numbers must be supplied when using any SCCS command that modifies a delta commentary (see **sccsfile**(4) in the *Programmer's Reference Manual*) in the SCCS file. Thus:

```
admin -ifirst -mmrnum1 -fv s.abc
```

Note that **–y** and **–m** are effective only if a new SCCS file is being created.

Initialization and Modification of SCCS File Parameters

Part of an SCCS file is reserved for descriptive text, usually a summary of the file's contents and purpose. It can be initialized or changed by using **admin –t**.

When an SCCS file is first being created and **–t** is used, it must be followed by the name of a file from which the descriptive text is to be taken. For example, the command:

```
admin -ifirst -tdesc s.abc
```

specifies that the descriptive text is to be taken from file **desc**.

When processing an existing SCCS file, **–t** specifies that the descriptive text (if any) currently in the file is to be replaced with the text in the named file. Thus, the command:

```
admin -tdesc s.abc
```

specifies that the descriptive text of the SCCS file is to be replaced by the contents of **desc**.

Omission of the filename after the **–t** keyletter, as in:

```
admin –t s.abc
```

causes the removal of the descriptive text from the SCCS file.

The flags of an SCCS file may be initialized or changed by **admin –f**, or deleted via **–d**.

SCCS file flags are used to direct certain actions of the various commands. (See **admin**(1) in the *Programmer's Reference Manual* for a description of all the flags.) For example, the **i** flag specifies that a warning message (stating that there are no ID keywords contained in the SCCS file) should be treated as an error. The **d** (default SID) flag specifies the default version of the SCCS file to be retrieved by the **get** command.

admin –f is used to set flags and, if desired, their values. For example, the command:

```
admin –ifirst –fi –fmmodname s.abc
```

sets the **i** and **m** (module name) flags. The value *modname* specified for the **m** flag is the value that the **get** command will use to replace the **%M%** ID keyword. (In the absence of the **m** flag, the name of the g.file is used as the replacement for the **%M%** ID keyword.) Several **–f** keyletters may be supplied on a single **admin**, and they may be used whether the command is creating a new SCCS file or processing an existing one.

admin –d is used to delete a flag from an existing SCCS file. As an example, the command:

```
admin –dm s.abc
```

removes the **m** flag from the SCCS file. Several **–d** keyletters may be used with one **admin** and may be intermixed with **–f**.

SCCS files contain a list of login names and/or group IDs of users who are allowed to create deltas. This list is empty by default, allowing anyone to create deltas. To create a user list (or add to an existing one), **admin –a** is used. For example, the command:

```
admin –axyz –awql –a1234 s.abc
```

adds the login names **xyz** and **wql** and the group ID **1234** to the list. **admin –a** may be used whether creating a new SCCS file or processing an existing one.

admin –e (erase) is used to remove login names or group IDs from the list.

The prs Command

The **prs**(1) command is used to print all or part of an SCCS file on the standard output. If **prs –d** is used, the output will be in a format called data specification. Data specification is a string of SCCS file data keywords (not to be confused with **get** ID keywords) interspersed with optional user text.

Data keywords are replaced by appropriate values according to their definitions. For example, the symbol:

```
:I:
```

is defined as the data keyword replaced by the SID of a specified delta. Similarly, **:F:** is the data keyword for the SCCS filename currently being processed, and **:C:** is the comment line associated with a specified delta. All parts of an SCCS file have an associated data keyword. For a complete list, see **prs**(1) in the *Programmer's Reference Manual*.

There is no limit to the number of times a data keyword may appear in a data specification. Thus, for example, the command line:

```
prs –d":I: this is the top delta for :F: :I:" s.abc
```

may produce on the standard output:

```
2.1 this is the top delta for s.abc 2.1
```

Information may be obtained from a single delta by specifying its SID using **prs –r**. For example, the command line:

```
prs –d":F:: :I: comment line is: :C:" –r1.4 s.abc
```

may produce the following output:

```
s.abc: 1.4 comment line is: THIS IS A COMMENT
```

If **–r** is not specified, the value of the SID defaults to the most recently created delta.

In addition, information from a range of deltas may be obtained with **–l** or **–e**. The use of **prs –e** substitutes data keywords for the SID designated via **–r** and all deltas created earlier, while **prs –l** substitutes data keywords for the SID designated via **–r** and all deltas created later. Thus, the command:

```
prs –d:I: –r1.4 –e s.abc
```

may output:

```
1.4
1.3
1.2.1.1
1.2
1.1
```

and the command:

prs -d:I: -r1.4 -l s.abc

may produce:

```
3.3
3.2
3.1
2.2.1.1
2.2
2.1
1.4
```

Substitution of data keywords for all deltas of the SCCS file may be obtained by specifying both **-e** and **-l**.

The sact Command

sact(1) is like a special form of the **prs** command that produces a report about files that are out for edit. The command takes only one type of argument: a list of file or directory names. The report shows the SID of any file in the list that is out for edit, the SID of the impending delta, the login of the user who executed the **get -e** command, and the date and time the **get -e** was executed. It is a useful command for an administrator.

The help Command

The **help**(1) command prints the syntax of SCCS commands and of messages that may appear on the user's terminal. Arguments to **help** are simply SCCS commands or the code numbers that appear in parentheses after SCCS messages. (If no argument is given, **help** prompts for one.) Explanatory information is printed on the standard output. If no information is found, an error message is printed. When more than one argument is used, each is processed independently, and an error resulting from one will not stop the processing of the others.

NOTE

There is no conflict between the **help**(1) command of SCCS and the operating system **help**(1) utilities. The installation procedure for each package checks for the prior existence of the other.

Explanatory information related to a command is a synopsis of the command. For example, the command:

help ge5 rmdel

produces:

```
ge5:
"nonexistent sid"
The specified sid does not exist in the
given file.
Check for typos.

rmdel:
  rmdel  -rSID  name  ...
```

The rmdel Command

The **rmdel**(1) command allows removal of a delta from an SCCS file. Its use should be reserved for deltas in which incorrect global changes were made. The delta to be removed must be a leaf delta. That is, it must be the most recently created delta on its branch or on the trunk of the SCCS file tree. In Figure 15-3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed. Only after they are removed can deltas 1.3.2.1 and 2.1 be removed.

To be allowed to remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must be either the one who created the delta being removed or the owner of the SCCS file and its directory.

The **–r** keyletter is mandatory with **rmdel**. It is used to specify the complete SID of the delta to be removed. Thus, the command:

rmdel –r2.3 s.abc

specifies the removal of trunk delta 2.3.

Before removing the delta, **rmdel** checks that the release number (R) of the given SID satisfies the relation:

floor less than or equal to R less than or equal to ceiling

The **rmdel** command also checks the SID to make sure it is not for a version on which a **get** for editing has been executed and whose associated **delta** has not yet been made. In addition, the login name or group ID of the user must appear in the file's user list (or the user list must be empty). Also, the release specified cannot be locked against editing. That is, if the **l** flag is set (see **admin**(1) in the *Programmer's Reference Manual*), the release must not be contained in the list. If these conditions are not satisfied, processing is terminated, and the delta is not removed.

Once a specified delta has been removed, its type indicator in the delta table of the SCCS file is changed from D (delta) to R (removed).

The cdc Command

The **cdc**(1) command is used to change the commentary made when the delta was created. It is similar to the **rmdel** command (e.g., **–r** and full SID are necessary), although the delta need not be a leaf delta. For example, the command:

```
cdc –r3.4 s.abc
```

specifies that the commentary of delta 3.4 is to be changed. New commentary is then prompted for as with **delta**.

The old commentary is kept, but it is preceded by a comment line indicating that it has been superseded, and the new commentary is entered ahead of the comment line. The inserted comment line records the login name of the user executing **cdc** and the time of its execution.

The **cdc** command also allows for the insertion of new and deletion of old ("!" prefix) MR numbers. Thus, the command:

```
cdc –r1.4 s.abc
MRs? mrnum3 !mrnum1           (The MRs? prompt appears only
                               if the v flag has been set.)
comments? deleted wrong MR number and inserted correct MR number
```

inserts **mrnum3** and deletes **mrnum1** for delta 1.4.

NOTE

An MR (Modification Request) is described above under the **delta** command.

The what Command

The **what**(1) command is used to find identifying information within any file whose name is given as an argument. No keyletters are accepted. The **what** command searches the given file(s) for all occurrences of the string **@(#)**, which is the replacement for the **%Z%** ID keyword (see **get**(1)). It prints on the standard output whatever follows the string until the first double quote, ", greater than, >, backslash, \, newline, or nonprinting NUL character.

For example, if an SCCS file called **s.prog.c** (a C language program) contains the following line:

```
char  id[]= "%W%";
```

and if the command:

```
get –r3.4 s.prog.c
```

is used, the resulting g.file is compiled to produce **prog.o** and **a.out**. Then, the command:

```
what prog.c prog.o a.out
```

produces:

```
prog.c:
  prog.c:  3.4
prog.o:
  prog.c:  3.4
a.out:
  prog.c:  3.4
```

The string searched for by **what** need not be inserted via an ID keyword of **get**; it may be inserted in any convenient manner.

The sccsdiff Command

The **sccsdiff**(1) command determines (and prints on the standard output) the differences between any two versions of an SCCS file. The versions to be compared are specified with **sccsdiff –r** in the same way as with **get –r**. SID numbers must be specified as the first two arguments. Any following keyletters are interpreted as arguments to the **pr**(1) command (which prints the differences) and must appear before any filenames. The SCCS file(s) to be processed are named last. Directory names and a name of – (a lone minus sign) are not acceptable to **sccsdiff**.

The following is an example of the format of **sccsdiff**:

```
sccsdiff -r3.4 -r5.6 s.abc
```

The differences are printed the same way as by **diff**(1).

The comb Command

The **comb**(1) command lets the user try to reduce the size of an SCCS file. It generates a shell procedure (see **sh**(1) in the *Programmer's Reference Manual*) on the standard output, which reconstructs the file by discarding unwanted deltas and combining other specified deltas. (It is not recommended that **comb** be used as a matter of routine.)

In the absence of any keyletters, **comb** preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the shape of an SCCS tree. The effect of this is to eliminate middle deltas on the trunk and on all branches of the tree. Thus, in Figure 15-3, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated.

Some of the keyletters used with this command are:

comb –s	This option generates a shell procedure that produces a report of the percentage space (if any) the user will save. This is often useful as an advance step.
comb –p	This option is used to specify the oldest delta the user wants preserved.
comb –c	This option is used to specify a list (see **get**(1) in the *Programmer's Reference Manual* for its syntax) of deltas the user wants preserved. All other deltas will be discarded.

The shell procedure generated by **comb** is not guaranteed to save space. A reconstructed file may even be larger than the original. Note, too, that the shape of an SCCS file tree may be altered by the reconstruction process.

The val Command

The **val**(1) command is used to determine whether a file is an SCCS file meeting the characteristics specified by certain keyletters. It checks for the existence of a particular delta when the SID for that delta is specified with **–r**.

The string following **–y** or **–m** is used to check the value set by the **t** or **m** flag, respectively. See **admin**(1) in the *Programmer's Reference Manual* for descriptions of these flags.

The **val** command treats the special argument – differently from other SCCS commands. It allows **val** to read the argument list from the standard input instead of from the command line, and the standard input is read until an end-of-file (CTRL-D) is entered. This permits one **val** command with different values for keyletters and file arguments. For example, the command:

```
val – –yc –mabc s.abc –mxyz –ypl1 s.xyz
```

first checks if file **s.abc** has a value **c** for its type flag and value **abc** for the module name flag. Once this is done, **val** processes the remaining file, in this case **s.xyz**.

The **val** command returns an 8-bit code. Each bit set shows a specific error (see **val**(1) for a description of errors and codes). In addition, an appropriate diagnostic is printed unless suppressed by **–s**. A return code of 0 means all files met the characteristics specified.

The vc Command

The **vc**(1) command is an **awk**-like tool used for version control of sets of files. While it is distributed as part of the SCCS package, it does not require the files it operates on to be under SCCS control. A complete description of **vc** may be found in the *Programmer's Reference Manual*.

SCCS Files

This section covers protection mechanisms used by SCCS, the format of SCCS files, and the recommended procedures for auditing SCCS files.

Protection

SCCS relies on the capabilities of the operating system for most of the protection mechanisms required to prevent unauthorized changes to SCCS files—that is, changes by non-SCCS commands. Protection features provided directly by SCCS are the release lock flag, the release floor and ceiling flags, and the user list.

Files created by the **admin** command are given access permission mode 444 (read only). This mode should remain unchanged because it prevents modification of SCCS files by non-SCCS commands. Directories containing SCCS files should be given mode 755, which allows only the owner of the directory to modify it.

SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies their protection and auditing. The contents of directories should be logical groupings—subsystems of the same large project, for example.

SCCS files should have only one link (name) because commands that modify them do so by creating a copy of the file (the x.file; see "SCCS Command Conventions"). When processing is done, the old file is automatically removed and the x.file renamed (**s.** prefix). If the old file had additional links, this breaks them. Then, rather than process such files, SCCS commands will produce an error message.

When only one person uses SCCS, the real and effective user IDs are the same; and the user ID owns the directories containing SCCS files. Therefore, SCCS may be used directly without any preliminary preparation.

When several users with unique user IDs are assigned SCCS responsibilities (e.g., on large development projects), one user—that is, one user ID—must be chosen as the owner of the SCCS files. This person will administer the files (e.g. use the **admin** command) and will be SCCS administrator for the project. Because other users do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the **get**, **delta**, and, if desired, **rmdel** and **cdc** commands.

The interface program must be owned by the SCCS administrator and must have the set user ID on execution bit on (see **chmod**(1) in the *User's Reference Manual*). This assures that the effective user ID is the user ID of the SCCS administrator. With the privileges of the interface program during command execution, the owner of an SCCS file can modify it at will. Other users whose login names or group IDs are in the user list for that file (but are not the owner) are given the necessary permissions only for the duration of the execution of the interface program. Thus, they may modify SCCS only with **delta** and, possibly, **rmdel** and **cdc**.

A project-dependent interface program, as its name implies, can be custom built for each project. Its creation is discussed later under "An SCCS Interface Program."

Formatting

SCCS files are composed of lines of ASCII text arranged in six parts as follows:

Checksum	a line containing the logical sum of all the characters of the file (not including the checksum itself)
Delta Table	information about each delta, such as type, SID, date and time of creation, and commentary
User Names	list of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas
Flags	indicators that control certain actions of SCCS commands
Descriptive Text	usually a summary of the contents and purpose of the file
Body	the text administered by SCCS, intermixed with internal SCCS control lines

Details on these file sections may be found in **sccsfile**(4). The checksum is discussed below under "Auditing."

Since SCCS files are ASCII files they can be processed by non-SCCS commands like **ed**(1), **grep**(1), and **cat**(1). This is convenient when an SCCS file must be modified manually (e.g., a delta's time and date were recorded incorrectly because the system clock was set incorrectly), or when a user wants simply to look at the file.

CAUTION

Extreme care should be exercised when modifying SCCS files with non-SCCS commands.

Auditing

When a system or hardware malfunction destroys an SCCS file, any command will issue an error message. Commands also use the checksum stored in an SCCS file to determine whether the file has been corrupted since it was last accessed (possibly by having lost one or more blocks or by having been modified with **ed**(1)). No SCCS command will process a corrupted SCCS file except the **admin** command with **–h** or **–z**, as described below.

SCCS files should be audited for possible corruptions on a regular basis. The simplest and fastest way to do an audit is to use **admin –h** and specify all SCCS files, as in:

> **admin –h s.***file1* **s.***file2* ...

or

> **admin –h** *directory1 directory2* ...

If the new checksum of any file is not equal to the checksum in the first line of that file, the message:

```
corrupted file (co6)
```

is produced for that file. The process continues until all specified files have been examined. When examining directories (as in the second example above), the checksum process will not detect missing files. A simple way to learn whether files are missing from a directory is to execute the **ls**(1) command periodically, and compare the outputs. Any file whose name appeared in a previous output but not in the current one no longer exists.

When a file has been corrupted, the way to restore it depends on the extent of the corruption. If damage is extensive, the best solution is to contact the local system administrator and request that the file be restored from a backup copy. If the damage is minor, repair through editing may be possible. After such a repair, the **admin** command must be executed:

> **admin –z s.***file*

The purpose of this is to recompute the checksum and bring it into agreement with the contents of the file. After this command is executed, any corruption that existed in the file will no longer be detectable.

CHAPTER 16
sdb—THE SYMBOLIC DEBUGGER

Introduction

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

Using sdb

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 16-1, 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 way, 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 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, the command:

```
*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 the command:

```
*1000.x/
```

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

```
*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 the command:

```
*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. There are facilities 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 operating 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 indicates 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 following forms may be used:

```
*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 that search 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 the command:

```
*+15z
```

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

A Controlled Environment for Program Testing

One 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, the command:

```
        *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 from 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 the command:

```
        *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 the command:

```
*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 the command:

```
*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 16-1. 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
0x2462 (window:1459+2) b
*r < m.s > out.m.s     # run the program
Breakpoint at
0x2462 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=0x1FFFE43,-1073610300)    [optim.c:227]
*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
0x86b38
*x                     # print the register contents
  r0/ 0x86b38           r1/ 0                    r2/ 0x8796c
  r3/ 0x85830           r4/ 0x1FFFB8F            r5/ 0x1FFF907
  r6/ 0x1FFFB87            r7/ 0x86b38           r8/ 2
  ap/ 0x1FFFD23            fp/ 0x1FFFCF7            sp/ 0x1FFFCF7
 psw/ 0x2004             pc/ 0x24b0
0x24b0 (window:1475):         MOVW    0x80d8c,%r0   [-0x7f77f274,%r0]
```

Figure 16-1. Example of **sdb** Usage (Sheet 1 of 2)

```
*pl[0]                  # dereference the pointer
pl[0].forw/ 0x86b6c
pl[0].back/ 0x86ac8
pl[0].ops[0]/ mov.w
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/ 0x86ca0
pl->forw[0].back/ 0x86b38
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
0x86b6c
*c                      # continue
Breakpoint at
0x2462 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 16-1. Example of **sdb** Usage (Sheet 2 of 2)

CHAPTER 17
lint

Introduction

The **lint** program examines C language source programs for 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 portability restrictions involved in moving programs between different machines and/or operating systems. It detects a number of legal but wasteful or error prone constructions. **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 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 that 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 common for external variables or even entire functions to become unnecessary and yet not be removed from the source. Although these types of errors rarely cause working programs to fail, they 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 (unless suppressed by the **–u** or **–x** option) about variables and functions which are defined but not otherwise mentioned.

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. Sometimes, 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, the comment:

```
/* 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 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 the statements:

```
return(  expr );
```

and:

```
return ;
```

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. This comment in the source code will cause the message to be suppressed:

```
/*NOTREACHED*/
```

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), as in:

```
        (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**. It is hoped the messages 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 of the following:

```
a =- 1 ;

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* 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.

So that the efficiency of C language on a particular machine is not 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
```

to call attention to this condition.

CHAPTER 18
C LANGUAGE

Introduction

This chapter contains a summary of the grammar and syntax rules of the C Programming Language. 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	**double**	**if**	**struct**
auto	**else**	**int**	**switch**
break	**enum**	**long**	**typedef**
case	**external**	**register**	**union**
char	**float**	**return**	**unsigned**
continue	**for**	**short**	**void**
default	**fortran**	**sizeof**	**while**
do	**goto**	**static**	

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, integer and long values may be considered identical on some computers.

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 18-1.

```
new-lineNL (LF)\n
horizontal tabHT\t
vertical tabVT\v
backspaceBS\b
carriage returnCR\r
form feedFF\f
backslash\\\
single quote´\´
bit patternddd\ddd
```

Figure 18-1. 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 on 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 on exit from the block. Static variables are local to a block but retain their values on 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 computers based on the M68000 family of microprocessors are shown in Figure 18-2.

Motorola M68000 Family (ASCII)	
char	8 bits
int	32
short	16
long	32
float	32
double	64
float range	IEEE specification
double range	IEEE specification

Figure 18-2. M68000 Family-Based 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 computer sign extension of **char** variables does not occur. 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; 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 from 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 from 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 way 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 from 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. 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 from 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 always 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. 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 from 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 from 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 from 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 from 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 from right to left. The first expression is evaluated; 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 from 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 from 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 from 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, the expression:

```
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. It 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 class: 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 with **int**; the meaning is the same as if **int** were not mentioned. The word **long** may be specified with **float**; the meaning is the same as **double**. The word **unsigned** may be specified alone, or with **int** or any of its short or long varieties, or with **char**.

Otherwise, at most one 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 of the form:

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 the use of 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. This function returns an integer. As another example, the function:

```
float fa[17], *afp[17];
```

declares an array of **float** numbers and an array of pointers to **float** numbers. Finally, the expression:

```
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 from 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 18-2 for sizes of basic types.)

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; the expression:

s.left

refers to the left subtree pointer of the structure **s**; and the expression:

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, the segment:

```
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, the expression:

```
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. The expression:

```
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, the expression:

```
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, the expression:

```
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 via 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, the expressions:

```
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, **distance** in the example above 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 ;

Expression statements are usually 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** ($exp\text{–}1_{opt}$; $exp\text{–}2_{opt}$; $exp\text{–}3_{opt}$) *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 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 below, a **continue** is equivalent to **goto contin**:

```
while (...)          do                   for (...)
{                    {                    {
      ...                  ...                  ...
contin: ;            contin: ;            contin: ;
}                    } while (...);       }
```

(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 below:

> **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 subblocks 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 the "Structure and Union" and "Enumeration" sections under "Declarations") that tags, identifiers associated with ordinary variables, and identities associated with structure and union members form three distinct 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 second declaration must contain the **int**, or it would be taken as 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 mult-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 **#define**d 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.)

The **#include**s 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 1 if the identifier is currently defined in the preprocessor and to 0 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.

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 i×j×...×k, then **E** appearing in an expression is converted to a pointer to an (n–1)-dimensional array with rank j×...×k. 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 of the object to which the pointer points, namely five-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 when used 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 or integer division have proven 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 from left to right. The unary operators:

`*  &  -  !  ~  ++  --` **sizeof** *(type-name)*

have priority below the primary operators but higher than any binary operator. They group from right to left. Binary operators group from left to right; they have priority decreasing as indicated below.

binop:

```
*   /   %
+   -
>>  <<
<   >   <=   >=
==  !=
&
^
|
&&
||
```

The conditional operator groups from right to left.

Assignment operators all have the same priority and all group from right to left.

asgnop:
`= += -= *= /= %= >>= <<= &= ^= |=`

The comma operator has the lowest priority and groups from 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*"

CHAPTER 19
SYSTEM ASSEMBLER

Introduction

This is a reference manual for the SYSTEM V/68 resident assembler, *as*. Programmers familiar with the M68XXX family of processors should be able to program in *as* by referring to this manual, but this is not a manual for the processors. Details about the effects of instructions, the meanings of status register bits, the handling of interrupts, and many other issues are not dealt with here. This manual, therefore, should be used in conjunction with the following reference manuals:

- M68000 8-/16-/32-Bit Microprocessors Programmer's Reference Manual, Fifth Edition; Englewood Cliffs, NJ: PRENTICE-HALL, 1986. This manual is also available from the Motorola Literature Distribution Center, part number M68000UM/AD REV 4.
- MC68020 32-Bit Microprocessor User's Manual; Englewood Cliffs, NJ: PRENTICE-HALL, 1984. This manual is also available from the Motorola Literature Distribution Center, part number MC68020UM/AD REV 1.
- MC68030 Enhanced 32-Bit Microprocessor User's Manual; MOTOROLA, 1987. This manual is available from the Motorola Literature Distribution Center, part number MC68030UM/AD.
- MC68881 Floating Point Coprocessor User's Manual, MC68881UM/AD; MOTOROLA, 1985. This manual is available from the Motorola Literature Distribution Center, part number MC68881UM/AD.
- MC68851 Paged Memory Management Unit User's Manual, MC68851UM/AD; Englewood Cliffs, NJ: PRENTICE-HALL, 1986. This manual is also available from the Motorola Literature Distribution Center, part number MC68851UM/AD.
- M68000 Family Resident Structured Assembler Reference Manual, M68KMASM.
- SYSTEM V/68 User's Reference Manual, MU43810UR/D3.

For users of the SGS M68020 Cross Compilation System, references to *as*(1) and *cc*(1) should be read as *as20*(1) and *cc20*(1) if you have a MC68020 processor system or *as30*(1) and *cc30*(1) if you have a MC68030 processor system. Information about the MC68020 commands is provided in the SGS M68020 Cross Compilation System Reference Manual, M68KUNASX.

Warnings

A few important warnings to the *as* user should be emphasized at the outset. Though for the most part there is a direct correspondence between *as* notation and the notation used in the documents listed in the preceding section, several exceptions exist that could lead the unsuspecting user to write incorrect code. In addition to the exceptions described in the following paragraphs, refer also to the "Address Mode Syntax" and "Machine Instructions" sections later in this chapter for further information.

Comparison Instructions

First, the order of the operands in *compare* instructions follows one convention in the M68000 Programmer's Reference Manual and the opposite convention in *as*. Using the convention of the M68000 Programmer's Reference Manual, one might write:

```
CMP.W   D5, D3    Is D3 less than D5?
BLE     IS_LESS   Branch if less.
```

Using the *as* convention, one would write:

```
cmp.w   %d3,%d5   # Is d3 less than d5 ?
ble     is_less   # Branch if less.
```

This convention makes for straightforward reading of *compare*-and-*branch* instruction sequences, but does nonetheless lead to the peculiarity that if a *compare* instruction is replaced by a *subtract* instruction, the effect on the condition codes will be entirely different. This may be confusing to programmers who are used to thinking of a comparison as a subtraction whose result is not stored. Users of *as* who become accustomed to the convention will find that both the *compare* and *subtract* notations make sense in their respective contexts.

Overloading of Opcodes

Another issue that users must be aware of arises from the M68000 processors' use of several different instructions to do more or less the same thing. For example, the M68000 Programmer's Reference Manual lists the instructions **SUB, SUBA, SUBI**, and **SUBQ**, which all have the effect of subtracting their source operand from their destination operand. *As* provides the convenience of allowing all these operations to be specified by a single assembly instruction **sub**. Based on the operands given to the **sub** instruction, the *as* assembler selects the appropriate M68000 operation code. The danger created by this convenience is that it could leave the misleading impression that all forms of the **SUB** operation are semantically identical. In fact, they are not. The careful reader of the *M68000 Programmer's Reference Manual* will notice that whereas **SUB**, **SUBI**, and **SUBQ** all affect the condition codes in a consistent way, **SUBA** does not affect the condition codes at all. Consequently, the *as* user must be aware that when the destination of a **sub** instruction is an address register (which causes the **sub** to be mapped into the operation code for **SUBA**), the condition codes will not be affected.

Use of the Assembler

The SYSTEM V/68 command *as* invokes the assembler and has the following syntax:

as [–o *output* **]** *file*

When *as* is invoked with the **–o** *output* flag, the output of the assembly is put in the file *output*. If the **–o** flag is not specified, the output is left in a file whose name is formed by removing the **.s** suffix, if there is one, from the input filename and appending a **.o** suffix.

The M68020 cross assembler, *as20*(1), is invoked with the same syntax as *as*(1). For information about additional options for these commands, refer to the *SYSTEM V/68 Programmer's Reference Manual* for *as*(1) and the *SGS M68020 Cross Compilation System Reference Manual* for *as20*(1).

General Syntax Rules

Format of Assembly Language Line

Typical lines of *as* assembly code look like these:

```
# Clear a block of memory at location %a3

        text        2
        mov.w       &const,%d1
loop:   clr.l       (%a3)+
        dbf         %d1,loop              # go back for const
                                          # repetitions

init2:
        clr.l count;  clr.l credit;  clr.l debit;
```

These general points about the example should be noted:

— An identifier occurring at the beginning of a line and followed by a colon (:) is a *label*. One or more *labels* may precede any assembly language instruction or pseudo-operation. Refer to "Location Counters and Labels" later in this chapter.

— A line of assembly code need not include an instruction. It may consist of a comment alone (introduced by #), a label alone (terminated by :), or it may be entirely blank.

— It is good practice to use tabs to align assembly language operations and their operands into columns, but this is not a requirement of the assembler. An opcode may appear at the beginning of the line, if desired, and spaces may precede a label. A single blank or tab suffices to separate an opcode from its operands. Additional blanks and tabs are ignored by the assembler.

— It is permissible to write several instructions on one line separating them by semicolons. The semicolon is syntactically equivalent to a newline character; however, a semicolon inside a comment is ignored.

Comments

Comments are introduced by the character # and continue to the end of the line. Comments may appear anywhere and are completely disregarded by the assembler.

Identifiers

An identifier is a string of characters taken from the set **a-z, A-Z, _ , ~ , %,** and **0-9**. The first character of an identifier must be a letter (uppercase or lowercase) or an underscore. Uppercase and lowercase letters are distinguished; for example, **con35** and **CON35** are two distinct identifiers.

There is no limit on the length of an identifier. The value of an identifier is established by the **set** pseudo-operation (refer to "Symbol Definition Operations") or by using it as a label (refer to "Location Counters and Labels").

The tilde character (˜) has special significance to the assembler. A ˜ used alone, as an identifier, means "the current location." More specifically, a ˜ used in an instruction means the value of the program counter at the beginning of that instruction and a tilde used in a pseudo–instruction means the current value of the location counter for the current section. A ˜ used as the first character in an identifier becomes a period (.) in the symbol table, allowing symbols such as **.eos** and **.0fake** to be entered into the symbol table, as required by the Common Object File Format (COFF). Information about file formats is provided in the *Programmer's Reference Manual.*

Register Identifiers

A register identifier is an identifier preceded by the character **%**. It represents one of the MC68000 processor's registers.
The predefined register identifiers are:

%d0	%d4	%a0	%a4	%cc	%usp
%d1	%d5	%a1	%a5	%pc	%fp
%d2	%d6	%a2	%a6	%sp	%ccr
%d3	%d7	%a3	%a7	%sr	

Notes:

%cc and **%ccr** are equivalent.

The identifiers **%a7** and **%sp** represent the same machine register. Likewise, **%a6** and **%fp** are equivalent. Use of both **%a7** and **%sp**, or **%a6** and **%fp**, in the same program may result in confusion.

With the proper option, the assembler will correctly assemble instructions intended for the M68010. The entire register set of the MC68000 is included in the MC68010 register set. The following are new control registers for the MC68010.

REGISTERS ADDED FOR THE MC68010	
NAME	DESCRIPTION
%sfc,%sfcr	Source Function Code Register
%dfc,%dfcr	Destination Function Code Register
%vbr	Vector Base Register

Notes:

%sfc and **%sfcr** are equivalent.
%dfc and **%dfcr** are equivalent.

The entire register set of the MC68010 is included in the MC68020 register set. The following are new control registers for the MC68020:

MC68020 REGISTERS	
NAME	DESCRIPTION
%caar	Cache Address Register
%cacr	Cache Control Register
%isp	Interrupt Stack Pointer
%msp	Master Stack Pointer

The entire register set of the MC68020 is included in the MC68030 register set. The following are control registers for the MC68030:

MC68030 REGISTERS	
NAME	DESCRIPTION
%crp	Cpu Root Pointer Register
%srp	Supervisor Root Pointer Register
%tc	Translation Control Register
%tt0	Transparent Translation Register 0
%tt1	Transparent Translation Register 1
%mmusr	Memory Management Unit Status Register

Notes:

The new MC68030 registers are dedicated to memory management.

%mmusr is equivalent to the **%psr** on the MC68851.

The following are suppressed registers (zero registers) used in various MC68020 addressing modes.

MC68020 ZERO REGISTERS		
SUPPRESSED ADDRESS REGISTERS	SUPPRESSED DATA REGISTERS	SUPPRESSED PROGRAM COUNTER
%za0	%zd0	%zpc
%za1	%zd1	
%za2	%zd2	
%za3	%zd3	
%za4	%zd4	
%za5	%zd5	
%za6	%zd6	
%za7	%zd7	

Constants

as deals only with integer constants. They may be entered in decimal, octal, or hexadecimal, or they may be entered as character constants. Internally, *as* treats all constants as 32-bit binary two's complement quantities.

Numerical Constants.

A decimal constant is a string of digits beginning with a non-zero digit. An octal constant is a string of digits beginning with zero. A hexadecimal constant consists of the characters **0x** or **0X** followed by a string of characters from the set **0-9**, **a-f**, and **A-F**. In hexadecimal constants, uppercase and lowercase letters are not distinguished.

Examples:

```
set       const,35      # Decimal 35
mov.w     &035,%d1      # Octal 35 (decimal 29)
set       const, 0x35   # Hex 35 (decimal 53)
mov.w     &0xff,%d1     # Hex ff (decimal 255)
```

Character Constants.

An ordinary character constant consists of a single-quote character (') followed by an arbitrary ASCII character other than the backslash (\ ). The value of the constant is equal to the ASCII code for the character. Special meanings of characters are overridden when used in character constants; for example, if '# is used, the # is not treated as introducing a comment.

A special character constant consists of '\ followed by another character. All the special character constants and examples of ordinary character constants are listed in the following table.

CONSTANT	VALUE	MEANING
'\b	0x08	Backspace
'\t	0x09	Horizontal Tab
'\n	0x0a	Newline (Line Feed)
'\v	0x0b	Vertical Tab
'\f	0x0c	Form Feed
'\r	0x0d	Carriage Return
'\\	0x5c	Backslash
''	0x27	Single Quote
'0	0x30	Zero
'A	0x41	Uppercase A
'a	0x61	Lowercase a

Other Syntactic Details

For a discussion of expression syntax, see "Expressions" in this chapter. For information about the syntax of specific components of *as* instructions and pseudo-operations, see "Pseudo-Operations" and "Address Mode Syntax."

Segments, Location Counters, And Labels

Segments

A program in *as* assembly language may be broken into segments known as *text*, *data*, and *bss* segments. The convention regarding the use of these segments is to place instructions in *text* segments, initialized data in *data* segments, and uninitialized data in *bss* segments. However, the assembler does not enforce this convention; for example, it permits intermixing of instructions and data in a *text* segment.

Primarily to simplify compiler code generation, the assembler permits up to four separate *text* segments and four separate *data* segments named **0**, **1**, **2**, and **3**. The assembly language program may switch freely between them by using assembler pseudo-operations (refer to "Location Counter Control Operations"). When generating the object file, the assembler concatenates the *text* segments to generate a single *text* segment, and the *data* segments to generate a single *data* segment. Thus, the object file contains only one *text* segment and only one *data* segment. There is always only one *bss* segment and it maps directly into the object file.

Because the assembler keeps together everything from a given segment when generating the object file, the order in which information appears in the object file may not be the same as in the assembly language file. For example, if the data for a program consisted of:

```
data    1           # segment 1
short   0x1111
data    0           # segment 0
long    0xffffffff
data    1           # segment 1
byte    0xff
```

then equivalent object code would be generated by:

```
data    0
long    0xffffffff
short   0x1111
byte    0xff
```

Location Counters and Labels

The assembler maintains separate *location counters* for the *bss* segment and for each of the *text* and *data* segments. The location counter for a given segment is incremented by one for each byte generated in that segment.

The location counters allow values to be assigned to labels. When an identifier is used as a label in the assembly language input, the current value of the current location counter is assigned to the identifier. The assembler also keeps track of which segment the label appeared in. Thus, the identifier represents a memory location relative to the beginning of a particular segment. Any label relative to the location counter should be within the text segment.

Types

Identifiers and expressions may have values of different types.

— In the simplest case, an expression (or identifier) may have an *absolute* value, such as 29, -5000, or 262143.

— An expression (or identifier) may have a value relative to the start of a particular segment. Such a value is known as a *relocatable* value. The memory location represented by such an expression cannot be known at assembly time, but the relative values of two such expressions (i.e., the difference between them) can be known if they refer to the same segment.

Identifiers which appear as labels have *relocatable* values.

— If an identifier is never assigned a value, it is assumed to be an *undefined external*. Such identifiers may be used with the expectation that their values will be defined in another program, and therefore known at load time; but the relative values of *undefined externals* cannot be known.

Expressions

For conciseness, the following abbreviations are useful:

abs	absolute expression
rel	relocatable expression
ext	undefined external

All constants are absolute expressions. An identifier may be thought of as an expression having the identifier's type. Expressions may be built up from lesser expressions using the operators +, –, *, and /, according to the following type rules:

abs + abs = abs
abs + rel = rel + abs = rel
abs + ext = ext + abs = ext

abs – abs = abs
rel – abs = rel
ext – abs = ext
rel – rel = abs
(provided that the two relocatable expressions are relative to the same segment)

abs * abs = abs

abs / abs = abs

– abs = abs

Note that **rel – rel** expressions are permitted only within the context of a switch statement (refer to "Switch Table Operation"). Use of a **rel – rel** expression is dangerous, particularly when dealing with identifiers from *text* segments. The problem is that the assembler will determine the value of the expression before it has resolved all questions concerning span-dependent optimizations.

The unary minus operator takes the highest precedence; the next highest precedence is given to * and /, and lowest precedence is given to + and binary –. Parentheses may be used to coerce the order of evaluation.

If the result of a division is a positive non-integer, it will be truncated toward zero. If the result is a negative non-integer, [redacted] guaranteed.

Pseudo-Operations

Data Initialization Operations

byte *abs,abs,...*
One or more arguments, separated by commas, may be given. The values of the arguments are computed to produce successive bytes in the assembly output.

short *abs,abs,...*
One or more arguments, separated by commas, may be given. The values of the arguments are computed to produce successive 16-bit words in the assembly output.

long *expr,expr,...*
One or more arguments, separated by commas, may be given. Each expression may be *absolute, relocatable*, or *undefined external*. A 32-bit quantity is generated for each such argument (in the case of *relocatable* or *undefined external* expressions, the actual value may not be filled in until load time).

Alternatively, the arguments may be bit-field expressions. A bit-field expression has the form:

n : *value*

where both *n* and *value* denote *absolute* expressions. The quantity *n* represents a field width; the low-order *n* bits of *value* become the contents of the bit-field. Successive bit-fields fill up 32-bit long quantities starting with the high-order part. If the sum of the lengths of the bit-fields is less than 32 bits, the assembler creates a 32-bit long with zeros filling out the low-order bits. For example:

```
        long                4: -1, 16: 0x7f,  12:0, 5000
```

and:

```
        long                4: -1, 16: 0x7f, 5000
```

are equivalent to:

```
        long            0xf007f000, 5000
```

Bit-fields may not span pairs of 32-bit longs. Thus:

```
        long            24: 0xa, 24: 0xb, 24:0xc
```

yields the same thing as:

```
        long            0x00000a00, 0x00000b00, 0x00000c00
```

space *abs*

The value of *abs* is computed, and the resultant number of bytes of zero data is generated.
For example:

```
        space           6
```

is equivalent to:

```
        byte            0,0,0,0,0,0
```

Symbol Definition Operations

set *identifier,expr*

The value of *identifier* is set equal to *expr,* which may be absolute or relocatable.

comm *identifier,abs*

The named identifier is to be assigned to a common area of size *abs* bytes. If *identifier* is not defined by another program, the loader will allocate space for it.

lcomm *identifier,abs*

The named *identifier* is assigned to a *local common* of size *abs* bytes. This results in allocation of space in the *bss* segment.

The type of *identifier* becomes *relocatable.*

global *identifier*

This causes *identifier* to be externally visible. If *identifier* is defined in the current program, then declaring it global allows the loader to resolve references to *identifier* in other programs.

If *identifier* is not defined in the current program, the assembler expects an external resolution; in this case, therefore, *identifier* is global by default.

Location Counter Control Operations

data *abs*

The argument, if present, must evaluate to 0, 1, 2, or 3; this indicates the number of the *data* segment into which assembly is to be directed. If no argument is present, assembly is directed into *data* segment 0.

text *abs*

The argument, if present, must evaluate to 0, 1, 2, or 3; this indicates the number of the *text* segment into which assembly is to be directed. If no argument is present, assembly is directed into *text* segment 0.

Before the first **text** or **data** operation is encountered, assembly is directed by default into *text* segment 0.

org *expr*

The current location counter is set to *expr*. *Expr* must represent a value in the current segment, and must not be less than the current location counter.

even

The current location counter is rounded up to the next even value.

Symbolic Debugging Operations

The assembler allows for symbolic debugging information to be placed into the object code file with special pseudo-operations. The information typically includes line numbers and information about C language symbols, such as their type and storage class. The C compiler (*cc*(1)) generates symbolic debugging information when the **-g** option is used. Assembler programmers may also include such information in source files.

file and ln

The **file** pseudo-operation passes the name of the source file into the object file symbol table. It has the form:

file *filename*

where *filename* consists of one to 14 characters enclosed in quotation marks.

The **ln** pseudo-operation makes a line number table entry in the object file. That is, it associates a line number with a memory location. Usually the memory location is the current location in text. The format is:

ln *line*[,*value*]

where *line* is the line number. The optional value is the address in *text, data,* or *bss* to associate with the line number. The default when *value* is omitted (which is usually the case) is the current location in *text*.

Symbol Attribute Operations.

The basic symbolic testing pseudo-operations are **def** and **endef**. These operations enclose other pseudo-operations that assign attributes to a symbol and must be paired.

```
def      name
.                  # Attribute
.                  # Assigning
.                  # Operations
endef
```

NOTES

- **def** does not define the symbol, although it does create a symbol table entry. Because an undefined symbol is treated as external, a symbol which appears in a **def**, but which never acquires a value, will ultimately result in an error at link edit time.
- To allow the assembler to calculate the sizes of functions for other tools, each **def/endef** pair that defines a function name must be matched by a **def/endef** pair after the function in which a storage class of **–1** is assigned.

The paragraphs below describe the attribute-assigning operations. Keep in mind that all these operations apply to symbol *name* which appeared in the opening **def** pseudo-operation.

val *expr*
Assigns the value *expr* to *name*. The type of the expression *expr* determines with which section *name* is associated. If the value is ˜ , the current location in the *text* section is used.

scl *expr*
Declares a storage class for *name*. The expression *expr* must yield an **ABSOLUTE** value that corresponds to the C compiler's internal representation of a storage class. The special value **–1** designates the physical end of a function.

type *expr*
Declares the C language type of *name*. The expression *expr* must yield an **ABSOLUTE** value that corresponds to the C compiler's internal representation of a basic or derived type.

tag *str*
Associates *name* with the structure, enumeration, or union named *str* which must have already been declared with a **def/endef** pair.

line *expr*
Provides the line number of *name*, where *name* is a block symbol. The expression *expr* should yield an **ABSOLUTE** value that represents a line number.

size *expr*
Gives a size for *name*. The expression *expr* must yield an **ABSOLUTE** value. When *name* is a structure or an array with a predetermined extent, *expr* gives the size in bytes. For bit fields, the size is in bits.

dim *expr1,expr2,...*
Indicates that *name* is an array. Each of the expressions must yield an **ABSOLUTE** value that provides the corresponding array dimension.

Switch Table Operation

The C compiler generates a compact set of instructions for the C language *switch* construct. An example is shown below.

```
        sub.l   &1,%d0
        cmp.l   %d0,&4
        bhi     L%21
        mov.w   (%d0.w*2,L%22),%d0
        jmp     (%d0.w,L%22)
        swbeg   &5
L%22:
        short   L%15-L%22
        short   L%21-L%22
        short   L%16-L%22
        short   L%21-L%22
        short   L%17-L%22
```

The special **swbeg** pseudo-operation communicates to the assembler that the lines following it contain **rel-rel** subtractions. Remember that ordinarily such subtractions are risky because of span-dependent optimization. In this case, however, the assembler makes special allowances for the subtraction because the compiler guarantees that both symbols will be defined in the current assembler file, and that one of the symbols is a fixed distance away from the current location.

The **swbeg** pseudo-operation takes an argument that looks like an immediate operand. The argument is the number of lines that follow **swbeg** and that contain switch table entries. **Swbeg** inserts two words into text. The first is the **ILLEGAL** instruction code. The second is the number of table entries that follow. The disassembler *dis*(1) needs the **ILLEGAL** instruction as a hint that what follows is a switch table. Otherwise, it would get confused when it tried to decode the table entries, differences between two symbols, as instructions.

Span-Dependent Optimization

The assembler makes certain choices about the object code it generates based on the distance between an instruction and its operand(s). Choosing the smallest, fastest form is called span-dependent optimization. Span-dependent optimization occurs most obviously in the choice of object code for branches and jumps. It also occurs when an operand may be represented by the program counter relative address mode instead of as an absolute 2-word (**long**) address. The span-dependent optimization capability is normally enabled; the **–n** command line flag disables it. When this capability is disabled, the assembler makes worst-case assumptions about the types of object code that must be generated. Span-dependent optimizations are performed only within **text** segment 0. Any reference outside **text** segment 0 is assumed to be worst-case.

The C compiler (*cc*(1)) generates branch instructions without a specific offset size. When the optimizer is used, it identifies branches which could be represented by the short form, and it changes the operation accordingly. The assembler chooses only between word (16 bits) and long–word (32 bits) representations for branches.

For the MC68000 and MC68010 processors, branch instructions, e.g., **bra, bsr,** or **bgt,** can have either a byte or a word pc-relative address operand. A byte or word size specification should be used only when the user is sure that the address intended can be represented in the byte or word allowed. The assembler will take one of these instructions with a size specification and generate the byte or word form of the instruction without asking questions.

Although the largest offset specification allowed for the MC68000 and MC68010 processors is a word, large programs could conceivably have need for a branch to location not reachable by a word displacement. Therefore, equivalent long–word forms of these instructions might be needed. When the assembler encounters a branch instruction without a size specification, it tries to choose between the word and long–word forms of the instruction. If the operand can be represented in a word, then the word form of the instruction will be generated. Otherwise, the long–word form will be generated. For unconditional branches, e.g., **br, bra,** and

bsr, the long-word form is just the equivalent jump (**jmp** and **jsr**) with an absolute address operand (instead of pc-relative). For conditional branches, the equivalent long-word form is a conditional branch around a jump, where the conditional test has been reversed.

The following table summarizes span-dependent optimizations. The optimizer chooses only between the word form and long–word forms for branches (but not **bsr**).

Assembler Span-Dependent Optimizations

Instruction	Word Form	Long-word Form
br, bra, bsr	word offset	**jmp** or **jsr** with absolute long address
conditional branch	word offset	short conditional branch with reversed condition around **jmp** with absolute long address

For the MC68020 and MC68030 processors, branch instructions can have either a byte, word, or long-word pc-relative address operand.

Address Mode Syntax

The following table summarizes the *as* syntax for MC68000, MC68010, MC68020 and MC68030 addressing modes. Addressing modes for the MC68020 and MC68030 are shown with "MC68020 Only" in parentheses beneath the MC68000 notation; modes not specified in this way are for all four processors.

In the table, the following abbreviations are used:

an	Address register, where *n* is any digit from 0 through 7.
dn	Data register, where *n* is any digit from 0 through 7.
ri	Index register *i* may be any address (%an) or data register (%dn) with an optional size designation (i.e., **ri.w** for 16 bits or **ri.l** for 32 bits); default size is **.w**.
scl	Optional scale factor that may be multiplied times index register in some modes. Values for *scl* are 1, 2, 4, or 8; default is 1. Only MC68020 and MC68030 instructions can have scale factors.
bd	Two's complement base displacement that is added before indirection takes place; size can be 16 or 32 bits. Only MC68020 and MC68030 instructions can have scale factors.

od	Outer displacement that is added as part of effective address calculation after memory indirection; size can be 16 or 32 bits. Only MC68020 and MC68030 instructions can have scale factors.
d	Two's complement or sign-extended displacement that is added as part of effective address calculation; size may be 8 or 16 bits; when omitted, assembler uses value of zero.
pc	Program counter
[ ]	Grouping characters used to enclose an indirect expression; required characters. Addressing arguments can occur in any order within the brackets.
( )	Grouping characters used to enclose an entire effective address; required characters. Addressing arguments can occur in any order within the parentheses.
{ }	Indicate that a scale factor is optional; not required characters.

It is important to note that expressions used for the **absolute** addressing modes need not be *absolute expressions* in the sense described earlier under "Types." Although the addresses used in those addressing modes must ultimately be filled in with constants, that can be done later by the loader. There is no need for the assembler to be able to compute them. Indeed, the **Absolute Long** addressing mode is commonly used for accessing *undefined external* addresses.

Effective Address Modes

M68000 FAMILY NOTATION	*as* NOTATION	EFFECTIVE ADDRESS MODE
Dn	%dn	Data Register Direct
An	%an	Address Register Direct
(An)	(%an)	Address Register Indirect
(An)+	(%an)+	Address Register Indirect With Postincrement
–(An)	–(%an)	Address Register Indirect With Predecrement
d(An)	d(%an)	Address Register Indirect With Displacement (*d* signifies a signed 16-bit absolute displacement)

d(An,Ri)	d(%an,%ri.w)
	d(%an,%ri.l)
		Address Register Indirect With Index Plus Displacement (*d* signifies a signed 8-bit absolute displacement)

(bd,An,Ri{*scl}) (MC68020/MC68030 Only)	(bd,%an,%ri{*scl})	Address Register Direct With Index Plus Base Displacement
([bd,An,Ri{*scl}],od) (MC68020/MC68030 Only)	([bd,%an,%ri{*scl}],od)	Memory Indirect With Preindexing Plus Base and Outer Displacement
([bd,An],Ri{*scl},od) (MC68020/MC68030 Only)	([bd,%an],%ri{*scl},od)	Memory Indirect With Postindexing Plus Base and Outer Displacement
d(PC)	d(%pc)	Program Counter Indirect With Displacement (d signifies 16-bit displacement)

d(PC,Ri)	d(%pc,%rn.l)
	d(%pc,%rn.w)
		Program Counter Direct With Index and Displacement (d signifies 8-bit displacement)

(bd,PC,Ri{*scl}) (MC68020/MC68030 Only)	(bd,%pc,%ri{*scl})	Program Counter Direct With Index and Base Displacement
([bd,PC],Ri{*scl},od) (MC68020/MC68030 Only)	([bd,%pc],%ri{*scl},od)	Program Counter Memory Indirect With Post-indexing Plus Base and Outer Displacement

M68000 FAMILY NOTATION	*as* NOTATION	EFFECTIVE ADDRESS MODE
([bd,PC,Ri{*scl}],od) (MC68020/MC68030 Only)	([bd,%pc,%ri{*scl}],od)	Program Counter Memory Indirect With Preindexing Plus Base and Outer Displacement
xxx.W	xxx	Absolute Short Address (*xxx* signifies an expression yielding a 16-bit memory address)
xxx.L	xxx	Absolute Long Address (*xxx* signifies an expression yielding a 32-bit memory address)
#xxx	&xxx	Immediate Data (*xxx* signifies an absolute constant expression)

In the table above, the index register notation should be understood as ri.size*scale, where both size and scale are optional. Refer to Chapter 2 of the *M68000 Family Resident Structured Assembler Reference Manual* for additional information about effective address modes. Section 2 of the *MC68020 32-Bit Microprocessor User's Manual* also provides information about generating effective addresses and assembler syntax.

Note that suppressed address register **%zan** can be used in place of **%an**, suppressed PC register **%zpc** can be used in place of **%pc**, and suppressed data register **%zdn** can be used in place of **%dn**, if suppression is desired.

The address modes for the MC68020 and MC68030 use two different formats of extension. The brief format provides fast indexed addressing, while the full format provides a number of options in size of displacement and indirection. The assembler will generate the brief format if the effective address expression is not memory indirect, value of displacement is within a byte, and no base or index suppression is specified; otherwise, the assembler will generate the full format.

Some source code variations of the new modes may be redundant with the MC68000 address register indirect, address register indirect with displacement, and program counter with displacement modes. The assembler will select the more efficient mode when redundancy occurs. For example, when the assembler sees the form **(An)**, it will generate address register indirect mode (mode 2).

The assembler will generate address register indirect with displacement (mode 5) when seeing any of the following forms (as long as bd fits in 16 bits or less):

```
bd(An)
(bd,An)
(An,bd)
```

Machine Instructions

Instructions For The MC68000/MC68010/MC68020/MC68030

The following table shows how MC68000/MC68010/MC68020/MC68030 instructions should be written in order to be understood correctly by the *as* assembler. The entire instruction set for the MC68030 can be used. Instructions that are MC68010/MC68020/MC68030-only, MC68020-only or MC68020/MC68030-only are noted as such in the "OPERATION" column. Additional MC68030-only instructions which deal specifically with memory management are listed separately as a subset of the MC68851 instructions.

Several abbreviations are used in the table:

S The letter **S**, as in **add.S**, stands for one of the operation size attribute letters **b**, **w**, or **l**, representing a byte, word, or long operation.

A The letter **A**, as in **add.A**, stands for one of the address operation size attribute letters **w** or **l**, representing a word or long operation.

CC In the contexts **bCC**, **dbCC**, and **sCC**, the letters **CC** represent any of the following condition code designations (except that **f** and **t** may not be used in the **bCC** instruction):

cc	carry clear	ls	low or same
cs	carry set	lt	less than
eq	equal	mi	minus
f	false	ne	not equal
ge	greater or equal	pl	plus
gt	greater than	t	true
hi	high	vc	overflow clear
hs	high or same (=cc)	vs	overflow set
le	less or equal		
lo	low (=cs)		

EA This represents an arbitrary effective address.

I An absolute expression, used as an immediate operand.

Q An absolute expression evaluating to a number from 1 to 8.

L A label reference, or any expression representing a memory address in the current segment.

d Two's complement or sign-extended displacement that is added as part of effective address calculation; size may be 8 or 16 bits; when omitted, assembler uses value of zero.

%dx, %dy, %dn Represent data registers.

%ax, %ay, %an Represent address registers.

%rx, %ry, %rn Represent either data or address registers.

%rc Represents control register (**%sfc**, **%dfc**, **%cacr**, **%vbr**, **%caar**, **%msp**, **%isp**).

offset Either an immediate operand or a data register.

width Either an immediate operand or a data register.

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
ABCD	**abcd.b**	**%dy,%dx** **–(%ay),–(%ax)**	Add Decimal with Extend
ADD	**add.S**	**EA,%dn** **%dn,EA**	Add Binary
ADDA	**add.A** **adda.A**	**EA,%an** **EA,%an**	Add Address. Second form is PMMU-supported *as20* only.
ADDI	**add.S** **addi.S**	**&I,EA** **&I,EA**	Add Immediate. Second form is PMMU-supported *as20* only.
ADDQ	**add.S** **addq.S**	**&Q,EA** **&Q,EA**	Add Quick. Second form is PMMU-supported *as20* only.
ADDX	**addx.S**	**%dy,%dx** **–(%ay),–(%ax)**	Add Extended
AND	**and.S**	**EA,%dn** **%dn,EA**	AND Logical
ANDI	**and.S** **andi.S**	**&I,EA** **&I,EA**	AND Immediate Second form is PMMU-supported *as20* only.
ANDI to CCR	**and.b**	**&I,%cc**	AND Immediate to Condition Codes
ANDI to SR	**and.w**	**&I,%sr**	AND Immediate to the Status Register
ASL	**asl.S**	**%dx,%dy** **&Q,%dy**	Arithmetic Shift (Left)
	asl.w	**&1,EA**	
ASR	**asr.S**	**%dx,%dy** **&Q,%dy**	Arithmetic Shift (Right)
	asr.w	**&1,EA**	

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
Bcc	**bCC**	**L**	Branch Conditionally (16-bit Displacement)
	bCC.b	**L**	Branch Conditionally (Short) (8-bit Displacement)
	bCC.l	**L**	Branch Conditionally (Long) (32-bit Displacement) (MC68020/MC68030 Only)
BCHG	**bchg**	**%dn,EA** **&I,EA**	Test a Bit and Change Note: **bchg** should be written with no suffix. If the second operand is a data register, **.l** is assumed; otherwise, **.b** is.
BCLR	**bclr**	**%dn,EA** **&I,EA**	Test a Bit and Clear Note: **bclr** should be written with no suffix. If the second operand is a data register, **.l** is assumed; otherwise, **.b** is.
BFCHG	**bfchg**	**EA{offset:width}**	Complement Bit Field (MC68020/MC68030 Only)
BFCLR	**bfclr**	**EA{offset:width}**	Clear Bit Field (MC68020/MC68030 Only)
BFEXTS	**bfexts**	**EA{offset:width},%dn**	Extract Bit Field (Signed) (MC68020/MC68030 Only)
BFEXTU	**bfextu**	**EA{offset:width},%dn**	Extract Bit Field (Unsigned) (MC68020/MC68030 Only)
BFFFO	**bfffo**	**EA{offset:width},%dn**	Find First One in Bit Field (MC68020/MC68030 Only)
BFINS	**bfins**	**%dn,EA{offset:width}**	Insert Bit Field (MC68020/MC68030 Only)
BFSET	**bfset**	**EA{offset:width}**	Set Bit Field (MC68020/MC68030 Only)

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
BFTST	**bftst**	**EA{offset:width}**	Test Bit Field (MC68020/MC68030 Only)
BKPT	**bkpt**	**&I**	Breakpoint (MC68020/MC68030 Only)
BRA	**bra**	**L**	Branch Always (16-bit Displacement)
	bra.b	**L**	Branch Always (Short) (8-bit Displacement)
	br.l	**L**	Branch Always (Long) (32-bit Displacement) (MC68020/MC68030 Only)
	br	**L**	Same as **bra**
	br.b	**L**	Same as **bra.b**
BSET	**bset**	**%dn,EA** **&I,EA**	Test a Bit and Set Note: **bset** should be written with no suffix. If the second operand is a data register, **.l** is assumed; otherwise, **.b** is.
BSR	**bsr**	**L**	Branch to Subroutine (16-bit Displacement)
	bsr.b	**L**	Branch to Subroutine (Short) (8-bit Displacement)
	bsr.l	**L**	Branch to Subroutine (Long) (32-bit Displacement) (MC68020/MC68030 Only)
BTST	**btst**	**%dn,EA** **&I,EA**	Test a Bit and Set Note: **btst** should be written with no suffix. If the second operand is a data register, **.l** is assumed; otherwise, **.b** is.
CALLM	**callm**	**&I,EA**	Call Module (MC68020 Only)

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
CAS	**cas**	**%dx,%dy,EA**	Compare and Swap Operands (MC68020/MC68030 Only)
CAS2	**cas2**	**%dx:%dy,%dx:%dy,%rx:%ry**	Compare and Swap Dual Operands (MC68020/MC68030 Only)
CHK	**chk.w**	**EA,%dn**	Check Register Against Bounds
	chk.l	**EA,%dn**	Check Register Against Bounds (Long) (MC68020/MC68030 Only)
CHK2	**chk2.S**	**EA,%rn**	Check Register Against Bounds (MC68020/MC68030 Only)
CLR	**clr.S**	**EA**	Clear an Operand
CMP	**cmp.S**	**%dn,EA**	Compare
CMPA	**cmp.A** **cmpa.A**	**%an,EA** **%an,EA**	Compare Address. Second form is PMMU-supported *as20* only.
CMPI	**cmp.S** **cmpi.S**	**EA,&I** **EA,&I**	Compare Immediate. Second form is PMMU-supported *as20* only.
CMPM	**cmp.S** **cmpm.S**	**(%ax)+,(%ay)+** **(%ax)+,(%ay)+**	Compare Memory. Second form is PMMU-supported *as20* only.
CMP2	**cmp.S** **cmp2.A**	**%rn,EA** **%rn,EA**	Compare Register Against Bounds (MC68020/MC68030 Only).[1] Second form is PMMU-supported *as20* only.
DBcc	**dbCC**	**%dn,L**	Test Condition, Decrement, and Branch
	dbra	**%dn,L**	Decrement and Branch Always
	dbr	**%dn,L**	Same as **dbra**

1. Note: The order of operands in *as* is the reverse of that in the *M68000 Programmer's Reference Manual*.

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
DIVS	**divs.w**	**EA,%dx**	Signed Divide 32/16–> 32
	tdivs.l **divs.l**	**EA,%dx** **EA,%dx**	Signed Divide (Long) 32/32 –> 32 (MC68020/MC68030 Only)
	tdivs.l **divsl.l**	**EA,%dx:%dy** **EA,%dx:%dy**	Signed Divide (Long) 32/32 –> 32r:32q[2] (MC68020/MC68030 Only). Second form is PMMU-supported *as20* only.
	divs.l	**EA,%dx:%dy**	Signed Divide (Long) 64/32 –> 32r:32q[3] (MC68020/MC68030 Only)
DIVU	**divu.w**	**EA,%dn**	Unsigned Divide 32/16 –> 32
	tdivu.l **divu.l**	**EA,%dx** **EA,%dx**	Unsigned Divide (Long) 32/32 –> 32 (MC68020/MC68030 Only)
	tdivu.l **divul.l**	**EA,%dx:%dy** **EA,%dx:%dy**	Unsigned Divide (Long) 32/32 –> 32r:32q (MC68020/MC68030 Only)[4] Second form is PMMU-supported *as20* only.
	divu.l	**EA,%dx:%dy**	Unsigned Divide (Long) 64/32 –> 32r:32q (MC68020/MC68030 Only)[5]

2. Whenever %dx and %dy are the same register, the form is equivalent to the tdivs.l EA,%dx form (PMMU-supported *as20* only).
3. Whenever %dx and %dy are the same register, the form is equivalent to the divs.l EA,%dx form (PMMU-supported *as20* only).
4. Whenever %dx and %dy are the same register, then the form is equivalent to the tdivu.l EA,%dx form.
5. Whenever %dx and %dy are the same register, then the form is equivalent to the divu.l EA,%dx form.

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
EOR	**eor.S**	**%dn,EA**	Exclusive OR Logical
EORI	**eor.S** **eori.S**	**&I,EA** **&I,EA**	Exclusive OR Immediate. Second form is PMMU-supported *as20* only.
EORI to CCR	**eor.b** **eori.b** **eori.b**	**&I,%cc** **&I,%cc** **&I,%ccr**	Exclusive OR Immediate to Condition Code Register. Second and third forms PMMU-supported *as20* only.
EORI to SR	**eor.w** **eori.w**	**&I,%sr** **&I,%sr**	Exclusive OR Immediate to the Status Register. Second form is PMMU-supported *as20* only.
EXG	**exg**	**%rx,%ry**	Exchange Registers
EXT	**ext.w**	**%dn**	Sign-Extend Low-Order Byte of Data to Word
	ext.l	**%dn**	Sign-Extend Low-Order Word of Data to Long
	extb.l	**%dn**	Sign-Extend Low-Order Byte of Data to Long (MC68020/MC68030 Only)
	extw.l	**%dn**	Same as **ext.l** (MC68020/MC68030 Only)
JMP	**jmp**	**EA**	Jump
JSR	**jsr**	**EA**	Jump to Subroutine
LEA	**lea.l**	**EA,%an**	Load Effective Address
LINK	**link**	**%an,&I**	Link and Allocate

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
LSL	**lsl.S**	**%dx,%dy** **&Q,%dy**	Logical Shift (Left)
	lsl.w	**&I,EA**	
LSR	**lsr.S**	**%dx,%dy** **&Q,&dy**	Logical Shift (Right)
	lsr.w	**&I,EA**	
MOVE	**mov.S** **move.S**	**EA,EA** **EA,EA**	Move Data from Source to Destination. move.S form is PMMU-supported *as20* only. Note: If the destination is an address register, the instruction generated is **MOVEA.**
MOVE to CCR	**mov.w** **move.w**	**EA,%cc** **EA,%ccr**	Move to Condition Codes. move.w form is PMMU-supported *as20* only.
MOVE from CCR	**mov.w** **move.w**	**%cc,EA** **%ccr,EA**	Move from Condition Codes. (MC68010/MC68020/MC68030 Only) move.w form is PMMU-supported *as20* only.
MOVE to SR	**mov.w** **move.w**	**EA,%sr** **EA,%sr**	Move to the Status Register. move.w form is PMMU-supported *as20* only.
MOVE from SR	**mov.w** **move.w**	**%sr,EA** **%sr,EA**	Move from the Status Register. move.w form is PMMU-supported *as20* only.
MOVE USP	**mov.l** **move.l**	**%usp,%an** **%an,%usp** **%usp,%an** **%an,%usp**	Move User Stack Pointer. move.l form is PMMU-supported *as20* only.

MC68000 INSTRUCTION FORMATS

MNEMONIC	ASSEMBLER SYNTAX		OPERATION
MOVEA	**mov.A**	**EA,%an**	Move Address.
	mova.A	**EA,%an**	movea.A and movea.A PMMU-
	movea.A	**EA,%an**	supported *as20* only.
MOVEC to CR	**mov.l**	**&I,EA**	Move to Control Register.
	movc.l	**%rn,%rc**	(MC68010/MC68020/MC68030 Only)
	movec.l	**%rn,%rc**	movc.l and movec.l PMMU-
		%rn,%rc	supported *as20* only.
MOVEC from CR	**mov.l**	**%rc,%rn**	Move from Control Register.
	movc.l	**%rc,%rn**	(MC68010/MC68020/MC68030 Only)
	movec.l	**%rc,%rn**	movc.l and movec.l PMMU- supported *as20* only.
MOVEM	**movm.A**	**EA,&I**	Move Multiple Registers[6] (See footnote).
	movem.A	**&I,EA**	movem.A form is PMMU-
		EA,&I	supported *as20* only.
MOVEP	**movp.A**	**%dx,d(%ay)**	Move Peripheral Data.
		d(%ay),%dx	movep.A form is PMMU-
	movep.A	**%dx,d(%ay)**	supported *as20* only.
		d(%ay),%dx	
MOVEQ	**mov.l**	**&I,%dn**	Move Quick. movq.l and moveq.l forms
	movq.l	**&I,%dn**	PMMU-supported *as20* only.
	moveq.l	**&I,%dn**	
MOVES	**movs.S**	**%rn,EA**	Move to/from Address Space
	movs.S	**EA,%rn**	(MC68010/MC68020/MC68030 Only).
	moves.S	**%rn,EA**	moves.S forms PMMU-
	moves.S	**EA,%rn**	supported *as20* only.

6. The immediate operand is a mask designating which registers are to be moved to memory or which are to receive memory data. Not all addressing modes are permitted, and the correspondence between mask bits and register numbers depends on the addressing mode. Unlike the other M68000 family of assemblers, only a mask is allowed for the *as* assembler (PMMU-supported *as20* only).

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
MULS	muls.w	EA,%dx	Signed Multiply 16*16 –> 32
	tmuls.l muls.l	EA,%dx EA,%dx	Signed Multiply (Long) 32*32 –> 32 (MC68020/MC68030 Only)
	muls.l	EA,%dx:%dy	Signed Multiply (Long) 32*32 –> 64 (MC68020/MC68030 Only)[7]
MULU	mulu.w	EA,%dx	Unsigned Multiply 16*16 –> 32
	tmulu.l mulu.l	EA,%dx EA,%dx	Unsigned Multiply (Long) 32*32 –> 32 (MC68020/MC68030 Only)
	mulu.l	EA,%dx:%dy	Unsigned Multiply (Long) 32*32 –> 64 (MC68020/MC68030 Only)[8]
NBCD	nbcd.b	EA	Negate Decimal with Extend
NEG	neg.S	EA	Negate
NEGX	negx.S	EA	Negate with Extend
NOP	nop		No Operation
NOT	not.S	EA	Logical Complement
OR	or.S	EA,%dn %dn,EA	Inclusive OR Logical
ORI	or.S ori.S	&I,EA &I,EA	Inclusive OR Immediate. ori.S form is PMMU-supported *as20* only..
ORI to CCR	or.b ori.b ori.b	&I,%cc &I,%cc &I,%ccr	Inclusive OR Immediate to Condition Codes. ori.b forms are PMMU-supported *as20* only.

7. Whenever %dx and %dy are the same register, the form is equivalent to the muls.l EA,%dx form (PMMU-supported *as20* only).

8. Whenever %dx and %dy are the same register, the form is equivalent to the mulu.l DA,%dx form (PMMU-supported *as20* only).

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
ORI to SR	or.w ori.w	&I,%sr &I,%sr	Inclusive OR Immediate to the Status Register. ori.w form is PMMU-supported *as20* only.
PACK	pack pack	–(%ax),–(%ay),&I %dx,%dy,&I	Pack BCD (MC68020/MC68030 Only)
PEA	pea.l	EA	Push Effective Address
RESET	reset		Reset External Devices
ROL	rol.S rol.w	%dx,%dy &Q,%dy &I,EA	Rotate (without Extend) (Left)
ROR	ror.S ror.w	%dx,%dy &Q,%dy &I,EA	Rotate (without Extend) (Right)
ROXL	roxl.S roxl.w	%dx,%dy &Q,%dy &I,EA	Rotate with Extend (Left)
ROXR	roxr.S roxr.w	%dx,%dy &Q,%dy &I,EA	Rotate with Extend (Right)
RTD	rtd	&I	Return and Deallocate Parameters (MC68010/MC68020/MC68030 Only)
RTE	rte		Return from Exception
RTM	rtm	%rn	Return from Module (MC68020 Only)
RTR	rtr		Return and Restore Condition Codes
RTS	rts		Return from Subroutine

MC68000 INSTRUCTION FORMATS

MNEMONIC	ASSEMBLER SYNTAX		OPERATION
SBCD	**sbcd.b**	**%dy,%dx** **–(%ay),–(%ax)**	Subtract Decimal with Extend
Scc	**sCC.b**	**EA**	Set According to Condition
STOP	**stop**	**&I**	Load Status Register and Stop
SUB	**sub.S**	**EA,%dn** **%dn,EA**	Subtract Binary
SUBA	**sub.A** **suba.A**	**EA,%an** **EA,%an**	Subtract Address. suba.A form is PMMU-supported *as20* only.
SUBI	**sub.S** **subi.S**	**&I,EA** **&I,EA**	Subtract Immediate. subi.S form is PMMU-supported *as20* only.
SUBQ	**sub.S** **subq.S**	**&Q,EA** **&Q,EA**	Subtract Quick. subq.S form is PMMU-supported *as20* only.
SUBX	**subx.S**	**%dy,%dx** **–(%ay),–(%ax)**	Subtract with Extend
SWAP	**swap.w**	**%dn**	Swap Register Halves
TAS	**tas.b**	**EA**	Test and Set an Operand
TRAP	**trap**	**&I**	Trap
TRAPV	**trapv**		Trap on Overflow
TRAPcc	**tCC trapCC** **tpCC.A** **trapCC.A**	 **&I** **&I**	Trap on Condition (MC68020/MC68030 Only)
TST	**tst.S**	**EA**	Test an Operand
UNLK	**unlk**	**%an**	Unlink
UNPK	**unpk**	**–(%ax),–(%ay),&I** **%dx,%dy,&I**	Unpack BCD (MC68020/MC68030 Only)

Instructions For The MC68881

The following table shows how the floating point co-processor (MC68881) instructions should be written to be understood by the *as* assembler.

In the table, *cc* represents any of the following floating point condition code designations:

	TRAP ON UNORDERED
cc	MEANING
ge	greater than or equal
gl	greater or less than
gle	greater or less than or equal
gt	greater than
le	less than or equal
lt	less than
ngt	not greater than
nge	not (greater than or equal)
nlt	not less than
ngl	not (greater or less than)
nle	not (less than or equal)
ngle	not (greater or less than or equal)
sneq	signaling not equal
sf	signaling false
seq	signaling equal
st	signaling true

NO TRAP ON UNORDERED	
cc	MEANING
eq	equal
oge	ordered greater than or equal
ogl	ordered greater or less than
ogt	ordered greater than
ole	ordered less than or equal
olt	ordered less than
or	ordered
t	true
ule	unordered or less or equal
ult	unordered or less than
uge	unordered or greater than or equal
ueq	unordered or equal
ugt	unordered or greater than
un	unordered
neq	not equal
f	false

The designation *ccc* represents a group of constants in MC68881 constant ROM which have the following values:

ccc	VALUE	*ccc*	VALUE
00	pi	35	10**4
0B	log10(2)	36	10**8
0C	e	37	10**16
0D	log2(e)	38	10**32
0D	log10(e)	39	10**64
0F	0.0	3A	10**128
10	ln(2)	3B	10**256
11	ln(10)	3C	10**512
32	10**0	3D	10**1024
33	10**1	3E	10**2048
34	10**2	3F	10**4096

Additional abbreviations used in the table are:

EA	represents an effective addresss
L	a label reference or any expression representing a memory address in the current segment
I	represents an absolute expression, used as an immediate operand
%dn	represents data register
%fpm,%fpn,%fpq	represent floating point data registers
%control	represents floating point control register
%fpcr	represents floating point control register (PMMU-supported *as20* only)
%status	represents floating point status register
%fpsr	represents floating point status register (PMMU-supported *as20* only)
%iaddr	represents floating point instruction address register
%fpiar	represents floating point instruction address register (PMMU-supported *as20* only)
SF	represents source format letters:

b	byte integer
w	word integer
l	long word integer
s	single precision
d	double precision
x	extended precision
p	packed binary code decimal

A	represents source format letters w or l
B	represents source format letters b, w, l, s, or p

Note: The source format must be specified if more than one source format is permitted or a default source format *x* is assumed. Source format need not be specified if only one format is permitted by the operation.

MC68881 INSTRUCTION FORMATS

MNEMONIC	ASSEMBLER SYNTAX		OPERATION
FABS	fabs.SF	EA,%fpn	absolute value function
	fabs.x	%fpm,%fpn	
	fabs.x	%fpn	
FACOS	facos.SF	EA,%fpn	arccosine function
	facos.x	%fpm,%fpn	
	facos.x	%fpn	
FADD	fadd.SF	EA,%fpn	floating point add
	fadd.x	%fpm,%fpn	
ftst form.sp			
FASIN	fasin.SF	EA,%fpn	arcsine function
	fasin.x	%fpm,%fpn	
	fasin.x	%fpn	
FATAN	fatan.SF	EA,%fpn	arctangent function
	fatan.x	%fpm,%fpn	
	fatan.x	%fpn	
FATANH	fatanh.SF	EA,%fpn	hyperbolic arctangent function
	fatanh.x	%fpm,%fpn	
	fatanh.x	%fpn	
FBcc	fbcc.A	L	co-processor branch conditionally
FCMP	fcmp.SF	%fpn,EA	floating point compare
	fcmp.x	%fpn,%fpm	
FCOS	fcos.SF	EA,%fpn	cosine function
	fcos.x	%fpm,%fpn	
	fcos.x	%fpn	
FCOSH	fcosh.SF	EA,%fpn	hyperbolic cosine function
	fcosh.x	%fpm,%fpn	
	fcosh.x	%fpn	
FDBcc	fdbcc.w	%dn,L	decrement and branch on condition
FDIV	fdiv.SF	EA,%fpn	floating point divide
	fdiv.x	%fpm,%fpn	

MC68881 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
FETOX	fetox.SF fetox.x fetox.x	EA,%fpn %fpm,%fpn %fpn	e**x function
FETOXM1	fetoxm1.SF fetoxm1.x fetoxm1.x	EA,%fpn %fpm,%fpn %fpn	e**x(x-1) function
FGETEXP	fgetexp.SF fgetexp.x fgetexp.x	EA,%fpn %fpm,%fpn %fpn	get the exponent function
FGETMAN	fgetman.SF fgetman.x fgetman.x	EA,%fpn %fpm,%fpn %fpn	get the mantissa function
FINT	fint.SF fint.x fint.x	EA,%fpn %fpm,%fpn %fpn	integer part function
FINTRZ	fintrz.SF fintrz.x fintrz.x	EA,%fpn %fpm,%fpn %fpn	integer part, round-to-zero function
FLOG2	flog2.SF flog2.x flog2.x	EA,%fpn %fpm,%fpn %fpn	binary log function
FLOG10	flog10.SF flog10.x flog10.x	EA,%fpn %fpm,%fpn %fpn	common log function

MC68881 INSTRUCTION FORMATS

MNEMONIC	ASSEMBLER SYNTAX		OPERATION
FLOGN	**flogn.SF**	**EA,%fpn**	natural log function
	flogn.x	**%fpm,%fpn**	
	flogn.x	**%fpn**	
FLOGNP1	**flognp1.SF**	**EA,%fpn**	natural log (x+1) function
	flognp1.x	**%fpm,%fpn**	
	flognp1.x	**%fpn**	
FMOD	**fmod.SF**	**EA,%fpn**	floating point module
	fmod.x	**%fpm,%fpn**	
FMOVE	**fmov.SF**	**EA,%fpn**	move to floating point register (fmove.SF and fmove.x forms PMMU-supported *as20* only)
	fmov.x	**%fpm,%fpn**	
	fmove.SF	**EA,%fpn**	
	fmove.x	**%fpm,%fpn**	
	fmov.SF	**%fpn,EA**	move from floating point register to memory (fmove.SF and fmove.p forms PMMU-supported *as20* only)
	fmov.p	**%fpn,EA{&l}**	
	fmov.p	**%fpn,EA{%dn}**	
	fmove.SF	**%fpn,EA**	
	fmove.p	**%fpn,EA{&l}**	
	fmove.p	**%fpn,EA{%dn}**	
	fmov.l	**EA,%control**	move from memory to special register (fmove.l forms PMMU-supported *as20* only)
	fmov.l	**EA,%status**	
	fmov.l	**EA,%iaddr**	
	fmove.l	**EA,%control**	
	fmove.l	**EA,%status**	
	fmove.l	**EA,%iaddr**	move to memory from special register (fmove.l forms PMMU-supported *as20* only)
	fmov.l	**%control,EA**	
	fmov.l	**%status,EA**	
	fmov.l	**%iaddr,EA**	
	fmove.l	**%control,EA**	
	fmove.l	**%status,EA**	
	fmove.l	**%iaddr,EA**	
FMOVECR	**fmovcr.x**	**&ccc,%fpn**	move a ROM-stored to a floating point register

MC68881 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
FMOVEM	**fmovm.x** **fmovem.x**	**EA,&l** **EA,&l**	move to multiple floating point register (fmovem.x form PMMU-supported *as20* only)
	fmovm.x **fmovem.x**	**&l,EA** **&l,EA**	move from multiple registers to memory (fmovem.x PMMU-supported *as20* only)
	fmovm.x **fmovem.x**	**EA,%dn** **EA,%dn**	move to a data register (fmovem.x form PMMU-supported *as20* only)
	fmovm.x **fmovem.x**	**%dn,EA** **%dn,EA**	move a data register to memory (fmovem.x PMMU-supported *as20* only)
	fmovm.l **fmovem.l**	**EA,%control/%status/%laddr** **EA,%control/%status/%laddr**	move to special registers (fmovem.l form PMMU-supported *as20* only)
	fmovm.l **fmovem.l**	**%control/%status/%laddr,EA** **%control/%status/%laddr,EA**	move from special registers (fmovem.l form PMMU-supported *as20* only)
FMUL	**fmul.SF** **fmul.x**	**EA,%fpn** **%fpm,%fpn**	floating point multiply
FNEG	**fneg.SF** **fneg.x** **fneg.x**	**EA,%fpn** **%fpm,%fpn** **%fpn**	negate function

NOTE: The immediate operand is a mask designating which registers are to be moved to memory or which registers are to receive memory data. Not all addressing modes are permitted and the correspondence between mask bits and register numbers depends on the addressing mode used.

MC68881 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
FNOP	fnop		floating point no-op
FREM	frem.SF frem.x	EA,%fpn %fpm,%fpn	floating point remainder
FRESTORE	frestore	EA	restore internal state of co-processor
FSAVE	fsave	EA	co-processor save
FSCALE	fscale.SF fscale.x	EA,%fpn %fpm,%fpn	floating point scale exponent
FScc	fscc.b	EA	set on condition
FSGLDIV	fsgldiv.B fsgldiv.s	EA,%fpn %fpm,%fpn	floating point single precision divide
FSGLMUL	fsglmul.B fsglmul.s	EA,%fpn %fpm,%fpn	floating point single precision multiply
FSIN	fsin.SF fsin.x fsin.x	EA,%fpn %fpm,%fpn %fpn	sine function
FSINCOS	fsincos.SF fsincos.x	EA,%fpn:%fpq %fpm,%fpn:%fpq	sine/cosine function
FSINH	fsinh.SF fsinh.x fsinh.x	EA,%fpn %fpm,%fpn %fpn	hyperbolic sine function
FSQRT	fsqrt.SF fsqrt.x fsqrt.x	EA,%fpn %fpm,%fpn %fpn	square root function
FSUB	fsub.SF fsub.x	EA,%fpn %fpm,%fpn	square root function

MC68881 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
FTAN	**ftan.SF** **ftan.x** **ftan.x**	**EA,%fpn** **%fpm,%fpn** **%fpn**	tangent function
FTANH	**ftanh.SF** **ftanh.x** **ftanh.x**	**EA,%fpn** **%fpm,%fpn** **%fpn**	hyperbolic tangent function
FTENTOX	**ftentox.SF** **ftentox.x** **ftentox.x**	**EA,%fpn** **%fpm,%fpn** **%fpn**	10**x function
FTcc	**ftcc**		trap on condition without a parameter[9]
FTRAPcc	**ftrapcc**		trap on condition without a parameter (PMMU-supported *as20* only)
FTPcc	**ftpcc.A**	**&I**	trap on condition with a parameter
FTRAPcc	**ftrapcc.A**	**&I**	trap on condition with a parameter (PMMU-supported *as20* only)
FTST	**ftest.SF** **ftest.x** **ftst.SF** **ftst.x**	**EA** **%fpm** **EA** **%fpm**	floating point test an operand (ftst.SF and ftst.x forms PMMU-supported *as20* only)
FTWOTOX	**ftwotox.SF** **ftwotox.x** **ftwotox.x**	**EA,%fpn** **%fpm,%fpn** **%fpn**	2**x function

9. The ftst form (floating point trap on signal true) is no longer supported due to a conflict with the FTST (floating point test and operand instruction) - PMMU-supported *as20* only.

Instructions For The MC68851

The following table shows how the paged memory management unit (PMMU) (MC68851) instructions should be written to be understood by the *as* assembler. Instructions that are MC68030-only or MC68851-only are noted as such in the "OPERATION" column. Additional MC68030 instructions which do not deal with memory management are listed separately with the MC68000 instructions.

In the table, *cc* represents any of the following floating point condition code designations:

SET PSR BIT

cc/CC	MEANING
bs	bus error
ls	limit violation
ss	supervisor violation
as	access level violation
ws	write protected
is	invalid
gs	gate
cs	globally shared

CLEAR PSR BIT

cc/CC	MEANING
bc	bus error
lc	limit violation
sc	supervisor violation
ac	access level violation
wc	write protected
ic	invalid
gc	gate
cc	globally shared

Additional abbreviations used in the table are:

EA	represents an effective addresss
L	a label reference or any expression representing a memory address in the current segment
I	represents an absolute expression, used as an immediate operand

FC	represents one of the following function codes:
	I represents an absolute expression used as an immediate operand
	%dn represents a data register
	%sfc represents the source function code register
	%sfcr represents the source function code register
	%dfc represents the destination function code register
	%dfcr represents the destination function code register
M	represents an absolute expression used as an immediate operand mask in the PFLUSH/PFLUSHS instructions where $0 \leq M \leq 15$
D	represents an absolute expression used as an immediate operand depth level in the PTESTR/PTESTW instructions where $0 \leq D \leq 7$
PMRn	represents any of the MC68881 registers
MRn	represents any of the MC68030 memory management registers
%dn	represents a data register 0 through 7
%an	represents an address register 0 through 7
%ac	represents pmmu access control register (MC68851 only)
%bac	represents pmmu breakpoint acknowledge control register 0 through 7 (MC68851 only)
%bad	represents pmmu breakpoint acknowledge data register 0 through 7 (MC68851 only)
%cal	represents pmmu current access level register (MC68851 only)
%crp	represents pmmu CPU root pointer register
%drp	represents pmmu DMA root pointer register (MC68851 only)
%mmusr	represents pmmu status register
%pcsr	represents pmmu cache status register (MC68851 only)
%psr	represents pmmu status register
%scc	represents pmmu stack change control register (MC68851 only)
%srp	represents pmmu supervisor root pointer register
%tc	represents pmmu translation control register
%tt	represents pmmu transparent translation control registers 0 or 1 (MC68030 only)
%val	represents pmmu validate access level register (MC68851 only)

Note: The source format must be specified if more than one source format is permitted or a default source format *w* is assumed. Source format need not be specified if only one format is permitted by the operation.

MC68851 INSTRUCTION FORMATS

MNEMONIC	ASSEMBLER SYNTAX		OPERATION
PBcc	pbCC.A	L	Branch on PMMU Condition (MC68851 only)
PDBcc	pdbCC.w	%dn,L	Test, decrement, branch (MC68851 only)
PFLUSH	pflush	FC,&M	Invalidate entries in ATC
	pflush	FC,&M,EA	
PFLUSHA	pflusha		Invalidate all ATC entries
PFLUSHS	pflushs	FC,&M	Invalidate entries in ATC including shared entries (MC68851 only)
	pflushs	FC,&M,EA	
PFLUSHR	pflushr	EA	Invalidate ATC and RPT entries
PLOADR	ploadr	FC,EA	Load an entry into ATC
PLOADW	ploadw	FC,EA	Load an entry into ATC
PMOVE			Move to/from MMU register
	pmove	EA,PMRn	(MC68851 only)
	pmove	PMRn,EA	(MC68851 only)[10]
	pmove	EA,MRn	(MC68030 only)
	pmove	MRn,EA	(MC68030 only)
	pmove.d	%crp,EA	
	pmove.d	EA,%crp	
	pmove.d	%srp,EA	
	pmove.d	EA,%srp	
	pmove.d	%drp,EA	(MC68881 only)
	pmove.d	EA,%drp	(MC68881 only)
	pmove.l	%tc,EA	
	pmove.l	EA,%tc	
	pmove.l	%tt,EA	(MC68881 only)
	pmove.l	EA,%tt	(MC68881 only)
	pmove.w	%bac,EA	(MC68881 only)
	pmove.w	EA,%bac	(MC68881 only)
	pmove.w	%bad,EA	(MC68881 only)
	pmove.w	EA,%bad	(MC68881 only)
	pmove.w	%ac,EA	
	pmove.w	EA,%ac	
	pmove.w	%psr,EA	(MC68881 only)
	pmove.w	EA,%psr	
	pmove.w	%pcsr,EA	

10. Cannot move to %pcsr register.

MC68851 INSTRUCTION FORMATS		
MNEMONIC	ASSEMBLER SYNTAX	OPERATION
PMOVE (cont'd)		
	pmove.b %cal,EA	(MC68881 only)
	pmove.b EA,%cal	(MC68881 only)
	pmove.b %val,EA	(MC68881 only)
	pmove.b EA,%val	(MC68881 only)
	pmove.b %scc,EA	(MC68881 only)
	pmove.b EA,%scc	(MC68881 only)
PMOVEFD		Move to MMU register, flush disabled
	pmovefd EA,MRn	(MC68030 only)
	pmovefd.dEA,%crp	(MC68030 only)
	pmovefd.dEA,%srp	(MC68030 only)
	pmovefd.l EA,%tc	(MC68030 only)
	pmovefd.l EA,%tt	(MC68030 only)
PRESTORE	**prestore EA**	PMMU restore function (MC68881 only)
PSAVE	**psave EA**	PMMU save function (MC68881 only)
PScc	**psCC EA**	Set on PMMU condition (MC68881 only)
PTESTR	**ptestr FC,EA,&D** **ptestr FC,EA,&D,%an**	Get information about logical address
[redacted]	[redacted] **FC,EA,&D** **ptestr FC,EA,&D,%an**	Get information about logical address
PTRAPcc	**ptrapCC** **ptrapCC.A &I**	Trap on PMMU condition (MC68881 only)
PVALID	**pvalid %val,EA** **pvalid %an,EA**	Validate a pointer (MC68881 only)

INDEX

When using this index, keep in mind that a page number indicates only where referenced material begins; it may extend to the following page or pages.