Release and Installation Notes

Microport's System V/AT Software Development System

Release 2.3

April 1988

You have received *Microport's System V/AT Software Development System*, release 2.3. There have been many improvements in this release. Some of the changes are minor, like fixing the shell scripts so they can be run with the csh command processor. Some major library and utility changes were also made, and the programs that were previously distributed on the Upgrade disk are now incorporated in this product.

Preface

These *Release Notes* use the same typesetting conventions as used by all UNIX documentation to denote command names, command line format, file and directory names, and the display of terminal input and output.

- **Boldface** words must be typed as they appear.
- *Italicized* words are variables; you substitute the appropriate values. These values may be file names or data values. *Italics* are also used to denote product names and references.
- `Constant Width` font shows terminal output and examples of source code.
- Square brackets, [], surround characters or words that are optional.

A command name followed by a number, for example, ed(1), refers to that command's manual page, and the number refers to the section of the manual. Pages noted like *section (1)* can be found, unless otherwise noted, in the *User's Reference Manual* in the *Runtime Manual*. Pages noted like *section (1M)* appear in the *System Administrator's Reference Manual*, also contained in the *Runtime Manual*.

Examples in these *Release Notes* show the default system prompt for UNIX System V, the dollar sign ($). They also show the default prompt when you log in as the superuser, the pound sign (#).

From the Upgrade Disk

- The **lint** utility and its associated libraries are now part of this product.
- The improved **sdb**, for better application debugging, is now included.
- The **prof** utility has also been moved from the Upgrade diskette to this product.
- The **libdial.a** library, for accessing the modem capability database, is also included.

New in this Release

- The newer command processor, **ksh**, is included in this release. There is also a manual page on line that is available with the **man** command.
- There have been some fixes to the standard C library, libc.a, to fix a problem with the *printf()* family of functions. (See section on fixed problems)
- There have been some fixes to the math library, libm.a, to fix the *log()* and *log10()* functions. (See section on fixed problems)
- The utility **ctags** is now available with this product.
- The **cflow** utility bas been fixed for this release. (See section on fixed problems)
- A new version of **yacc** that can handle more complicated syntax has been included.
- The include files in "/usr/include" and "/usr/include/sys" have been updated.

Installation Description

There are 4 diskettes that make up the Software Development System. The fourth diskette is an optional diskette. These diskettes are in the **installit** format. Please follow the steps described below to install this package.

Steps to Install

Follow these steps to install this Software package:

1. Log in as **root**.

2. Insert the first floppy into the floppy drive.
3. Type **installit** to start the software package installation program.
4. Answer '**y**' to the question "`Is this ok? (y or n)`".
5. Insert the next floppy and repeat step 3 for each of the remaining diskettes.
6. If you are adding the *Software Development System* after having completely installed the rest of the system, it may be necessary to install the *Upgrade Disk* again. The *Upgrade Disk* is used to upgrade all the packages to the current release. If the *Upgrade Disk* has not yet been installed, proceed with the installation instructions in the *Runtime Installation Notes*.

Fixed Problems

- A process which makes a **sleep** system call in the large model may grow in size over time (#764, fixed in 2.3.1).
- **cflow** gets "`lpfx: PANIC! nargs=258`" no matter what args it's fed. There is no workaround available (#716, fixed in 2.3).

Open Problems

- File systems greater than approximately 130,000 blocks may experience corruption over time that **fsck** can't repair. **fsck** can give negative numbers and corrupt the file system further. Workaround: repair with **fsck -f**, check again with **fsck**; Also, make file systems smaller than 130,000 blocks (#605, verified).
- Some systems with two drives installed can experience a problem which may be hardware-related, and which seems to occur rarely, as follows: Phantom hard disk I/O errors can appear at random, not consistently, when both filesystems are being accessed at once, such as by running **cpio -p** between the two drives. There is no workaround (#1047, verified).
- In the C compiler, declaring an array of pointers to structures whose total size would be greater than 64K causes the C compiler to complain, even though the array itself is not greater than 64K. Workaround: Avoid this configuration (#329, verified).
- In the C compiler, a "`wasted space`" error message is generated when a variable has the same name as a function. Workaround: Change the variable

name (#350, verified).

- The **-L** option to **ld** does not work. Workaround: Keep all libraries in /lib or /usr/lib (#435, verified).
- Compiling the following code with **-O -Ml** option to **cc** gets "`Fatal error in /lib/optim: Status 0213`": `struct { float a,b; } *p; c(x.y) float *x,*y; { *x=p->a; *y=p->b; }` Workaround: compile without the optimizer; also, fixed in the V.3 optimizer (#472, fixed, not incorporated).
- The C compiler generates "`compiler error: faulty register move`" with the following code: `main() { int size,i; size = (i%2==0?3:4)<<(i/2-2); }` (#608, verified).
- The following C program gets "`illegal indirection`" error at compile time. Okay if only 3 dimensions. `main() { char a[5][5][5][5]; a[1][1][1][1]=0; }` (#828, verified).
- The following code causes the user to be logged out: `#include <curses.h> main() { initscr(); nodelay(stdscr,TRUE); endwin(); }` Compiled with **cc -Ml tiny.c -lcurses -otiny** Workaround: add to the program: `nodelay(stdscr,FALSE);` (#1124, verified).

Software Development System

Vol. I

Reorder No. 0102

MICROPORT SYSTEMS

The material contained in this manual was reprinted with permission from AT&T and is comprised of excerpts from the following AT&T manuals.

*UNIX **System V - Release 2.0 Support Tools Guide**	April 1984 307-108, Issue 2
UNIX **System V - Release 2.0 Supplement** †INTEL **Processors**	September 1985 307-624, Issue 2
UNIX **System V - Release 2.0 Programming Guide**	April 1984 307-103, Issue 2
UNIX **System V - Release 2.0 Programmer Reference Manual** INTEL **Processors**	March 1985 307-627, Issue 1

QUICK REFERENCE GUIDE

COMPILER AND C LANGUAGE	C-1
C LANGUAGE	C-2
C PROGRAM CHECKER (Lint)	C-3
PROGRAM PROCEDURES	C-4
MAINT COMPUTER PROGS (Make)	C-5
AUG VERSION OF (Make)	C-6
SCCS USER'S GUIDE	C-7
M4 MACRO PROCESSOR	C-8
LINK EDITOR	C-9
COFF	C-10
SYMBOLIC DEBUGGING PROG.	C-11
FORTRAN 77	C-12
RATFOR	C-13
PROGRAMMING LANGUAGE EFL	C-14
CURSES & TERMINFO PACKAGE	C-15
CURSES EXAMPLES	C-16

Table of Contents

TITLE	CHAPTER
COMPILER AND C LANGUAGE	1
C LANGUAGE	2
C PROGRAM CHECKER (Lint)	3
PROGRAM PROCEDURES FOR UNIX SYSTEM V/AT	4
MAINT COMPUTER PROGS (Make)	5
AUG VERSION OF (Make)	6
SCCS USER'S GUIDE	7
M4 MACRO PROCESSOR	8
LINK EDITOR	9
COFF	10
SYMBOLIC DEBUGGING PROGRAM	11
FORTRAN 77	12
RATFOR	13
PROGRAMMING LANGUAGE EFL	14
CURSES AND TERMINFO PACKAGE	15
CURSES EXAMPLES	16

Chapter 1

COMPILER AND C LANGUAGE

This chapter describe the UNIX System's C compiler, **cc**, and the C programming language that the compiler translates. The compiler is part of the UNIX System Software Generation System (SGS).

The SGS is a package of tools used to create and test programs for UNIX Systems. These tools allow high-level program coding and source-level testing of code. The C language is implemented for high-level programming; it contains many control and structuring facilities that greatly simplify the task of algorithm construction. Within the SGS, a C compiler converts C programs into assembly language programs that are ultimately translated into object files by the assembler, **as**. The link editor, **ld**, collects and merges object files into executable load modules. Each of these tools preserves all symbolic information necessary for meaningful symbolic testing at C-language source level. In addition, a utility package aids in testing and debugging.

The current manual page for the C compiler can be obtained with the SGS command:

man cc

USE OF THE COMPILER

The main command of the SGS is **cc**; it operates much like the UNIX system **cc** command. To use the compiler, first creat a file (typically by using the UNIX system text editor) containing C source code. The name of the file created must have a special format; the last two characters of the file name must be **.c** as in *file1.c*.

Next, enter the SGS command

cc *options file.c*

to invoke the compiler on the C source file *file.c* with the appropriate

options selected. The compilation process creates an absolute binary file named **a.out** that reflects the contents of *file.c* and any referenced library routines. The resulting binary file, **a.out**, can then be executed on the target system.

Options can control the steps in the compilation process. When none of the controlling options are used, and only one file is named, **cc** automatically calls the assembler, **as**, and the link editor, **ld**, thus resulting in an executable file, named **a.out**. If more than one file is named in a command,

cc *file1.c file2.c file3.c*

then the output will be placed on files *file1.o*, *file2.o*, and *file3.o*. These files can then be linked and executed through the **ld** command.

The **cc** compiler also accepts input file names with the last two characters **.s**. The **.s** signifies a source file in assembly language. The **cc** compiler passes this type of file directly to **as**, which assembles the file and places the output on a file of the same name with **.o** substituted for **.s**.

Cc is based on a portable C compiler and translates C source files into assembly code. Whenever the command **cc** is used, the standard C preprocessor (which resides on the file **/lib/cpp**) is called. The preprocessor performs file inclusion and macro substitution. The preprocessor is always invoked by **cc** and need not be called directly by the programmer. Then, unless the appropriate flags are set, **cc** calls the assembler and the link editor to produce an executable file.

COMPILER OPTIONS

For more detailed information, see **cc**(1), **cpp**(1), **ld**(1) in the Runtime System manual.

Option	*Argument*	*Description*
-c	none	Suppress the link-editing phase of compilation and force an object file to be produced even if only one file is compiled.
-g	none	Produce symbolic debugging information.
-o	outfile	Produce an output object file by the name *outfile*. The name of the default object file is **a.out.**
-p	none	Reserved for invoking a profiler.
-D	*identifier[=constant]*	Define the external symbol *identifier* to the preprocessor, and give it the value *constant* (if specified).
-E	none	Same as the **-P** option except output is directed to the standard output.
-I	*directory*	Change the algorithm that searches for **#include** files whose names do not begin with / to look in the named *directory* before looking in the directories on the standard list. Thus, **#include** files whose names are enclosed in "" are searched for first in the directory of the file being compiled, then in directories named by the **-I** options, and last in directories on the standard list. For **#include** files whose names are enclosed in <>, the directory of the *file* argument is not searched.

-O	none	Invoke an object code optimizer.
-P	none	Suppress compilation and loading; i.e., invoke only the preprocessor and leave out the output on corresponding files suffixed **.i**.
-U	*identifier*	Undefine the named *identifier* to the preprocessor.
-V	none	Print the version of the assembler that is invoked.
-W	*c,arg1[,arg2...]*	Pass along the argument(s) *argi* to pass *c*, where *c* is one of [**p012al**], indicating preprocessor, compiler first pass, compiler second pass, optimizer, assembler, or link editor, respectively.

This part provides additional information for those options not completely described above.

By using appropriate options, compilation can be terminated early to produce one of several intermediate translations such as relocatable object files (**-c** option), assembly source expansions for C code (**-S** option), or the output of the preprocessor (**-P** option). In general, the intermediate files may be saved and later resubmitted to the **cc** command, with other files or libraries included as necessary.

When compiling C source files, the most common practice is to use the **-c** option to save relocatable files. Subsequent changes to one file do not then require that the others be recompiled. A separate call to **cc** without the **-c** option then creates the linked executable **a.out** file. A relocatable object file created under the **-c** option is named by adding a **.o** suffix to the source file name.

The **-W** option provides the mechanism to specify options for each step that is normally invoked from the **cc** command line. These steps are preprocessing, the first pass of the compiler, the second pass of the compiler, optimization, assembly, and link editing. At this time,

only assembler and link editor options can be used with the **-W** option. The most common example of use of the -W option is "-W*a,-m*", which passes the **-m** option to the assembler. Specifying "**-wl,-m**" passes the **-m** option to the link editor.

When the **-P** option is used, the compilation process stops after only preprocessing, with output left on *file.i*. This file will be unsuitable for subsequent processing by **cc**.

The **-O** option decreases the size and increases the execution speed of programs by moving, merging, and deleting code. However, line numbers used for symbolic debugging may be transposed when the optimizer is used.

The **-g** option produces information for a symbolic debugger. The SGS currently supports the SDB symbolic debugger.

Chapter 2

C LANGUAGE

LEXICAL CONVENTIONS

There are six classes of tokens - identifiers, keywords, constants, strings, 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 which could possibly constitute a token.

Comments

The characters /* introduce a comment which 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. Although there is no limit on the length of a name, only initial characters are significant: at least eight characters of a non-external name, and perhaps fewer for external names. Moreover, some implementations may collapse case distinctions for external names. The external name sizes include:

PDP-11	7 characters, 2 cases
VAX-11	>100 characters, 2 cases
AT&T 3B 20	>100 characters, 2 cases

Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

auto	**do**	**for**	**return**	**typedef**
break	**double**	**goto**	**short**	**union**
case	**else**	**if**	**sizeof**	**unsigned**
char	**enum**	**int**	**static**	**void**
continue	**external**	**long**	**struct**	**while**
default	**float**	**register**	**switch**	

Some implementations also reserve the words **fortran** and **asm.**

Constants

There are several kinds of constants. Each has a type; an introduction to types is given in "NAMES." Hardware characteristics that affect sizes are summarized in "Hardware Characteristics" under "LEXICAL CONVENTIONS."

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 which exceeds the largest unsigned machine integer is likewise taken to be **long**. Otherwise, integer constants are **int.**

Explicit Long Constants

A decimal, octal, or hexadecimal integer constant immediately followed by **l** (letter ell) or **L** is a long constant. As discussed below, on some machines integer and long values may be considered identical.

Character Constants

A character constant is a character enclosed in single quotes, as in '**x**'. The value of a character constant is the numerical value of the character in the machine's character set.

Certain nongraphic characters, the single quote (') and the backslash (\), may be represented according to the following table of escape sequences:

new-line	NL (LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
single quote	'	\'
bit pattern	*ddd*	*ddd*

The escape *ddd* consists of the backslash followed by 1, 2, or 3 octal digits which 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 character **NUL**. If the character following a backslash is not one of those specified, the behavior is undefined. A 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**.

Strings

A string is a sequence of characters surrounded by double quotes, as in "...". A string has type "array of **char**" and storage class **static** (see "NAMES") and is initialized with the given characters. The compiler places a null byte (**\0**) at the end of each string so that programs which scan the string can find its end. In a string, 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 strings, even when written identically, are distinct.

Hardware Characteristics

The following figures summarize certain hardware properties that vary from machine to machine.

IBM AT & Compatibles (ASCCI)	
char	8 bits
int	16
short	16
long	32
float	32
double	64
float range	$\pm 10^{\pm 38}$
double range	$\pm 10^{\pm 38}$

Figure 2.0. **IBM AT HARDWARE CHARACTERISTICS**

DEC PDP-11 (ASCII)	
char	8 bits
int	16
short	16
long	32
float	32
double	64
float range	$\pm 10^{\pm 38}$
double range	$\pm 10^{\pm 38}$

Figure 2-1. DEC PDP-11 HARDWARE CHARACTERISTICS

DEC VAX-11 (ASCII)	
char	8 bits
int	32
short	16
long	32
float	32
double	64
float range	$\pm 10^{\pm 38}$
double range	$\pm 10^{\pm 38}$

Figure 2-2. DEC VAX-11 HARDWARE CHARACTERISTICS

AT&T 3B (ASCII)	
char	8 bits
int	32
short	16
long	32
float	32
double	64
float range	$\pm 10^{\pm 38}$
double range	$\pm 10^{\pm 308}$

Figure 2-3. AT&T 3B HARDWARE CHARACTERISTICS

SYNTAX NOTATION

Syntactic categories are indicated by *italic* type and literal words and characters in **bold** type. Alternative categories are listed on separate lines. An optional terminal or nonterminal symbol is indicated by the subscript "opt," so that

$$\{\ expression_{opt}\ \}$$

indicates an optional expression enclosed in braces. The syntax is summarized in "SYNTAX SUMMARY".

NAMES

The C language bases the interpretation of an identifier upon two attributes of the identifier - its *storage class* and its *type*. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

Storage Class

There are four declarable storage classes:

- Automatic
- Static
- External
- Register.

Automatic variables are local to each invocation of a block (see "Compound Statement or Block" in "STATEMENTS") and are discarded upon exit from the block. Static variables are local to a block but retain their values upon reentry to a block even after control has left the block. External variables exist and retain their values throughout the execution of the entire program and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables, they are local to each block and disappear on exit from the block.

Type

The C language supports several fundamental types of objects. Objects declared as characters (**char**) are large enough to store any member of the implementation's character set. If a genuine character from that character set is stored in a **char** variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine dependent. In particular, **char** may be signed or unsigned by default.

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 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. (On the PDP-11, unsigned long quantities are not supported.)

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* which 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 which 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.

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. Whether or not sign-extension occurs for characters is machine dependent, but it is guaranteed that a member of the standard character set is non-negative. Of the machines treated here, only the PDP-11 and VAX-11 sign-extend. On these machines, **char** variables range in value from -128 to 127. The more explicit type **unsigned char** forces the values to range from 0 to 255.

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 are well behaved. Some loss of accuracy occurs if the destination lacks sufficient bits.

Pointers and Integers

An expression of integral type may be added to or subtracted from a pointer; in such a case, the first is converted as specified in the discussion of the addition operator. Two pointers to objects of the same type may be subtracted; in this case, the result is converted to an integer as specified in the discussion of the subtraction operator.

Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value is the least unsigned integer congruent to the signed integer (modulo $2^{wordsize}$). In a 2's complement representation, this conversion is conceptual; and there is no actual change in the bit pattern.

When an unsigned **short** integer is converted to **long**, the value of the result is the same numerically as that of the unsigned integer. Thus the conversion amounts to padding with zeros on the left.

Arithmetic Conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the "usual arithmetic conversions."

1. First, any operands of type **char** or **short** are converted to **int**, and any operands of type **unsigned char** or **unsigned short** are converted to **unsigned int**.

2. Then, if either operand is **double**, the other is converted to **double** and that is the type of the result.

3. Otherwise, if either operand is **unsigned long**, the other is converted to **unsigned long** and that is the type of the result.

4. Otherwise, if either operand is **long**, the other is converted to **long** and that is the type of the result.

5. Otherwise, if one operand is **long**, and the other is **unsigned int**, they are both converted to **unsigned long** and that is the type of the result.

6. Otherwise, if either operand is **unsigned**, the other is converted to **unsigned** and that is the type of the result.

7. Otherwise, both operands must be **int**, and that is the type of the result.

Void

The (nonexistent) value of a **void** object may not be used in any way, and neither explicit nor implicit conversion may be applied. Because a void expression denotes a nonexistent value, such an expression may be used only as an expression statement (see "Expression Statement" under "STATEMENTS") or as the left operand of a comma expression (see "Comma Operator" under "EXPRESSIONS").

An expression may be converted to type **void** by use of a cast. For example, this makes explicit the discarding of the value of a function call used as an expression statement.

EXPRESSIONS

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. The order in which subexpression evaluation takes place is unspecified. Expressions involving a commutative and associative operator (*, +, &, |, ^) may be rearranged arbitrarily even in the presence of parentheses; to force a particular order of evaluation, an explicit temporary must be used.

The handling of overflow and divide check in expression evaluation is undefined. Most existing implementations of C ignore integer overflows; treatment of division by 0 and all floating-point exceptions varies between machines and is usually adjustable by a library function.

Primary Expressions

Primary expressions involving ., ->, subscripting, and function calls group left to right.

primary-expression:
identifier
constant
string
(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 which 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 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. (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 which 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; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type **float** are converted to **double** before the call. Any of type **char** or **short** are converted to **int**. Array names are converted to pointers. No other conversions are performed automatically; in particular, the compiler does not compare the types of actual arguments with those of formal arguments. If conversion is needed, use a cast; see "Unary Operators" and "Type Names" under "DECLARATIONS."

In preparing for the call to a function, a copy is made of each actual parameter. Thus, all argument passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. It is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. An array name is a pointer expression. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ. Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be a structure or a union,

and the identifier must name a member of the structure or union. The value is the named member of the structure or union, and it is an lvalue if the first expression is an lvalue.

A primary expression followed by an arrow (built from - and >) followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points. Thus the expression **E1->MOS** is the same as **(*E1).MOS**. Structures and unions are discussed in "Structure, Union, and Enumeration Declarations" under "DECLARATIONS."

Unary Operators

Expressions with unary operators group right to left.

unary-expression:
- ** expression*
- *& lvalue*
- *- expression*
- *! expression*
- *~ expression*
- *++ lvalue*
- *--lvalue*
- *lvalue ++*
- *lvalue --*
- *(type-name) expression*
- sizeof *expression*
- sizeof *(type-name)*

The unary * operator means *indirection* ; the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is "pointer to ...," the type of the result is "...".

The result of the unary & operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is "...", the type of the result is "pointer to ...".

The result of the unary - operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from 2^n where n is the number of bits in the corresponding signed type.

There is no unary + operator.

The result of the logical negation operator ! is one if the value of its operand is zero, zero if the value of its operand is nonzero. The type of the result is **int**. It is applicable to any arithmetic type or to pointers.

The ~ operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix ++ is incremented. The value is the new value of the operand but is not an lvalue. The expression **++x** is equivalent to **x=x+1**. See the discussions "Additive Operators" and "Assignment Operators" for information on conversions.

The lvalue operand of prefix -- is decremented analogously to the prefix ++ operator.

When postfix ++ is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix ++ operator. The type of the result is the same as the type of the lvalue expression.

When postfix -- is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the prefix -- operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. This construction is called a *cast*. Type names are described in "Type Names" under "Declarations."

The **sizeof** operator yields the size in bytes of its operand. (A *byte* is undefined by the language except in terms of the value of **sizeof**. However, in all existing implementations, a byte is the space required to hold a **char**.) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an **unsigned** constant and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The **sizeof** operator may also be applied to a parenthesized type name. In that case it yields the size in bytes of an object of the indicated type.

The construction **sizeof(***type***)** is taken to be a unit, so the expression **sizeof(***type***)-2** is the same as **(sizeof(***type***))-2**.

Multiplicative Operators

The multiplicative operators *, /, and % group left to right. The usual arithmetic conversions are performed.

multiplicative expression:
*expression * expression*
expression / expression
expression % expression

The binary * operator indicates multiplication. The * operator is associative, and expressions with several multiplications at the same level may be rearranged by the compiler. The binary / operator indicates division.

The binary % operator yields the remainder from the division of the first expression by the second. The operands must be integral.

When positive integers are divided, truncation is toward 0; but the form of truncation is machine-dependent if either operand is negative. On all machines covered by this manual, the remainder has the same sign as the dividend. It is always true that **(a/b)*b + a%b** is equal to **a** (if **b** is not 0).

Additive Operators

The additive operators + and - group left to right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

additive-expression:
expression + expression
expression - expression

The result of the + operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer which points to another object in the same array, appropriately offset from the original object. Thus if **P** is a pointer to an object in an array, the expression **P+1** is a pointer to the next object in the array. No further type combinations are allowed for pointers.

The + operator is associative, and expressions with several additions at the same level may be rearranged by the compiler.

The result of the - operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a pointer, and then the same conversions for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an **int** representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object length.

Shift Operators

The shift operators << and >> group left to right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to **int**; the type of the result is that of the left operand. The result is undefined if the right operand is negative or greater than or equal to the length of the object in bits.

shift-expression:
expression << *expression*
expression >> *expression*

The value of **E1<<E2** is **E1** (interpreted as a bit pattern) left-shifted **E2** bits. Vacated bits are 0 filled. The value of **E1>>E2** is **E1** right-shifted **E2** bit positions. The right shift is guaranteed to be logical (0 fill) if **E1** is **unsigned**; otherwise, it may be arithmetic.

Relational Operators

The relational operators group left to right.

relational-expression:
expression < *expression*
expression > *expression*
expression <= *expression*
expression >= *expression*

The operators < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is **int**. The usual arithmetic conversions are performed. Two pointers may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

Equality Operators

equality-expression:
expression == expression
expression != expression

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus **a<b == c<d** is 1 whenever **a<b** and **c<d** have the same truth value).

A pointer may be compared to an integer only if the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object and will appear to be equal to 0. In conventional usage, such a pointer is considered to be null.

Bitwise AND Operator

and-expression:
expression & expression

The **&** operator is associative, and expressions involving **&** may be rearranged. The usual arithmetic conversions are performed. The result is the bitwise AND function of the operands. The operator applies only to integral operands.

Bitwise Exclusive OR Operator

exclusive-or-expression:
expression ^ expression

The ^ operator is associative, and expressions involving ^ may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

Bitwise Inclusive OR Operator

inclusive-or-expression:
expression | *expression*

The | operator is associative, and expressions involving | may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

Logical AND Operator

logical-and-expression:
expression && *expression*

The **&&** operator groups left to right. It returns 1 if both its operands evaluate to nonzero, 0 otherwise. Unlike **&**, **&&** guarantees left to right evaluation; moreover, the second operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

Logical OR Operator

logical-or-expression:
expression || *expression*

The || operator groups left to right. It returns 1 if either of its operands evaluates to nonzero, 0 otherwise. Unlike |, || guarantees left to right evaluation; moreover, the second operand is not evaluated if the value of the first operand is nonzero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

Conditional Operator

conditional-expression:
expression ? expression : expression

Conditional expressions group right to left. The first expression is evaluated; and if it is nonzero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type. If both are structures or unions of the same type, the result has the type of the structure or union. If both pointers are of the same type, the result has the common type. Otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

Assignment Operators

There are a number of assignment operators, all of which group right to left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are separate tokens.

assignment-expression:
lvalue = expression
lvalue += expression
lvalue -= expression
*lvalue *= expression*
lvalue /= expression
lvalue %= expression
lvalue >>= expression
lvalue <<= expression
lvalue &= expression
lvalue ^= expression
lvalue |= expression

In the simple assignment with =, the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left

preparatory to the assignment. Second, both operands may be structures or unions of the same type. Finally, if the left operand is a pointer, the right operand must in general be a pointer of the same type. However, the constant 0 may be assigned to a pointer; it is guaranteed that this value will produce a null pointer distinguishable from a pointer to any object.

The behavior of an expression of the form **E1** *op* = **E2** may be inferred by taking it as equivalent to **E1** = **E1** *op* (**E2**); however, **E1** is evaluated only once. In += and -=, the left operand may be a pointer; in which case, the (integral) right operand is converted as explained in "Additive Operators." All right operands and all nonpointer left operands must have arithmetic type.

Comma Operator

comma-expression:
expression , *expression*

A pair of expressions separated by a comma is evaluated left to right, and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left to right. In contexts where comma is given a special meaning, e.g., in lists of actual arguments to functions (see "Primary Expressions") and lists of initializers (see "Initialization" under "DECLARATIONS"), the comma operator as described in this subpart can only appear in parentheses. For example,

```
f(a, (t=3, t+2), c)
```

has three arguments, the second of which has the value 5.

DECLARATIONS

Declarations are used to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

declaration:
decl-specifiers declarator-list$_{opt}$;

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

decl-specifiers:
type-specifier decl-specifiers$_{opt}$
sc-specifier decl-specifiers$_{opt}$

The list must be self-consistent in a way described below.

Storage Class Specifiers

The sc-specifiers are:

sc-specifier:
auto
static
extern
register
typedef

The **typedef** specifier does not reserve storage and is called a "storage class specifier" only for syntactic convenience. See "Typedef" for more information. The meanings of the various storage classes were discussed in "Names."

The **auto**, **static**, and **register** declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the **extern** case, there must be an external definition (see "External Definitions") for the given identifier 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; on the PDP-11, they are **int** or pointer. One other restriction applies to register variables: the address-of operator **&** cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately, but future improvements in code generation may render them unnecessary.

At most, one sc-specifier may be given in a declaration. If the sc-specifier is missing from a declaration, it is taken to be **auto** inside a function, **extern** outside. Exception: functions are never automatic.

Type Specifiers

The type-specifiers are

type-specifier:
    struct-or-union-specifier
    typedef-name
    enum-specifier
basic-type-specifier:
    basic-type
    basic-type basic-type-specifiers
basic-type:
    char
    short
    int
    long
    unsigned
    float
    double
    void

At most one of the words **long** or **short** may be specified in conjunction with **int**; the meaning is the same as if **int** were not mentioned. The word **long** may be specified in conjunction with **float**; the meaning is the same as **double**. The word **unsigned** may be specified alone, or in conjunction with **int** or any of its short or long varieties, or with **char**.

Otherwise, at most on type-specifier may be given in a declaration. In particular, adjectival use of **long**, **short**, or **unsigned** is not permitted with **typedef** names. If the type-specifier is missing from a declaration, it is taken to be **int**.

Specifiers for structures, unions, and enumerations are discussed in "Structure, Union, and Enumeration Declarations." Declarations with **typedef** names are discussed in "Typedef."

Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer.

declarator-list:
init-declarator
init-declarator , declarator-list

init-declarator:
declarator initializer$_{opt}$

Initializers are discussed in "Initialization". The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

declarator:
identifier
(declarator)
** declarator*
declarator ()
declarator [constant-expression$_{opt}$ *]*

The grouping is the same as in expressions.

Meaning of Declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class.

Each declarator contains exactly one identifier; it is this identifier that is declared. If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

Now imagine a declaration

T D1

where **T** is a type-specifier (like **int**, etc.) and **D1** is a declarator. Suppose this declaration makes the identifier have type "... **T** ," where the "..." is empty if **D1** is just a plain identifier (so that the type of **x** in **'int x"** is just **int**). Then if **D1** has the form

***D**

the type of the contained identifier is "... pointer to **T** ."

If **D1** has the form

D()

then the contained identifier has the type "... function returning **T**."

If **D1** has the form

D[*constant-expression*]

or

D[]

then the contained identifier has type "... array of **T**." In the first case, the constant expression is an expression whose value is determinable at compile time , whose type is **int**, and whose value is positive. (Constant expressions are defined precisely in "Constant Expressions.") When several "array of" specifications are adjacent, a multidimensional array is created; the constant expressions which 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 multidimensional 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**. Using indirection through the (pointer) result to yield an integer. In the declarator **(*pfi)()**, the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called; it returns an integer.

As another example,

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

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

```
static int x3d[3][5][7];
```

declares a static 3-dimensional array of integers, with rank 3×5×7. In complete detail, **x3d** is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions **x3d**, **x3d[i]**, **x3d[i][j]**, **x3d[i][j][k]** may reasonably appear in an expression. The first three have type "array" and the last has type **int**.

Structure and Union Declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object which 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 which increase as the declarations are read left to right. Each nonfield 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 which does not fit into the space remaining in a word is put into the next word. No field may be wider than a word.

Fields are assigned right to left on the PDP-11 and VAX-11, left to right on the 3B 20.

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 dependant boundary.

The language does not restrict the types of things that are declared as fields, but implementations are not required to support any but integer fields. Moreover, even **int** fields may be considered to be unsigned. On the PDP-11, fields are not signed and have only integer values; on the VAX-11, fields declared with **int** are treated as containing a sign. For these reasons, it is strongly recommended that fields be declared as **unsigned.** In all implementations, 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 which 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 which 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 which contain pointers to each other.

The names of members and tags do not conflict with each other or with ordinary variables. A particular name may not be used twice in the same structure, but the same name may be used in several different structures in the same scope.

A simple but important example of a structure declaration is the following binary tree structure:

```
struct tnode
{
        char tword[20];
        int count;
        struct tnode *left;
        struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the declaration

```
struct tnode s, *sp;
```

declares **s** to be a structure of the given sort and **sp** to be a pointer to a structure of the given sort. With these declarations, the expression

```
sp->count
```

refers to the **count** field of the structure to which **sp** points;

```
s.left
```

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

s.right->tword[0]

refers to the first character of the **tword** member of the right subtree of **s**.

Enumeration Declarations

Enumeration variables and constants have integral type.

enum-specifier:
enum { *enum-list* }
enum *identifier* { *enum-list* }
enum *identifier*

enum-list:
enumerator
enum-list , *enumerator*

enumerator:
identifier
identifier = *constant-expression*

The identifiers in an enum-list are declared as constants and may appear wherever constants are required. If no enumerators with = appear, then the values of the corresponding constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

The names of enumerators in the same scope must all be distinct from each other and from those of ordinary variables.

The role of the identifier in the enum-specifier is entirely analogous to that of the structure tag in a struct-specifier; it names a particular enumeration. For example,

```
enum color { chartreuse, burgundy, claret=20, winedark };
...
enum color *cp, col;
...
col = claret;
cp = &col;
...
if (*cp == burgundy) ...
```

makes **color** the enumeration-tag of a type describing various colors, and then declares **cp** as a pointer to an object of that type, and **col** as an object of that type. The possible values are drawn from the set {0,1,20,21}.

Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by = and consists of an expression or a list of values nested in braces.

initializer:
= *expression*
= { *initializer-list* }
= { *initializer-list* , }

initializer-list:
expression
initializer-list , *initializer-list*
{ *initializer-list* }
{ *initializer-list* , }

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in "CONSTANT EXPRESSIONS", or expressions which 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. In this case successive characters of the string initialize the members of the array.

For example,

```
int x[] = { 1, 3, 5 };
```

declares and initializes **x** as a one-dimensional array which has three members, since no size was specified and there are three initializers.

```
float y[4][3] =
{
        { 1, 3, 5 },
        { 2, 4, 6 },
        { 3, 5, 7 },
};
```

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array **y[0]**, namely **y[0][0]**, **y[0][1]**, and **y[0][2]**. Likewise, the next two lines initialize **y[1]** and **y[2]**. The initializer ends early and therefore **y[3]** is initialized with 0. Precisely, the same effect could have been achieved by

```
float y[4][3] =
{
        1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for **y** begins with a left brace but that for **y[0]** does not; therefore, three elements from the list are used. Likewise, the next three are taken successively for **y[1]** and **y[2]**. Also,

```
float y[4][3] =
{
        { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of **y** (regarded as a two-dimensional array) and leaves the rest 0.

Finally,

```
char msg[] = " Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string.

Type Names

In two contexts (to specify type conversions explicitly by means of a cast and as an argument of **sizeof**), it is desired to supply the name of a data type. This is accomplished using a "type name", which in essence is a declaration for an object of that type which omits the name of the object.

type-name:
type-specifier abstract-declarator

abstract-declarator:
empty
(abstract-declarator)
** abstract-declarator*
abstract-declarator ()
abstract-declarator [*constant-expression*$_{opt}$]

To avoid ambiguity, in the construction

(abstract-declarator)

the abstract-declarator is required to be nonempty. Under this restriction, it is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int *[3]
int (*)[3]
int *()
int (*)()
int (*[3])()
```

name respectively the types "integer," "pointer to integer," "array of three pointers to integers," "pointer to an array of three integers," "function returning pointer to integer," "pointer to function

returning an integer," and "array of three pointers to functions returning an integer."

Typedef

Declarations whose "storage class" is **typedef** do not define storage but instead define identifiers which 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 which could be specified in another way. Thus in the example above **distance** is considered to have exactly the same type as any other **int** object.

STATEMENTS

Except as indicated, statements are executed in sequence.

Expression Statement

Most statements are expression statements, which have the form

expression ;

Usually expression statements are assignments or function calls.

Compound Statement or Block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called "block") is provided:

compound-statement:
{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:
declaration
declaration declaration-list

statement-list:
statement
statement statement-list

If any of the identifiers in the declaration-list were previously declared, the outer declaration is pushed down for the duration of the block, after which it resumes its force.

Any initializations of **auto** or **register** variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of **static** variables are performed only once when the program begins execution. Inside a block, **extern** declarations do not reserve storage so initialization is not permitted.

Conditional Statement

The two forms of the conditional statement are

if (*expression*) *statement*
if (*expression*) *statement* **else** *statement*

In both cases, the expression is evaluated; and if it is nonzero, the first substatement is executed. In the second case, the second substatement is executed if the expression is 0. The "else" ambiguity is resolved by connecting an **else** with the last encountered **else**-less **if**.

While Statement

The **while** statement has the form

while (*expression*) *statement*

The substatement is executed repeatedly so long as the value of the expression remains nonzero. The test takes place before each execution of the statement.

Do Statement

The **do** statement has the form

do *statement* **while** (*expression*) ;

The substatement is executed repeatedly until the value of the expression becomes 0. The test takes place after each execution of the statement.

For Statement

The **for** statement has the form:

for (*exp-1*$_{opt}$; *exp-2*$_{opt}$; *exp-3*$_{opt}$) *statement*

Except for the behavior of **continue**, this statement is equivalent to

```
exp-1 ;
while ( exp-2 )
{
        statement
        exp-3 ;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0. The third expression often specifies an incrementing that is performed after each iteration.

Any or all of the expressions may be dropped. A missing *exp-2* makes the implied **while** clause equivalent to **while(1)**; other missing expressions are simply dropped from the expansion above.

Switch Statement

The **switch** statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

switch (*expression*) *statement*

The usual arithmetic conversion is performed on the expression, but the result must be **int**. The statement is typically compound. Any

statement within the statement may be labeled with one or more case prefixes as follows:

case *constant-expression* :

where the constant expression must be **int**. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in "CONSTANT EXPRESSIONS."

There may also be at most one statement prefix of the form

default :

When the **switch** statement is executed, its expression is evaluated and compared 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 prefixed statement. 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. 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.

Break Statement

The statement

break ;

causes termination of the smallest enclosing **while**, **do**, **for**, or **switch** statement; control passes to the statement following the terminated statement.

Continue Statement

The statement

```
continue ;
```

causes control to pass to the loop-continuation portion of the smallest enclosing **while**, **do**, or **for** statement; that is to the end of the loop. More precisely, in each of the statements

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

a **continue** is equivalent to **goto contin**. (Following the **contin:** is a null statement, see "Null Statement".)

Return Statement

A function returns to its caller by means of the **return** statement which has one of the forms

```
return ;
return expression ;
```

In the first case, the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value. The expression may be parenthesized.

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 which 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 which are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of a block persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also (see "Structure, Union, and Enumeration Declarations" in "DECLARATIONS") that tags, identifiers associated with ordinary variables, and identities associated with structure and union members form three disjoint classes which do not conflict. Members and tags follow the same scope rules as other identifiers. The **enum** constants are in the same class as ordinary variables and follow the same scope rules. The **typedef** names are in the same

class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
...
{
    auto int distance;
    ...
```

The **int** must be present in the second declaration, or it would be taken to be a declaration with no declarators and type **distance**.

Scope of Externals

If a function refers to an identifier declared to be **extern**, then somewhere among the files or libraries constituting the complete program there must be at least one external definition for the identifier. All functions in a given program which 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 which 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 compiler 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. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect which lasts (independent of scope) until the end of the source program file.

Token Replacement

A compiler-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 replacement.

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 compiler 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.)

#includes may be nested.

Conditional Compilation

A compiler control line of the form

#if *restricted-constant-expression*

checks whether the restricted-constant expression evaluates to nonzero. (Constant expressions are discussed in "CONSTANT EXPRESSIONS"; the following additional restrictions apply here: the constant expression may not contain **sizeof** casts, or an enumeration constant.)

A restricted constant expression may also contain the additional unary expression

defined *identifier*
or
defined(*identifier*

which evaluates to one if the identifier is currently defined in the preprocessor and zero if it is not.

All currently defined identifiers in restricted-constant-expressions are replaced by their token-strings (except those identifiers modified by **defined**) just as in normal text. The restricted constant expression will be evaluated only after all expressions have finished. During this evaluation, all undefined (to the procedure) identifiers evaluate to zero.

A control line of the form

#ifdef *identifier*

checks whether the identifier is currently defined in the preprocessor; i.e., whether it has been the subject of a **#define** control line. It is equivalent to **#ifdef(***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.

These constructions may be nested.

Line Control

For the benefit of other preprocessors which 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.

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**."

TYPES REVISITED

This part summarizes the operations which 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 m 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 which 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 the object to which the pointer points, namely 5-integer objects. The results are added and indirection applied to yield an array (of five integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript, the same argument applies again; this time the result is an integer.

Arrays in C are stored row-wise (last subscript varies fastest) and the first subscript in the declaration helps determine the amount of storage consumed by an array. Arrays play no other part in subscript calculations.

Explicit Pointer Conversions

Certain conversions involving pointers are permitted but have implementation-dependent aspects. They are all specified by means of an explicit type-conversion operator, see "Unary Operators" under"EXPRESSIONS" and "Type Names"under "DECLARATIONS."

A pointer may be converted to any of the integral types large enough to hold it. Whether an **int** or **long** is required is machine dependent. The mapping function is also machine dependent but is intended to be unsurprising to those who know the addressing structure of the machine. Details for some particular machines are given below.

An object of integral type may be explicitly converted to a pointer. The mapping always carries an integer converted from a pointer back to the same pointer but is otherwise machine dependent.

A pointer to one type may be converted to a pointer to another type. The resulting pointer may cause addressing exceptions upon use if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given size may be converted to a pointer to an object of a smaller size and back again without change.

For example, a storage-allocation routine might accept a size (in bytes) of an object to allocate, and return a **char** pointer; it might be used in this way.

```
extern char *alloc();
double *dp;

dp = (double *) alloc(sizeof(double));
*dp = 22.0 / 7.0;
```

The **alloc** must ensure (in a machine-dependent way) that its return value is suitable for conversion to a pointer to **double**; then the *use* of the function is portable.

The pointer representation on the PDP-11 corresponds to a 16-bit integer and measures bytes. The **char**'s have no alignment requirements; everything else must have an even address.

On the VAX-11, pointers are 32 bits long and measure bytes. Elementary objects are aligned on a boundary equal to their length, except that **double** quantities need be aligned only on even 4-byte boundaries. Aggregates are aligned on the strictest boundary required by any of their constituents.

The 3B 20 computer has 24-bit pointers placed into 32-bit quantities. Most objects are aligned on 4-byte boundaries. **Short**s are aligned in all cases on 2-byte boundaries. Arrays of characters, all structures, **int**s, **long**s, **float**s, and **double**s are aligned on 4-byte boundries; but structure members may be packed tighter.

CONSTANT EXPRESSIONS

In several places C requires expressions that evaluate to a constant: after **case**, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, casts to integral types, enumeration constants, and **sizeof**

expressions, possibly connected by the binary operators

+ - * / % & ¦ ^ << >> == != < > <= >= &&¦¦

or by the unary operators

- ~

or by the ternary operator

?:

Parentheses can be used for grouping but not for function calls.

More latitude is permitted for initializers; besides constant expressions as discussed above, one can also use floating constants and arbitrary casts and can also apply the unary & operator to external or static objects and to external or static arrays subscripted with a constant expression. The unary & can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

PORTABILITY CONSIDERATIONS

Certain parts of C are inherently machine dependent. The following list of potential trouble spots is not meant to be all-inclusive but to point out the main ones.

Purely hardware issues like word size and the properties of floating point arithmetic and integer division have proven in practice to be not much of a problem. Other facets of the hardware are reflected in differing implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are nuisances that must be carefully watched. Most of the others are only minor problems.

The number of **register** variables that can actually be placed in registers varies from machine to machine as does the set of valid types. Nonetheless, the compilers all do things properly for their own machine; excess or invalid **register** declarations are ignored.

Some difficulties arise only when dubious coding practices are used. It is exceedingly unwise to write programs that depend on any of these properties.

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
(expression)
primary (expression-list$_{opt}$ *)*
primary [expression]
primary . identifier
primary -> identifier

lvalue:
identifier
primary [expression]
lvalue . identifier
primary -> identifier
** expression*
(lvalue)

The primary-expression operators

() [] . ->

have highest priority and group left to right. The unary operators

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

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

binop:
* / %
\+ -
>> <<
< > <= >=
== !=
&
^
|
&&
||

The conditional operator groups right to left.

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

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

The comma operator has the lowest priority and groups left to right.

Declarations

declaration:
decl-specifiers init-declarator-list$_{opt}$;

decl-specifiers:
type-specifier decl-specifiers$_{opt}$
sc-specifier decl-specifiers$_{opt}$

sc-specifier:
auto
static
extern
register
typedef

type-specifier:
struct-or-union-specifier
typedef-name
enum-specifier
basic-type-specifier:
basic-type
basic-type basic-type-specifiers
basic-type:
char
short
int
long
unsigned
float
double
void

enum-specifier:
enum { *enum-list* }
enum *identifier* { *enum-list* }
enum *identifier*

enum-list:
enumerator
enum-list , *enumerator*

enumerator:
identifier
identifier = *constant-expression*

---

init-declarator-list:
init-declarator
init-declarator , *init-declarator-list*

init-declarator:
declarator initializer$_{opt}$

declarator:
identifier
(declarator)
** declarator*
declarator ()
declarator [constant-expression$_{opt}$ *]*

struct-or-union-specifier:
struct { *struct-decl-list* }
struct *identifier* { *struct-decl-list* }
struct *identifier*
union { *struct-decl-list* }
union *identifier* { *struct-decl-list* }
union *identifier*

struct-decl-list:
struct-declaration
struct-declaration struct-decl-list

struct-declaration:
type-specifier struct-declarator-list ;

struct-declarator-list:
struct-declarator
struct-declarator , struct-declarator-list

struct-declarator:
declarator
declarator : constant-expression
: constant-expression

initializer:
= expression
= { initializer-list }
= { initializer-list , }

initializer-list:
expression
initializer-list , initializer-list
{ initializer-list }
{ initializer-list , }

type-name:
type-specifier abstract-declarator

abstract-declarator:
empty
(abstract-declarator)
** abstract-declarator*
abstract-declarator ()
abstract-declarator [constant-expression$_{opt}$]

typedef-name:
identifier

Statements

compound-statement:
{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:
declaration
declaration declaration-list

statement-list:
statement
statement statement-list

statement:
compound-statement
expression ;
if *(expression) statement*
if *(expression) statement* **else** *statement*
while *(expression) statement*
do *statement* **while** *(expression) ;*
for *(*exp_{opt}*;*exp_{opt}*;*exp_{opt}*) statement*
switch *(expression) statement*
case *constant-expression : statement*
default *: statement*
break ;
continue ;
return ;
return *expression ;*
goto *identifier ;*
identifier : statement
;

External definitions

program:
external-definition
external-definition program

external-definition:
    function-definition
    data-definition

function-definition:
    decl-specifier$_{opt}$ *function-declarator function-body*

function-declarator:
    declarator (*parameter-list*$_{opt}$)

parameter-list:
    identifier
    identifier , *parameter-list*

function-body:
    declaration-list$_{opt}$ *compound-statement*

data-definition:
    extern *declaration* ;
    static *declaration* ;

Preprocessor

#define *identifier token-string*$_{opt}$
#define *identifier*(*identifier*,...)*token-string*$_{opt}$
#undef *identifier*
#include " *filename*"
#include <*filename*>
#if *restricted-constant-expression*
#ifdef *identifier*
#ifndef *identifier*
#else
#endif
#line *constant* " *filename*"

Chapter 3

A C PROGRAM CHECKER—"lint"

GENERAL

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

Usage

The **lint** command has the form:

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

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

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

-a Suppress messages about assignments of long values to variables that are not long.

-b Suppress messages about break statements that cannot be reached.

-c Only check for intra-file bugs; leave external information in files suffixed with **.ln**.

-h	Do not apply heuristics (which attempt to detect bugs, improve style, and reduce waste).
-n	Do not check for compatibility with either the standard or the portable **lint** library.
-0 *name*	Create a lint library from input files named **llib-l***name***.ln**.
-p	Attempt to check portability to other dialects of C language.
-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 or **lint** and the C compiler.

The **lint** program accepts certain arguments, such as:

```
-ly
```

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 **lint** library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined on a library file but are not used on a source file do not result in messages. The **lint** program 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 which contains descriptions of the programs which 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.

TYPES OF MESSAGES

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

Unused Variables and Functions

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

The **lint** program prints messages about variables and functions which are defined but not otherwise mentioned. An exception is

variables which are declared through explicit **extern** statements but are never referenced; thus the statement

```
extern double sin();
```

will evoke no comment if **sin** is never used. Note that this agrees with the semantics of the C compiler. In some cases, these unused external declarations might be of some interest and can be discovered by using the **–x** option with the **lint** command.

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The **–v** option is available to suppress the printing of messages about unused arguments. 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 program before the function. This has the effect of the **-v** option for only one function. Also, the comment:

```
/* VARARGS */
```

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

```
/* VARARGS2 */
```

will cause only the first two arguments to be checked.

There is one case where information about unused or undefined variables is more distracting than helpful. This is when **lint** is applied to some but not all files out of a collection which are to be loaded together. In this case, many of the functions and variables defined may not be used. Conversely, many functions and variables defined elsewhere may be used. The **–u** option may be used to suppress the spurious messages which might otherwise appear.

Set/Used Information

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

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement since the true flow of control need not be discovered. It does mean that **lint** can print messages about some programs which 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 which are set and never used. These form a frequent source of inefficiencies and may also be symptomatic of bugs.

Flow of Control

The **lint** program attempts to detect unreachable portions of the programs which it processes. It will print messages about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. An attempt is made to detect loops which can never be left at the bottom and to recognize the special cases **while(1)** and **for(;;)** as infinite loops. The **lint** program also prints messages about loops which cannot be entered at the top. Some valid

programs may have such loops which are considered to be bad style at best and bugs at worst.

The **lint** program has no way of detecting functions which are called and never returned. Thus, a call to **exit** may cause an 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 cannot be reached but it is not apparent to **lint**, the comment

```
/* NOTREACHED */
```

can be added at the appropriate place. This comment will inform **lint** that a portion of the program cannot be reached.

The **lint** program will not print a message about unreachable **break** statements. Programs generated by **yacc** and especially **lex** may have hundreds of unreachable **break** statements. The **−O** option in the C compiler will often eliminate the resulting object code inefficiency. Thus, these unreached statements are of little importance. There is typically nothing the user can do about them, and the resulting messages would clutter up the **lint** output. If these messages are desired, **lint** can be invoked with the **−b** option.

Function Values

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

Locally, within a function definition, the appearance of both

```
return( expr );
```

and

```
return ;
```

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

function *name* contains 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.

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

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

Type Checking

The **lint** program 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 program 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. The **lint** program 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. It seems harsh for **lint** to continue to print messages about this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The **–c** flag controls the printing of comments about casts. When **–c** is in effect, casts are treated as though they were assignments subject to messages; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

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. Since there are a number of legitimate reasons for assigning **longs** to **ints**, the detection of these assignments is enabled by the **–a** option.

Strange Constructions

Several perfectly legal, but somewhat strange, constructions are detected by **lint**. The messages hopefully encourage better code quality, clearer style, and may even point out bugs. The **–h** option is used to supress 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. The **lint** program 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 statement

```
if( x&077 == 0 ) ...
```

or

```
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.

Finally, when the **–h** option has not been used, **lint** prints messages about variables which are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal but is considered to be bad style, usually unnecessary, and frequently a bug.

Old Syntax

Several forms of older syntax are now illegal. These fall into two classes - assignment operators and initialization.

The older forms of assignment operators (e.g., =+, =-, ...) could cause ambiguous expressions, such as:

```
a =-1 ;
```

which could be taken as either

```
a =- 1 ;
```

or

```
a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer and preferred operators (e.g., +=, -=, ...) have no such ambiguities. To encourage the abandonment of the older forms, **lint** prints messages about these old-fashioned operators.

A similar issue arises with initialization. The older language allowed

```
int x 1 ;
```

to initialize *x* to 1. This also caused syntactic difficulties. For example, the initialization

```
int x ( -1 ) ;
```

looks somewhat like the beginning of a function definition:

```
int x ( y ) { . . .
```

and the compiler must read past *x* in order to determine the correct meaning. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

```
int x = -1 ;
```

This is free of any possible syntactic ambiguity.

Pointer Alignment

Certain pointer assignments may be reasonable on some machines and illegal on others due entirely to alignment restrictions. The **lint** program 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 (like the PDP-11) 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 which have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C language on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler. In fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect and also used elsewhere in the same expression, the result is explicitly undefined.

The **lint** program checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++];
```

will cause **lint** to print the message

```
warning: i evaluation order undefined
```

in order to call attention to this condition.

Chapter 4

PROGRAMMING PROCEDURES FOR UNIX SYSTEM V/AT

OVERVIEW

This chapter is intended to familiarize C and FORTRAN programmers with the special features of UNIX System V Release 2 iAPX 286 Version 1*. You should become familiar with the contents of this chapter before referring to the remainder of the Programming Guide.

The following is an overview of the contents of this chapter.

The iAPX SEGMENTED MEMORY ARCHITECTURE part briefly describes the Intel iAPX 286 segmented memory architecture.

The MEMORY MODELS part is an introduction to the UNIX System V/286 memory model system.

The SELECTING A MEMORY MODEL part describes how to select a memory model and how to specify a memory model when using the compile commands.

The MEMORY MODEL RESTRICTIONS AND PORTABILITY CONSIDERATIONS part deals with memory model system topics, such as memory allocation, pointer manipulation, C and FORTRAN language changes, and object file format translation.

The ERROR MESSAGES part lists the error messages that can occur when using memory models.

---

* Referred to as UNIX System V/286

The SUMMARY OF MEMORY MODEL FEATURES part summarizes the features of the various memory models.

THE iAPX SEGMENTED MEMORY ARCHITECTURE

The iAPX 286 processor can run in either Real Address Mode or Protected Virtual Address Mode. In Real Address Mode, the iAPX 286 functions like an iAPX 86 or iAPX 186 to allow existing programs written for these processors to be run. In Protected Virtual Address Mode, the iAPX 286's full capabilities -- such as virtual addressing, data protection, task concurrency, and memory management -- can be used. UNIX System *V/AT* executes in the iAPX 286 Protected Virtual Address Mode only.

In Protected Virtual Address Mode, the iAPX 286 treats logical memory as a large virtual space, rather than a set of directly accessible physical addresses. This virtual space is viewed by the programmer as a collection of linear subspaces, called segments. Each segment is a logical unit of contiguous memory that ranges in size from 1 to 64k bytes. These segments are accessed through the following 16-bit segment registers:

CS code segment register

DS data segment register

SS stack segment register

ES extra data segment register

The code segment (CS) register provides access to segments containing the currently executing sequence of instructions. The data segment (DS) register provides access to segments that store dynamic data, initialized data, and uninitialized data. The stack segment (SS) register provides access to the segment that contains stack data. The extra data segment (ES) register is used as an additional data

segment register to move data from one section of memory to another.

Each segment register contains the logical base address of a memory segment. This logical address is translated into the physical base address of that segment in memory. Depending on how the segment registers are used, a program can address approximately 8192 (8k) segments of memory, four segments at a time.

Refer to the Introduction to the iAPX 286 and the iAPX 286 Programmer's Reference Manual for more detailed information on the architecture of the iAPX 286.

MEMORY MODELS

In UNIX System *V/AT*, there are three segment-register/pointer configurations you can use to compile your program. These configurations are called memory models. When you compile your program, you need to specify one of the following memory models:

small model (default)

large model

huge model

The terms used in the memory model discussions are defined below.

UNINITIALIZED DATA: The uninitialized data area (also called BSS) holds global, uninitialized data. The size of this area is fixed at link time. This area is set to zero when a program begins execution.

INITIALIZED DATA: The initialized data area holds global, initialized data. The size of this area is fixed at link time.

DYNAMIC DATA: (also called the sbreak area) holds dynamically allocated data (data allocated with *malloc, calloc,* etc., during program execution). The dynamic data area starts with zero space, and expands (or shrinks) during execution, as directed by the program.

STACK: the stack area holds local, temporary data. In the large and huge models, the default initial size of the stack segment is 8k bytes, which grows as needed to a maximum of 64k bytes. In the small model, the stack size is fixed at link time (default size is 8k bytes) and shares the same segment as the data.

The initial stack size can be adjusted for any of the three models. See "ADJUSTING STACK SIZE" in SELECTING A MEMORY MODEL for details.

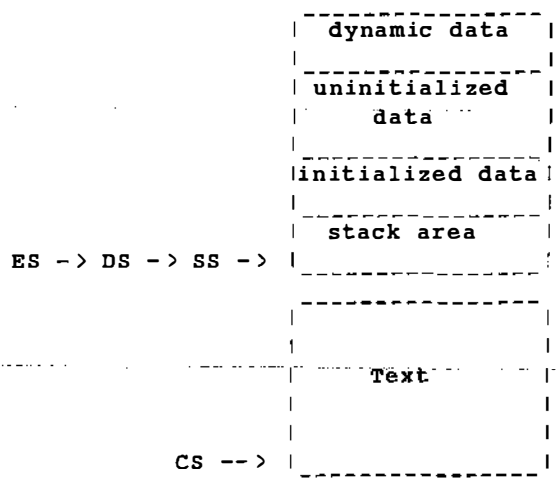
TEXT: the code segment register points to the current executing sequence of instructions. In the UNIX system, this area is referred to as the text area and is stored in one or more text segments, depending on the memory model used and the size of the program's text. The size of the text segment (also called the code segment) is fixed at link time.

SMALL MODEL

The small model provides the simplest UNIX programming environment. Of the three memory models discussed in this part, the small memory model is the fastest and uses the least amount of memory.

The small model uses one text segment and one data segment. Since the segments do not change during program execution, short pointers are used. Text is contained in its own segment, while the data segment is shared between data and the program stack. The area allocated for the stack defaults at 8k bytes, which can be adjusted by using a linker option (see "ADJUSTING STACK SIZE" in SELECTING A MEMORY MODEL).

The small memory model is used by default when no memory model is specified.



LARGE MODEL

The large memory model allows multiple text and data segments. All pointers (data and text) are 32 bits. Data arrays are limited to less than 64k bytes each. The only restriction on the amount of data and text that may be stored is the maximum size of physical memory available.

Large Model Data Segments

Separate segments exist for dynamic, initialized, and uninitialized data. All data segments are accessed through the data segment (DS) or extra segment (ES) registers. Shared memory system calls are supported to allow several processes to access the same data segment. All initialized data in a single file must fit into a single segment. If no uninitialized data exists, no uninitialized data segment is created.

Large Model Text Segment

One or more text segments may exist. More than one subroutine can be stored in a text segment. A subroutine or a module, however, cannot be larger than a segment. The linker determines the memory requirements for each program's text and stores as much of it as possible in a single segment.

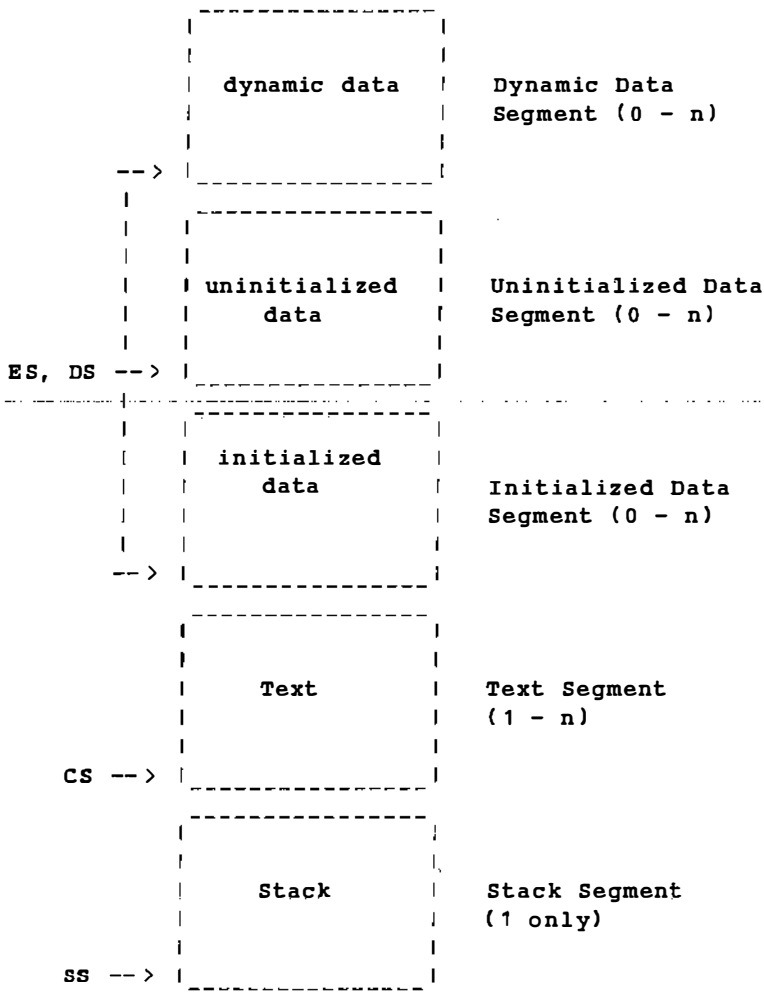
For multiple file programs, if a file's text is too large to be stored in the remainder of a segment, a new text segment is allocated. Suppose, for example, that four files are being linked. The first file has 20k bytes of text; the second file has 10k bytes of text; the third file has 10k bytes of text; the fourth file has 50k bytes of text. Since the first three files have a total of 40k bytes of text, the text for the first three files will be stored in a single memory segment. The fourth file, however, requires 50k bytes for its text, which is more than the 24k bytes of free space remaining in the existing text segment. Therefore, in order to keep the text for the fourth file from straddling two segments, a new text segment will be allocated for the fourth file.

You can use the *ld* command with the -m option to produce a memory map of the linked output.

Large Model Stack Segment

The stack is contained in its own memory segment, which defaults at 8k bytes and grows dynamically as needed up to a maximum size of 64k bytes. Since the stack is limited to 64k bytes, automatic arrays cannot be larger than 64k bytes.

See " Adjusting Stack Size" in SELECTING A MEMORY MODEL for information on adjusting the initial stack size.


dynamic data
Dynamic Data Segment (0 - n)
uninitialized data
Uninitialized Data Segment (0 - n)
ES, DS
initialized data
Initialized Data Segment (0 - n)
Text
Text Segment (1 - n)
CS
Stack
Stack Segment (1 only)
SS

HUGE MODEL

The huge memory model, like the large model, uses multiple text and data segments and a single stack segment. The huge model's basic design and operation is identical to that of the large model. However, unlike the large model, the huge model will compile programs that have huge arrays (arrays larger than 64k bytes).

Each time pointer arithmetic is done in the huge memory model, the compiler checks to see if a segment boundary has been crossed. If a pointer is adjusted to a value that exceeds the boundary of a segment, the compiler will determine the break-off point at the end of that segment and adjust the pointer's segment selector and offset to the correct location in another segment.

When you use the huge model, you must bear the following in mind:

- Each element of an array must be less than 64k bytes.
- Since the stack is limited to a maximum size of 64k bytes, automatic arrays cannot be larger than 64k bytes.
- Huge arrays cannot be used inside structures.

MODEL COMPARISON

This section compares the organization of the UNIX System *VIAT* memory models and discusses how the memory models affect program performance.

The following table summarizes each memory model's characteristics.

Model	Stack	Data			Text	
	segment usage	segment usage	pointer size	array size	segment usage	pointer size
Small	share 1	share 1	16 bits	<64k	1	16 bits
Large	1	n	32 bits	<64k	n	32 bits
Huge	1	n	32 bits	<=>64k	n	32 bits

Table Key

```
      n:  multiple segments are available

share 1:  stack and data share the same segment.

   <64k:  data arrays must be less than 64k bytes.

 <=>64k:  data arrays can be less than, equal to,
          or larger than 64k bytes.
```

Execution Speed

The memory model you use when compiling your program determines how often the iAPX 286 segment registers are loaded when that program is executed. A program that must reload segment registers during execution will execute more slowly than one that does not have to reload segment registers during execution.

Of the three memory models, the small memory model will provide the fastest execution environment. This is because the small model provides only one 64k memory segment each for the program's data and code, making it necessary for the segment registers to be loaded only once, prior to execution.

In contrast to the small model, the large and huge models allow a program to access multiple data and text segments. A program compiled using either the large or huge model may execute more

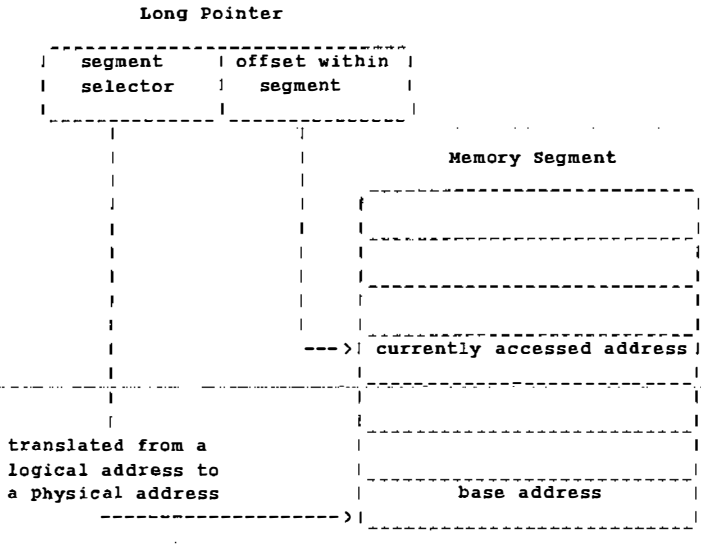
slowly, since the processor may need to reload segment registers frequently while the program is executed.

Pointer Size

The memory architecture of the iAPX 286 allows you to use either short (16-bit) or long (32-bit) pointers to specify locations in memory. The memory model you select determines the size of the pointers your program uses to access memory.

Because the small model is limited to single 64k-byte data and text segments, programs compiled using this model can use short pointers to specify the offset address within each memory segment.

Programs compiled using the large or huge memory models must use long pointers in order to identify both the memory segment and the offset within that segment. The upper 16 bits of a long pointer contain the segment selector, which is the logical address of the segment. At runtime, the segment selector is loaded into the appropriate segment register and is then translated into a physical address to locate the segment in memory. The lower 16 bits of a long pointer contain the offset address within the segment identified by the segment selector.



HOW SEGMENTS ARE ALLOCATED

After a program is compiled, the assembler translates all of the assembly code into relocatable machine code. This relocatable code contains unresolved references to the segment in which each symbol is to be located. The linker uses the relocation and symbol tables to determine the segment and segment references for each symbol.

SELECTING A MEMORY MODEL

When selecting a memory model to compile your program with, you should weigh the memory requirements of your program against its desired performance. The small model provides a faster execution environment than the large and huge models. However, in order to use the small model, the sum total of your program's text, data, and stack cannot exceed 128k bytes (one text segment of 64k bytes and one data segment of 64k bytes). If you want to use the large memory model, your arrays cannot exceed 64k bytes. If your program arrays exceed 64k bytes, you must use the huge model. If your program uses shared memory system calls, it must use the large or huge model.

If you are not sure about the memory requirements of your program, you should first compile and link your program using the small model. If you receive a text or data segment overflow error message, try using the large model. If your program has one or more arrays larger than 64k bytes and you attempt to compile it using tbe small or large models, you will receive an error message stating that your array is too large for the selected model. When this occurs, compile your program using the huge model.

COMMAND OPTIONS

When you execute the compile command, you select the desired memory model by including one of the following options:

-Ms specifies the small memory model (default)

-Ml specifies the large memory model

-Mh specifies the huge memory model

For example, to compile and link a C program named " prog1.c" using the large memory model, type:

```
cc -Ml prog1.c -o prog1
```

Large programs are often divided into smaller, more manageable files, called modules. If your program consists of several separate modules, you must compile each module using the same memory model. Otherwise, the modules will not link properly. Programs sharing data files that contain pointers must also be compiled using the same memory model.

If you attempt to link files compiled with incompatible memory models, the linker will return an error stating that incompatible models are being used.

DEFAULT MODEL

If you do not specify a memory model at compile time, your program is compiled using the small model, which will compile most programs. If the program is too large to be compiled using the small model, you will receive a segment overflow error message and a message suggesting that you use a larger model.

ADJUSTING STACK SIZE

It is possible to adjust the initial stack size in any of the three memory models. When using the small model, the stack size can be made larger (at the expense of the dynamic data area) to support a larger stack, or made smaller in order to increase the dynamic data area. You can also adjust the initial stack size in the large or huge model to a value between 512 bytes and 64k bytes. This adjustment is useful when you know your program's stack requirements and you want to bypass the runtime overhead of enlarging the stack.

You can determine your program's stack usage by:

- Compiling your program using the *cc*(1) command with the -p option. When the program is run a profile file, mon.out by default, is created.

- Displaying the profile file using *prof*(1) with the -k option. This will cause information on the stack usage of your program to be appended to the end of the output of the profile file.

If you wish to adjust the initial stack size for your program, add the following option to the link command:

-k <size of stack>

where <size of stack> is the number of bytes you want to adjust your stack size to. This value will be rounded up to the nearest multiple of 512 bytes. Therefore, the minimum you can adjust your initial stack size to is 512 bytes.

For example, if you are compiling prog1 using the large model and you know the stack in prog1 will require between 8k and 12k of memory, you can set your stack size to 12k by entering:

```
cc -Ml -k 12000 prog1.c -o prog1
```

This value will be rounded up to 12288 (12k bytes), which is the nearest multiple of 512.

Note that the stack size adjustment is a linker option that is passed on to the linker by the compiler. If you wish to change the stack size of a program without recompiling the program, you can relink the program using the -k option.

LIBRARIES

Methods of accessing global variables, function calls, and returns in general are different for programs compiled under different memory models. Because of this, a separate program library is required for each memory model.

Appropriately compiled C and FORTRAN program libraries exist for each of the three memory models. The linker automatically selects the library that corresponds to the memory model used.

If you build a new library, make sure it has been properly compiled using the appropriate memory model. If the library is for use with all three memory models, you must create three copies of the library and compile each copy with a different memory model.

After a library is compiled under all three memory models, each compiled copy of the library must be placed in the appropriate default directory. For example, if the library file, newlib, is compiled and archived for each memory model, you would place each compiled copy of the file into its respective default directory. These default directories are:

/usr/lib/small/libnewlib.a

/usr/lib/large/libnewlib.a

/usr/lib/huge/libnewlib.a

When linking library files with your object files, use the -l option followed by the name of the library file. The linker will determine which memory model the object files were compiled under and link them with the correctly compiled version of the library file. For example, when linking the library file: libnewlib.a with the object files: x.o, y.o, and z.o, you would enter:

```
ld x.o y.o z.o -lnewlib
```

The correctly compiled copy of libnewlib.a will be selected by the linker and linked with the object files.

MEMORY MODEL RESTRICTIONS AND PORTABILITY CONSIDERATIONS

One of the main objectives in developing UNIX System *V/AT* has been to support as much existing source code as possible and to quickly detect code that cannot be supported. The following sections describe some of the restrictions that exist on this system and what you can do to adapt your code to run on it. Most of the restrictions described apply only to the large or huge memory model environments.

UNIX SYSTEM V/286 MEMORY ALLOCATION

Because the iAPX 286 uses a segmented, non-contiguous memory architecture, special care must be taken when allocating memory space. This section describes what you should be aware of when porting existing C programs to UNIX System V/286 and how to write new C programs to use the memory architecture of the iAPX 286 most efficiently.

Porting Existing C Programs to UNIX System V/AT

The system calls *brk*(2) and *sbrk*(2) are used to dynamically change the amount of space allocated in the calling process's data segment. When you use the *brk*(2) system call, you specify the next address beyond the area you wish to allocate. This increases the break value which, in turn, increases the amount of allocated space. When you use *sbrk*(2) you need only specify the amount of space you wish to allocate. The system call *sbrk*(2) will change the break value to allocate the specified amount of space.

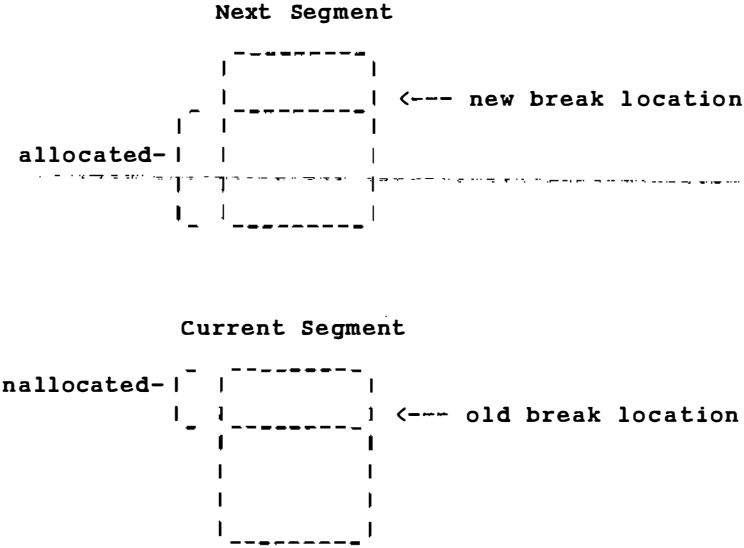
When the large or huge memory models are used, all *brk*(2) and *sbrk*(2) requests are rounded up to a 512-byte boundary.

When porting existing code that calls the *brk*(2) system call directly, you must be aware of the following:

- *brk*(2) takes a character pointer parameter, which is 16 bits in the small memory model and 32 bits in the large and huge memory models.
- In the huge memory model, *brk*(2) can be used to allocate multiple memory segments.
- *brk*(2) will return an error if given an address that is in the stack, part of the space between segments, or any other invalid address.
- You cannot *brk*(2) to an unallocated area between two allocated segments.
- If you give *brk*(2) an address outside of the segment containing the current break point, the remainder of the current segment

will not be allocated. Instead, the allocation will begin from the beginning of the segment containing the address specified by *brk*(2).

The following figure illustrates this operation.



When porting existing code that uses the *sbrk*(2) system call, you must be aware of the following:

- *sbrk*(2) takes a 16-bit signed integer parameter. Therefore, the most *sbrk*(2) can allocate or free is 32k of memory.

- When using the large or huge memory model, *sbrk*(+n) begins allocating memory from the start of the next memory segment and returns the starting address of that segment; *sbrk*(0) always returns the starting address of the next memory segment.

- When using *sbrk*(-n) to deallocate memory in the large or huge memory model, you can deallocate multiple memory segments.

As an illustration of the operation of *sbrk* in UNIX System *V/AT* , consider the following example: Memory is allocated using *sbrk*(+24) and *sbrk*(+24). Each *sbrk*(+24) request will create a new segment. And in each of these segments the requested allocation space of 24 bytes will be rounded up to 512 bytes. This will cause two segments, each containing 512 bytes, to be allocated. If *sbrk*(-48) is given, the requested deallocation of 48 bytes will be rounded up to 512 bytes. This will cause the last allocated 512-byte segment to be deallocated; leaving the first 512-byte segment allocated.

Because *sbrk*(2) can access only 32k bytes of memory, many programs are written to bypass this limitation by using *sbrk*(0) to return the next available location in memory and *brk*(2) to allocate the desired amount of memory. When these programs are compiled using the large or huge memory model, you must keep in mind that *sbrk*(0) returns the beginning address of the next memory segment, rather than the beginning address of the next empty space in the current memory segment.

The following table summarizes the actions of *brk*(2), and *sbrk*(2) in the different memory models (S = small, L = large, H = huge).

Operation	Model	Action
sbrk(0)	S	Returns current break value.*
	L,H	Returns starting address of NEXT data segment.
sbrk(+incr)	S	Allocates incr bytes in current segment.
	L,H	Allocates incr bytes in next data segment (space from old break value† to end of old segment is not allocated).
	S,L,H	Returns the same value as sbrk (0).
sbrk(-incr)	S	Frees incr bytes in current segment.
	L,H	Frees incr bytes from as many segments as needed.
	S,L,H	Returns the same value as sbrk (0).
brk(endds) (current segment)	S,L,H	Sets break value to endds and allocates or frees memory to that point.
brk(endds) (previous segment)	L,H	Sets break value to endds and frees memory between old break value and endds. Endds must be an allocated location. Can free multiple segments.
brk(endds) (new segment)	L,H	Sets break value to endds in next segment.
	L	Can allocate up to one segment.
	H	Can allocate multiple segments.
	L,H	Space from old break value to end of old segment is not allocated.

* " Break value" is the address of the first data location beyond the end of allocated data.

† " Old break value" is the break value previous to the execution of the current operation.

Writing New C Programs

When writing new code to be run under UNIX System V, use *malloc*(3c) and *calloc*(3c), instead of *brk*(2) and *sbrk*(2), to allocate memory. The calls *malloc*(3c) and *calloc*(3c), take unsigned arguments that allow up to 64k bytes of memory to be allocated, making them more adaptable to the iAPX 286 segmented memory structure than *brk*(2) and *sbrk*(2). In addition, *malloc*(3c) and *calloc*(3c) are more portable.

SIZEOF OPERATOR

When C programs are executed under the huge model, the *sizeof* operator causes a compiler error if the operand is equal to or greater than 64k bytes. For example, if you declare the array, array1, in the following manner:

```
#define ARRAY_SIZE 100000

int array1[ARRAY_SIZE];
```

and then attempt to determine the size of array1 by

```
int size;

size = sizeof array1
```

the compiler will return an error because the size of array1 is greater than 64k bytes. If this occurs, you must replace the *sizeof* operation with a constant expression. This can be done by converting the above operation to determine the size of an array element and multiplying the result by the number of elements in the array. For example, the above operation can be converted to:

```
long size;

size = ARRAY_SIZE * sizeof (int);
```

where " ARRAY_SIZE" is the number of elements in array1; " int" is a single element in array1, and " size" is the variable that stores the size of array1.

Integer arithmetic in C requires that one operand be long if the answer is to be long. In the above example, ARRAY_SIZE is made long by the 100k assignment. When converting code in this way, you must declare the variable storing the size of the array (" size" , in this example) as a long integer. Otherwise, if the variable is declared as an integer and the operation returns a value greater than 64k bytes, the variable storing the size of the array will be too small to contain the calculated value.

FORTRAN EQUIVALENCE STATEMENTS

In FORTRAN, free storage space can be maximized by using equivalence statements to allow two or more arrays to share the same storage area. Because the huge memory model does not exist for FORTRAN, FORTRAN dimension statements cannot use operands larger than 64k bytes and the total size of the combined storage of the arrays in equivalence statements cannot exceed 64k bytes.

For example, if you have two arrays, A and B, that are declared as follows:

```
INTEGER A(20000)

INTEGER B(20000)
```

you can use the following equivalence statement:

```
EQUIVALENCE (A(10001), B(1))
```

This is valid because the total size of the combined array storage adds up to 60000 bytes, which is less than 64k (65536) bytes. The total memory used is determined by the following equation: 2*(10000 + 20000), which is the offset in Array A added to the total size of Array B and multiplied by 2. (The sum resulting from combining the two

arrays is multiplied by 2 because integers in the iAPX 286 are 2 bytes.)

However, if you were to use the statement:

```
EQUIVALENCE (A(19001), B(1))
```

the total size of the combined array storage would be 78000 bytes (2*(19000 + 20000)), which exceeds 64k bytes. This equivalence statement is, therefore, invalid.

POINTER OPERATIONS

Integers and Pointers

Much of the existing code for the UNIX system was written under the assumption that integers and pointers are the same size. When using the large or huge model, avoid using integers as pointers, because 32-bit pointers are incompatible with 16-bit integers.

For example, if a 32-bit pointer is assigned to an integer, the segment selector portion of the pointer is not transferred in the assignment. Then, if the integer is assigned to a pointer, that pointer will not be valid.

You can add or subtract pointers and integers as long as the operation does not change the segment selector portion of the pointer. If, when using the large or huge memory model, you find it necessary to assign an arbitrary integer to a pointer, you can use a long integer. A fault will occur, however, if the assignment creates an invalid address.

Subtracting Pointers

In C, the difference between two pointers is defined as an integer. In UNIX System *VAT* , integers are 16 bits and, in the large or huge memory model, pointers are 32 bits. When the large or huge memory model is used, an integer is only adequate to describe the difference

between two pointers when those pointers reside in the same segment.

If it is necessary to subtract pointers, you should be aware of the size and boundaries of the segment to ensure that only pointers that access the same segment are used. When using the huge memory model, pointers existing in different segments can be subtracted if those segments were allocated using the same *malloc*(3c). For example, it is permitted to subtract pointers to elements in the same huge array.

Incrementing Long Pointers

When manipulating long pointers in the large memory model, you must be aware of the boundaries of the segment you are accessing. Incrementing a long pointer in order to access the next address should be avoided when using the large model, unless you are sure you will not increment to an address outside of the segment.

In the huge memory model it is allowable to increment long pointers across segment boundaries.

SHARED MEMORY SYSTEM CALLS

Shared memory system calls that allow more than one process to access the same memory segment are not permitted in programs compiled using the small model. However, these system calls will work in programs compiled using the large and huge models.

PROGRAMMING PROCEDURES FOR UNIX SYSTEM V/AT

ERROR MESSAGES

COMPILER ERRORS

The following error messages may occur when you compile your program:

DATA SEGMENT OVERFLOW

Occurs at link time if your program has more data than the current model can allocate. Try recompiling the program using the large model.

TEXT SEGMENT OVERFLOW

Occurs at link time if your program has more text than the current model can allocate. Try recompiling the program using the large model.

ARRAY OVERFLOW

Occurs if your program arrays are too large for the model you are using. Try recompiling the program using the huge model.

SIZEOF OVERFLOW

Occurs if the operand of the *sizeof* operator is equal to or larger than 64k bytes. If this occurs, you can replace the sizeof operation with a constant expression (see the " SIZEOF OPERATOR" section in MEMORY MODEL RESTRICTIONS AND PORTABILITY CONSIDERATIONS for details).

RUNTIME ERROR

The following error messages may occur at runtime.

SEGMENTATION VIOLATION or BUS ERROR

Several conditions could cause a segmentation violation error. This error can occur if your program attempts to use an invalid pointer, if the program stack grows beyond the limitations of the model you are using. This error also occurs frequently as a result of a bad pointer or an out-of-bounds array reference. If you receive this error message when using the small model, determine what your stack requirements are; then either adjust the stack size or recompile your program using the large model.

SUMMARY OF MEMORY MODEL FEATURES

The following table compares the features of the UNIX System V/286 memory models.

	Small	Large	Huge
Pointers and Integers Compatible	X		
Large* text area available		X	X
Large* data area available		X	X
Huge† arrays available			X
Modified sbrk system call		X	X
Shared memory available		X	X
FORTRAN support	X	X	

* Greater than or equal to 64k bytes.

† Greater than 64k bytes.

Chapter 5

A PROGRAM FOR MAINTAINING COMPUTER PROGRAMS (make)

GENERAL

In a programming project, a common practice is to divide large programs into smaller pieces that are more manageable. The pieces may require several different treatments such as being processed by a macro processor or sophisticated program generators (e.g., **Yacc** or **Lex**). The project continues to become more complex as the output of these generators are compiled with special options and with certain definitions and declarations. A sequence of code transformations develops which is difficult to remember. The resulting code may need further transformation by loading the code with certain libraries under control of special options. Related maintenance activities also complicate the process further by running test scripts and installing validated modules. Another activity that complicates program development is a long editing session. A programmer may lose track of the files changed and the object modules still valid especially when a change to a declaration can make a dozen other files obsolete. The programmer must also remember to compile a routine that has been changed or that uses changed declarations.

The " make" is a software tool that maintains, updates, and regenerates groups of computer programs.

A programmer can easily forget

- Files that are dependent upon other files.
- Files that were modified recently.
- Files that need to be reprocessed or recompiled after a change in the source.
- The exact sequence of operations needed to make an exercise a new version of the program.

The many activities of program development and maintenance are made simpler by the **make** program.

The **make** program provides a method for maintaining up-to-date versions of programs that result from many operations on a number of files. The **make** program can keep track of the sequence of commands that create certain files and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of a program, the **make** command creates the proper files simply, correctly, and with a minimum amount of effort. The **make** program also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

The basic operation of **make** is to

- Find the name of the needed target file in the description.
- Ensure that all of the files on which it depends exit and are up to date.
- Create the target file if it has not been modified since its generators were modified.

The descriptor file really defines the graph of dependencies. The **make** program determines the necessary work by performing a depth-first search of the graph of dependencies.

If the information on interfile dependencies and command sequences is stored in a file, the simple command

```
make
```

is frequently sufficient to update the interesting files regardless of the number edited since the last **make**. In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the **make** command than to issue even one of the needed operations, so the typical cycle of program development operations becomes

think - edit - **make** - test . . .

The **make** program is most useful for medium-sized programming projects. The **make** program does not solve the problems of maintaining multiple source versions or of describing huge programs.

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


```
# Description file for the Make command

p = lp
FILES = Makefile version.c defs main.c doname.c misc.c
        files.c dosys.c gram.y lex.c gcos.c
OBJECTS = version.o main.o doname.o misc.o files.o
          dosys.o gram.o
LIBES= -lS
LINT = lint -p
CFLAGS = -O

make: $(OBJECTS)
        cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
        size make

$(OBJECTS): defs
gram.o: lex.c

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

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

print: $(FILES)         # print recently changed files
        pr $? | $P
        touch print
```

```
test:
        make -dp | grep -v TIME >1zap
        /usr/bin/make -dp | grep -v TIME >2zap
        diff 1zap 2zap
        rm 1zap 2zap

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

arch:
        ar uv /sys/source/s2/make.a $(FILES)
```

The **make** program usually prints out each command before issuing it.

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

```
cc -O -c version.c
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 version.o main.o doname.o misc.o files.o dosys.o
   gram.o -lS -o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars were mentioned by name in the description file, **make** found them using its suffix rules and issued the needed commands. The string of digits results from the **size make** command. The printing of the command line itself was suppressed by an @ sign. The @ sign on the **size** command in the description file suppressed the printing of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The "print" entry prints only the files changed since the last **make print** command. A zero-length file *print* is maintained to keep track of the time of the printing. The **$?** macro in the command line then picks up only the names of the files changed since *print* was touched. The printed output can be sent to a different printer or to a file by changing the definition of the **P** macro as follows:

```
make print " P=  cat >zap"
```

BASIC FEATURES

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

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 **lS** 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*, but that *z.c* does not. That is, *x.c* and *y.c* have the line

```
#include " defs"
```

The following text describes the relationships and operations:

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

x.o  y.o :  defs
```

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

```
make
```

would perform the operations needed to recreate *prog* after any changes had been made to any of the four source files *x.c, y.c, z.c,* or *defs.*

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

- A user-supplied description file
- File names and "last-modified" times from the file system
- Built-in rules to bridge some of the gaps.

In the example, 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.* From the file system, **make** discovers that there are three "*.c*" files corresponding to the needed "*.o*" files and uses built-in information on how to generate an object from a source file (i.e., issue a "cc -c" command).

By not taking advantage of **make**'s innate knowledge, the following longer descriptive file results.

```
prog : x.o y.o z.o
       cc x.o y.o z.o -lS -o prog
x.o : x.c defs
       cc -c x.c
y.o : y.c defs
       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, all of the files are current, and the command

```
make
```

announces this fact and stops. If, however, the *defs* file has been edited, *x.c* and *y.c* (but not *z.c*) is recompiled; and then *prog* is created from the new "*.o*" files. 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 recompile *x.o* if *x.c* or *defs* had changed.

If the file exists after the 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. 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. Thus, an entry "save" might be included to copy a certain set of files, or an entry "cleanup" might be used to throw away unneeded intermediate files. In other cases, one may 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.

The **make** program has a simple macro mechanism for substituting in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the name by a dollar sign. Macro names longer than one character must be parenthesized. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations:

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

The last two invocations are identical. A $$ is a dollar sign.

The $*, $@, $?, and $< are four special macros which change values during the execution of the command. (These four macros are described in the part "DESCRIPTION FILES AND SUBSTITUTIONS".) The following fragment shows assignment and use of some macros:

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

The **make** command loads the three object files with the **lS** library. The command

```
make " LIBES= -ll -lS"
```

loads them with both the Lex (-ll) and the standard (-lS) libraries since macro definitions on the command line override definitions in the description. Remember to quote arguments with embedded blanks in UNIX software commands.

DESCRIPTION FILES AND SUBSTITUTIONS

A description file contains the following information:

- macro definitions
- dependency information
- executable commands.

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. 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.

A macro definition is a line containing an equal sign not 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 -lS
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as the macro's value.

Macro definitions may also appear on the **make** command line while other lines give information about target files. The general form of an entry is

```
target1 [target2 . .] :[:] [dependent1 . .] [; commands] [# . .]
[(tab) commands] [# . . .]
. . .
```

Items inside brackets may be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. Shell metacharacters such as "*" and "?" are expanded. 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 or not including a new line.

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type. For the usual 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 creation rule may be invoked. In the double-colon case, a command sequence may be associated with each 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. This detailed form is of particular value in updating archive-type files.

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 or if the command line 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** flags have 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. Some UNIX software commands return meaningless status. 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 file names. 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.

COMMAND USAGE

The **make** command takes macro definitions, flags, description file names, and target file names as arguments in the form:

make [flags] [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 flag arguments are examined. The permissible flags 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.

−d	Debug mode. Print out detailed information on files and times examined.
−f	Description file name. The next argument is assumed to be the name of a description file. A file name of "-" denotes the standard input. If there are no "*-f*" arguments, the file named *makefile* or *Makefile* in the current directory is read. The contents of the description files override the built-in rules if they are present.

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 files that does not begin with a period is "made".

SUFFIXES AND TRANSFORMATION RULES

The **make** program does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the **−r** flag is used, the internal table is not used.

The list of suffixes is actually the dependency list for the name ".SUFFIXES". The **make** program searches for a file with any of the suffixes on the list. If such a file exists and if there is a transformation rule for that combination, **make** transforms a file with one suffix into a file with another suffix. The transformation rule names are the concatenation of the two 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 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 his own 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. The following is an excerpt from the default rules file:


```
.SUFFIXES : .o .c .e .r .f .y .yr .ye .l .s
YACC = yacc
YACCR = yacc -r
YACCE = yacc -e
YFLAGS =
LEX = lex
LFLAGS =
CC = cc
AS = as -
CFLAGS =
RC = ec
RFLAGS =
EC = ec
EFLAGS =
FFlags =
.c.o :
        $(CC) $(CFLAGS) -c $<
.e.o .r.o .f.o :
        $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.s.o :
        $(AS) -o $@ $<
.y.o :
        $(YACC) $(YFLAGS) $<
        $(CC) $(CFLAGS) -c y.tab.c
        rm y.tab.c
        mv y.tab.o $@
.y.c :
        $(YACC) $(YFLAGS) $<
        mv y.tab.c $@
```

IMPLICIT RULES

The **make** program uses a table of interesting 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
.e	Efl source file
.r	Ratfor source file
.f	Fortran source file
.s	Assembler source file
.y	Yacc-C source grammar
.yr	Yacc-Ratfor source grammar
.ye	Yacc-Efl source grammar
.l	Lex source grammar.

Figure 2-1 summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.

If the file *x.o* were needed and there were an *x.c* in the description or directory, the *x.o* file would be compiled. If there were also an *x.l*, that grammar would be run through Lex before compiling the result. However, if there were no *x.c* but there were an *x.l*, **make** would discard the intermediate C language file and use the direct link as shown in Figure 2-1.

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**, **RC**, **EC**, **YACC**, **YACCR**, **YACCE**, and **LEX**. The command

```
make CC=newcc
```

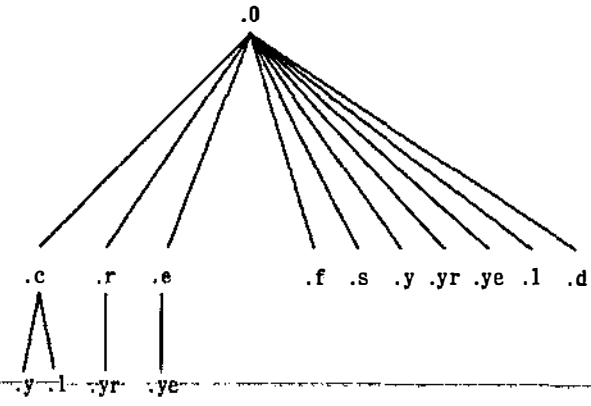


Figure 2-1. Summary of Default Transformation Path

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

```
make " CFLAGS=-O"
```

causes the optimizing C language compiler to be used.

SUGGESTIONS AND WARNINGS

The most common difficulties arise from **make**'s specific meaning of dependency. If file *x.c* has a "#include " defs" " line, then the object file *x.o* depends on **defs**; the source file *x.c* does not. If **defs** 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 which **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 new definition 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 since this mode of operation subverts the intention of **make** and destroys all memory of the previous relationships.

The debugging flag (**–d**) causes **make** to print out a very detailed description of what it is doing including the file times. The output is verbose and recommended only as a last resort.

Chapter 6

AUGMENTED VERSION OF make

GENERAL

This section describes an augmented version of the **make** command of the UNIX operating system. The augmented version is upward compatible with the old version. This section describes and gives examples of only the additional features. Further possible developments for **make** are also discussed. Some justification will be given for the chosen implementation, and examples will demonstrate the additional features.

The **make** command is an excellent program administrative tool used extensively in at least one project for over 2 years. However, **make** had the following shortcomings:

- Handling of libraries was tedious.
- Handling of the Source Code Control System (SCCS) file name format was difficult or impossible.
- Environment variables were completely ignored by **make**.
- The general lack of ability to maintain files in a remote directory.

These shortcomings hindered large scale use of **make** as a program support tool.

The AUGMENTED VERSION OF make is modified to handle the above problems. The additional features are within the original syntactic framework of **make** and few if any new syntactical entities are introduced. A notable exception is the *include* file capability. Further, most of the additions result in a "Don't know how to make ..." message from the old version of **make**.

The following paragraphs describe with examples the additional features of the **make** program. In general, the examples are taken from existing *makefiles*. Also, the illustrations are examples of working *makefiles*.

THE 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 new macro, **MAKEFLAGS**, is maintained by **make**. The new macro is defined as the collection of all input flag arguments into a string (without minus signs). The new 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 and set the input flags from the command line. The command line adds to the previous settings from the **MAKEFLAGS** environment variable.

3. Read macro definitions from the command line. These are made *not resettable*. Thus, any further assignments to these names are ignored.

4. Read the internal list of macro definitions. These are found in the file *rules.c* of the source for **make**. Figure 3-1 contains the complete makefile that represents the internally defined

macros and rules of the current version of **make**. Thus, if **make −r ...** is typed and a *makefile* includes the *makefile* in Figure 3-1, the results would be identical to excluding the **−r** option and the *include* line in the *makefile*. The Figure 3-1 output can be reproduced by the following:

```
make -fp - < /dev/null 2>/dev/null
```

The output appears on the standard output.
They give default definitions for the C language compiler (CC=cc), the assembler (AS=as), etc.

5. Read the environment. The environment variables are treated as macro definitions and marked as *exported* (in the shell sense). However, since **MAKEFLAGS*** is not an internally defined variable (in *rules.c*), this has the effect of doing the same assignment twice. The exception to this is when **MAKEFLAGS** is assigned on the command line. (The reason it was read previously was to turn the debug flag on before anything else was done.)

6. 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 an additional command 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*.

* **MAKEFLAGS** are read and set again.

† There is no way to override the command line assignments.

```
#               LIST OF SUFFIXES

.SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~
        .sh .sh~ .h .h~

#               PRESET VARIABLES

MAKE=make
YACC=yacc
YFLAGS=
LEX=lex
LFLAGS=
LD=ld
LDFLAGS=
CC=cc
CFLAGS=-o
AS=as
ASFLAGS=
GET=get
GFLAGS=
```

Figure 3-1. Example of Internal Definitions (Sheet 1 of 4)

#	SINGLE SUFFIX RULES
.c:	
	$(CC) -n -o $< -o $@
.c~:	
	$(GET) $(GFLAGS) -p $< > $*.c $(CC) -n -o $* .c -o $* -rm -f $*.c
.sh:	
	cp $< @
.sh~:	
	$(GET) &(GFLAGS) -p $< > .sh cp $* .sh $* -rm -f $* .sh
#	DOUBLE SUFFIX RULES
.c.o:	
	$(CC) $(CFLAGs) -c $<
.c~.o:	

Figure 3-1. Example of Internal Definitions (Sheet 2 of 4)

	$(GET) $(CFLAGS) -p $< > $*.c $(CC) $(CFLAGS) -c $*.c -rm -f $*.c
.c~ .c:	
	$(GET) $(GFLAGS) -p $< >$*.c
.s.o:	
	$(AS) $(ASFLAGS) -o $@ $<
.s~ .o:	
	$(GET) $(GFLAGS) -p $< > $*.s $(AS) $(ASFLAGS) -o $* .o $* .s -rm -f $*.s
.y.o:	
	$(YACC) $(YFLAGS) $< $(CC) $(CFLAGS) -c y.tab.c rm y.tab.o$@
.y~.o:	
	$(GET) $(GFLAG) -p $< > $*.y $(YACC) $(YFLAGS) $*.y $(CC) $(CFLAG) -c y.tab.c rm -f y.tab $*.y mv y.tab.o $*.o
.l.o:	
	$(LEX) $(LFLAGS) $< $(CC) $(CFLAGS) -c lex.yy.c rm lex.yy.c mv lex.yy.o $@

Figure 3-1. Example of Internal Definitions (Sheet 3 of 4)

```
.l~.o:
        $(GET) $(GFLAGS) -p $< > $*.l
        $(LEX) $(GFLAG) $*.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) -p $< > $*.y
        $(YACC) $(YFLAGS) $*.y
        mv -f $*.c
        -rm -f $*.y
.l.c:
        $(LEX) $<
        mv lex.yy.c$@
.c.a:
        $(CC) -c $(FLAGS) $<
        ar rv $@ $*.o
        rm -f $*.o
.c~.a:
        $(GET) $(GFLAGS) -p $< > $*.c
        $(CC) -c $(CFLAGS) $*.c
        ar rv $@ $*.o
.s~.a:
        $(GET) $(GFLAGS) -p $< > $*.s
        $(AS) $(ASFLAGS) -o $*.o $*.s
        ar rv $@ $*.o
        -rm -f $*.[so]
.h~.h
        $(GET) $(GFLAGS) -p $< > $*.h
```

Figure 3-1. Example of Internal Definitions (Sheet 4 of 4)

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 (from *rules.c*)
2. environment
3. *makefile(s)*
4. command line.

The **−e** flag has the effect of changing the order to:

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

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

RECURSIVE MAKEFILES

Another feature was added to **make** concerning 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 actually gets 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 get printed out including output from lower level invocations of **make**.

FORMAT OF SHELL COMMANDS WITHIN make

The **make** program remembers embedded newlines and tabs in shell command sequences. Thus, if the programmer puts a *for* loop in the makefile with indentation, when **make** prints it out, it retains the indentation and backslashes. The output can still be piped to the shell and is readable. This is obviously a cosmetic change; no new function is gained.

ARCHIVE LIBRARIES

The **make** program has an improved interface to archive libraries. Due to a lack of documentation, most people are probably not aware of the current syntax of addressing members of archive libraries. The previous version of **make** allows a user to name a member of a library in the following manner:

```
lib(object.o)
    or
lib((_localtime))
```

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 file name.)

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

```
lib:: lib(ctime.o)
      $(CC) -c -O ctime.c
      ar rv lib ctime.o
      rm ctime.o
lib:: lib(fopen.o)
      $(CC) -c -O fopen.c
      ar rv lib fopen.o
      rm fopen.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 file name being the only difference each time. (This is true in most cases.)

The current version 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 current archive rules defined internally are ".c.a", ".c{.a", and ".s{.a". [The tilde (~) syntax will be described shortly.] The user may define in makefile other rules needed.

The above 2-member library is then maintained with the following shorter makefile:

```
lib:    lib(ctime.o) lib(fopen.o)
        echo lib up-to-date.
```

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

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

Thus, the $@ macro is the ".a" target (lib); the $< and $* macros are set to the out-of-date C language file; and the file name scans the suffix, respectively (*ctime.c* and *ctime*). 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

```
lib:    lib(ctime.o)
        @echo lib up-to-date
```

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

1. Do *lib*.

2. To do *lib*, do each dependent of *lib*.

3. Do *lib*(*ctime.o*).

4. To do *lib*(*ctime.o*), do each dependent of *lib*(*ctime.o*). (There are none.)

5. Use internal rules to try to build *lib*(*ctime.o*). (There is no explicit rule.) Note that *lib*(*ctime.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 *lib* library name. The parenthesis forces the ".a" suffix. In this sense, the ".a" is hard wired into **make**.

6. Break the name *lib*(*ctime.o*) up into *lib* and *ctime.o*. Define two macros, $@ (=*lib*) and $* (=*ctime*).

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 *ctime.c*. Set $< to be *ctime.c* and execute the rule. In fact, **make** must then do *ctime.c*. However, the search of the current directory yields no other candidates, and the search ends.

8. The library has been updated. Do the rule associated with the "lib:" dependency; namely:

   echo lib up-to-date

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

```
lib(ctime.o):    $(INCDIR)/stdio.h
```

Thus, explicit references to *.o* files are unnecessary. There is also a new 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.

An example *makefile* for a larger library is given in Figure 3-2.

```
#               @(#)/usr/src/cmd/make/make.tm 3.2
LIB ==lsxlib
PR==lp
INSDIR == /ri/flop0/
INS == eval
lsx:            $(LIB) low.o mch.o
                ld -x low.o mch.o $(LIB)
                mv a.out lsx
                @size lsx
#               Here, $(INS) as either "." or "eval".
lsx:
                $(INS)'cp lsx $(INSDIR)lsx . .
                    strip $(INSDIR)lsx . .
                    ls -l $(INSDIR)lsx'
print:
                $(PR) header.slow.smch.s*.h*.c Makefile
```

Figure 3-2. Example of Library Makefile (Sheet 1 of 3)

$(LIB):	
	$(LIB)(CLOCK.o)
	$(LIB)(main.o)
	$(LIB)(tty.o)
	$(LIB)(trap.o)
	$(LIB)(sysent.o)
	$(LIB)(sys2.o)
	$(LIB)(syn3.o)
	$(LIB)(syn4.o)
	$(LIB)(sys1.o)
	$(LIB)(sig.o)
	$(LIB)(fio.o)
	$(LIB)(kl.o)
	$(LIB)(alloc.o)
	$(LIB)(nami.o)
	$(LIB)(iget.o)
	$(LIB)(rdwri.o)
	$(LIB)(subr.o)

Figure 3-2. Example of Library Makefile (Sheet 2 of 3)

	$(LIB)(bio.o)
	$(LIB)(decfd.o)
	$(LIB)(sip.o)
	$(LIB)(space.o)
	$(LIB)(puts.o)
	@echo $(LIB) now up to date.
.s.o:	
	as -o $*.o header.s $*.s
.o.a:	
	ar rv $@ $<
	rm -f $<
.s.a:	
	as -o $*.o header.s $*.s ar rv $@ $*.o rm -f $*.o
.PRECIOUS:	$(LIB)

Figure 3-2. Example of Library Makefile (Sheet 3 of 3)

The reader will note also that there are no lingering "*.o" files left around. The result is a library maintained directly from the source files (or more generally from the SCCS files).

SOURCE CODE CONTROL SYSTEM FILE NAMES: THE TILDE

The syntax of **make** does not directly permit referencing of prefixes. For most types of files on UNIX operating system machines, 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 file name part of the complete pathname.

To allow **make** easy access to the prefix "s." requires either a redefinition of the rule naming syntax of **make** or a trick. The trick is to use the tilde (~) as an identifier of SCCS files. Hence, ".c~.o" refers to the rule which transforms an SCCS C language source file into an object. Specifically, the internal rule is

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

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

The following SCCS suffixes are internally defined:

```
.c~
.y~
.s~
.sh~
.h~
```

The following rules involving SCCS transformations are internally defined:

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

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

THE NULL SUFFIX

In the UNIX system source code, there are many commands which consist of a single source file. It was wasteful to maintain an object of such files for **make**. The current implementation supports single suffix rules (a null suffix). Thus, to maintain the program *cat*, a rule in the *makefile* of the following form is needed:

```
.c:
        $(CC) -n  -O  $< -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 notable 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~:
.sh:
.sh~:
```

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

INCLUDE FILES

The **make** program has an include file capability. 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 string is assumed to be a file name which the current invocation of **make** will read. The file descriptors are stacked for reading *include* files so that no more than about 16 levels of nested *includes* are supported.

INVISIBLE SCCS MAKEFILES

The SCCS *makefiles* are invisible to **make**. That is, if **make** is typed and only a file named *s.makefile* exists, **make** will do a **get** on the file, then read and remove the file. Using the **−f, make** will get, read, and remove arguments and *include* files.

DYNAMIC DEPENDENCY PARAMETERS

A new dependency parameter has been defined. 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 UNIX software command directory could have a *makefile* like:

```
CMDS = cat dd echo date cc cmp comm ar ld 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 **$$(@F)**. It represents the file name part of $$@. 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 \
        $(INCIDR)/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.

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 letter macro. The "F" refers to the file name part of the single letter 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 shell command can be

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

An interesting example of the use of these features can be found in the set of *makefiles* in Figure 3-3. Each *makefile* is named "70.mk". The following command forces a complete rebuild of the operating system:

```
FRC=FRC make -f 70.mk
```

where the current directory is *ucb*. The FRC is a convention for *FoRC*ing **make** to completely rebuild a target starting from scratch.

OUTPUT TRANSLATIONS

Macros in shell commands can now be 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 **$(macro)** is considered as a bunch of strings each delimited by white space (blanks or tabs). Thus, the occurrence of *string1* in **$(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. A more general regular expression match could be implemented if the need arises. 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 rv $(LIB) $?
        rm $?
```

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

Chapter 7

SOURCE CODE CONTROL SYSTEM USER GUIDE

GENERAL

The Source Code Control System (SCCS) is a collection of the UNIX software commands that help individuals or projects control and account for changes to files of text. The source code and documentation of software systems are typical examples of files of text to be changed. The SCCS is a collection of programs that run under the UNIX operating system. It is convenient to conceive of SCCS as a custodian of files. The SCCS provides facilities for

- Storing files of text
- Retrieving particular versions of the files
- Controlling updating privileges to files
- Identifying the version of a retrieved file
- Recording when, where, and why the change was made and who made each change to a file.

These types of facilities are important when programs and documentation undergo frequent changes because of maintenance and/or enhancement work. It is often desirable to regenerate the version of a program or document as it existed before changes were applied to it. This can be done by keeping copies (on paper or other media), but this method quickly becomes unmanageable and wasteful as the number of programs and documents increases. The SCCS provides an attractive solution because the original file is stored on disk. Whenever changes are made to the file, the SCCS stores only the changes. Each set of changes is called a "delta".

This chapter, together with relevant portions of section 1 of the Runtime System manual is a complete user's guide to SCCS. The following topics are covered:

- SCCS for Beginners: How to make an SCCS file, how to update it, and how to retrieve a version thereof.
- How Deltas Are Numbered: How versions of SCCS files are numbered and named.
- SCCS Command Conventions: Conventions and rules generally applicable to all SCCS commands.
- SCCS Commands: Explanation of all SCCS commands with discussions of the more useful arguments.
- SCCS Files: Protection, format, and auditing of SCCS files including a discussion of the differences between using SCCS as an individual and using it as a member of a group or project. The role of a "project SCCS administrator" is introduced.

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

Throughout this section, each reference of the form **name**(1), **name**(1M), or **name**(7) refers to entries in the Runtime System manual. All other references to entries of the form **name**(N), where "N" is a number (2 through 5) possibly followed by a letter, refer to entry **name** in section N of the Software Development System manual.

SCCS FOR BEGINNERS

It is assumed that the reader knows how to log onto a UNIX system, create files, and use the text editor. A number of terminal-session fragments are presented. All of them should be tried since the best way to learn SCCS is to use it.

To supplement the material in this section, the detailed SCCS command descriptions in section 1 of the Runtime System manual should be consulted.

A. Terminology

Each SCCS file is composed of one or more sets of changes applied to the null (empty) version of the file, with each set of changes usually depending on all previous sets. Each set of changes is called a "delta" and is assigned a name, called the *S*CCS *ID*entification string (SID). The SID is composed of at most four components. The first two components are the "release" and "level" numbers which are separated by a period. Hence, the first delta (for the original file) is called "1.1", the second "1.2", the third "1.3", etc. The release number can also be changed allowing, for example, deltas "2.1", "3.1", etc. The change in the release number usually indicates a major change to the file.

Each delta of an SCCS file defines a particular version of the file. For example, delta 1.5 defines version 1.5 of the SCCS file, obtained by applying to the null (empty) version of the file the changes that constitute deltas 1.1, 1.2, etc., up to and including delta 1.5 itself, in that order.

C-7

B. Creating an SCCS File via "admin"

Consider, for example, a file called *lang* that contains a list of programming languages.

```
c
pl/i
fortran
cobol
algol
```

Custody of the *lang* file can be given to SCCS. The following **admin** command (used to "administer" SCCS files) creates an SCCS file and initializes delta 1.1 from the file *lang*:

```
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*, indicates that **admin** is to create a new SCCS file and "initialize" the new SCCS file with

the contents of the file *lang*. This initial version is a set of changes (delta 1.1) applied to the null SCCS file.

The **admin** command replies

No id keywords (cm7)

This is a warning message (which may also be issued by other SCCS commands) that is to be ignored for the purposes of this section. Its significance is described under the **get** command in the part "SCCS COMMANDS." In the following examples, this warning message is not shown although it may actually be issued by the various commands. The file *lang* should now be removed (because it can be easily reconstructed using the **get** command) as follows:

rm lang

C. Retrieving a File via "get"

The *lang* file can be reconstructed by using the following **get** command:

get s.lang

The command causes the creation (retrieval) of the latest version of file *s.lang* and prints the following messages:

1.1
5 lines

This means that **get** retrieved version 1.1 of the file, which is made up of five lines of text. The retrieved text is placed in a file whose name is formed by deleting the "s." prefix from the name of the SCCS file. Hence, the file *lang* is created.

The "get s.lang" command simply creates the file *lang* (read-only) and keeps no information regarding its creation. On the other hand,

in order to be able to subsequently apply changes to an SCCS file with the **delta** command, the **get** command must be informed of your intention to do so. This is done as follows:

```
get -e s.lang
```

The **-e** keyletter causes **get** to create a file *lang* for both reading and writing (so it may be edited) and places certain information about the SCCS file in another new file. The new file, called the *p-file,* will be read by the **delta** command. The **get** command prints the same messages as before except that the SID of the version to be created through the use of **delta** is also issued. For example,

```
get -e s.lang
1.1
new delta 1.2
5 lines
```


The file *lang* may now be changed, for example, by

```
ed lang
27
$a
snobol
ratfor
.
w
41
q
```

D. Recording Changes via "delta"

In order to record within the SCCS file the changes that have been applied to *lang*, execute the following command:

```
delta s.lang
```

Delta prompts with

```
comments?
```

The response should be a description of why the changes were made. For example,

```
comments? added more languages
```

The **delta** command then reads the *p-file* and determines what changes were made to the file *lang*. The **delta** command does this by doing its own **get** to retrieve the original version and by applying the **diff**(1) command to the original version and the edited version.

When this process is complete, at which point the changes to *lang* have been stored in *s.lang*, **delta** outputs

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

The number "1.2" is the name of the delta just created, and the next three lines of output refer to the number of lines in the file *s.lang*.

E. Additional Information About "get"

As shown in the previous example, the command

```
get s.lang
```

retrieves the latest version (now 1.2) of the file *s.lang*. This is done by starting with the original version of the file and successively applying deltas (the changes) in order until all have been applied.

In the example chosen, the following commands are all equivalent:

```
get s.lang
get -r1 s.lang
get -r1.2 s.lang
```

The numbers following the **-r** keyletter are SIDs. Note that omitting the level number of the SID (as in "get -r1 s.lang") is equivalent to specifying 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 truly major change is made to a file, the significance of that change is usually indicated by changing the release number (first component of the SID) of the delta being made. Since normal automatic numbering of deltas proceeds by incrementing the level number (second component of the SID), the user must indicate to SCCS the need to change the release number. This 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. The **get** command also interprets this as a request to change the release number of the delta which the user desires to create to 2, thereby causing it to be named 2.1, rather than 1.3. This information is conveyed to **delta** via the *p-file*. The **get** command then outputs

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

which indicates that version 1.2 has been retrieved and that 2.1 is the version **delta** will create. If the file is now edited, for example, by

```
ed lang
41
/cobol/d
w
35
q
```

and **delta** executed

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

the user will see by **delta**'s output that version 2.1 is indeed created.

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

Deltas may now be created in release 2 (deltas 2.2, 2.3, etc.), or another new release may be created in a similar manner. This process may be continued as desired.

F. The "help" Command

If the command

```
get abc
```

is executed, the following message will be output:

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

The string "co1" is a code for the diagnostic message and may be used to obtain a fuller explanation of that message by use of the **help** command.

```
help col
```

This produces the following output:

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

Thus, **help** is a useful command to use whenever there is any doubt about the meaning of an SCCS message. Detailed explanations of almost all SCCS messages may be found in this manner.

DELTA NUMBERING

It is convenient to conceive of the deltas applied to an SCCS file as the nodes of a tree in which the root is the initial version of the file. The root delta (node) is normally named "1.1" and successor deltas (nodes) are named "1.2", "1.3", etc. The components of the names of the deltas are called the "release" and the "level" numbers, respectively. Thus, normal naming of successor deltas proceeds by incrementing the level number, which is performed automatically by SCCS whenever a delta is made. In addition, the user may wish to change the release number when making a delta to indicate that a major change is being made. When this is done, the release number also applies to all successor deltas unless specifically changed again. Thus, the evolution of a particular file may be represented as in Figure 4-1.

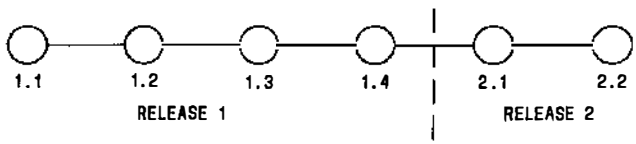


Figure 4-1. Evolution of an SCCS File

Such a structure may be termed the "trunk" of the SCCS tree. Figure 4-1 represents the normal sequential development of an SCCS file in which changes that are part of any given delta are dependent upon *all* the preceding deltas.

However, there are situations in which it is necessary to cause a branching in the tree in that changes applied as part of a given delta are *not* dependent upon all previous deltas. As an example, consider a program which is in production use at version 1.3 and for which development work on release 2 is already in progress. Thus, release 2 may already have some deltas precisely as shown in Figure 4-1. Assume that a production user reports a problem in version 1.3 and that the nature of the problem is such that it 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.).

The new delta is a node on a branch of the tree. Its name consists of four components; the release number and the level number (as with trunk deltas) plus the "branch" number and the "sequence" number. The delta name appears as follows:

release.level.branch.sequence

The branch number is assigned to each branch that is a descendant of a particular trunk delta with the first such branch being 1, the next one 2, etc. The sequence number is assigned, in order, to each delta on a particular branch. Thus, 1.3.1.2 identifies the second delta of the first branch that derives from delta 1.3. This is shown in Figure 4-2.

The concept of branching may be extended to any delta in the tree. The naming of the resulting deltas proceeds in the manner just illustrated.

Two observations are of importance with regard to naming deltas. First, the names of trunk deltas contain exactly two components, and the names of branch deltas contain exactly four components. Second,

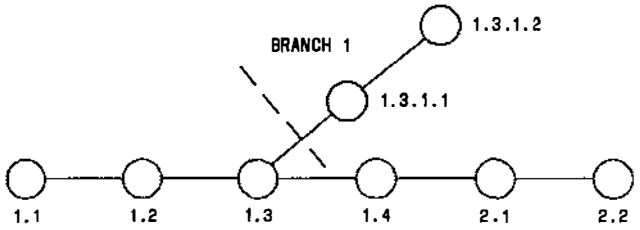


Figure 4-2. Tree Structure With Branch Deltas

the first two components of the name of branch deltas are always those of the ancestral trunk delta, and the branch component is assigned in the order of creation of the branch independently of its location relative to the trunk delta. Thus, a branch delta may always be identified as such from its name. Although the ancestral trunk delta may be identified from the branch delta's name, it is not possible to determine the entire path leading from the trunk delta to the branch delta. For example, if delta 1.3 has one branch emanating from it, all deltas on that branch will be named 1.3.1.n. If a delta on this branch then has another branch emanating from it, all deltas on the new branch will be named 1.3.2.n (see Figure 4-3). The only information that may be derived from the name of delta 1.3.2.2 is that it is the chronologically second delta on the chronologically second branch whose trunk ancestor is delta 1.3. In particular, it is *not* possible to determine from the name of delta 1.3.2.2 all the deltas between it and trunk ancestor 1.3.

C-7

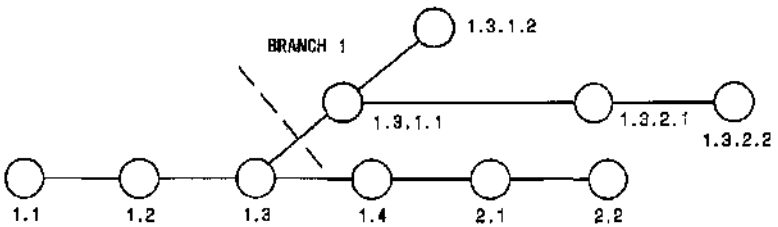


Figure 4-3. Extending the Branching Concept

It is obvious that the concept of branch deltas allows the generation of arbitrarily complex tree structures. Although this capability has been provided for certain specialized uses, it is strongly recommended that the SCCS tree be kept as simple as possible because comprehension of its structure becomes extremely difficult as the tree becomes more complex.

SCCS COMMAND CONVENTIONS

This part discusses the conventions and rules that apply to SCCS commands. These rules and conventions are generally applicable to all SCCS commands with exceptions indicated. The SCCS commands accept two types of arguments:

- Keyletter arguments
- File arguments.

Keyletter arguments (hereafter called simply "keyletters") begin with a minus sign (-), followed by a lowercase alphabetic character, and in some cases, followed by a value. These keyletters control the execution of the command to which they are supplied.

File arguments (names of files and/or directories) specify the file(s) that the given SCCS 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, file arguments may not begin with a minus sign. However, if the name "-" (a lone minus sign) is specified as an argument to a command, the command reads the standard input for lines and takes 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 **find**(1) or **ls**(1) commands. Again, names of non-SCCS files and of unreadable files are silently ignored.

All keyletters specified for a given command apply to all file arguments of that command. All keyletters are processed before any file arguments with the result that the placement of keyletters is arbitrary (i.e., keyletters may be interspersed with file arguments). File arguments, however, are processed left to right. Somewhat different argument conventions apply to the **help, what, sccsdiff,** and **val** commands.

Certain actions of various SCCS commands are controlled by flags appearing in SCCS files. Some of these flags are discussed in this part. For a complete description of all such flags, see **admin**(1) in the Runtime System manual.

The distinction between the real user [see **passwd**(1)] and the effective user of a UNIX system is of concern in discussing various actions of SCCS commands. For the present, it is assumed that both the real user and the effective user are one and the same (i.e., the user who is logged into a UNIX system). This subject is discussed further in "SCCS FILES."

All SCCS commands that modify an SCCS file do so by writing a temporary copy, called the *x-file.* This file ensures that the SCCS file is not damaged if processing should terminate abnormally. The name of the *x-file* is formed by replacing the "s." of the SCCS file name with "x.". When processing is complete, the old SCCS file is removed and the *x-file* is renamed to be the SCCS file. The *x-file* is created in the directory containing the SCCS file, given the same mode [see **chmod**(1)] as the SCCS file, and owned by the effective user.

To prevent simultaneous updates to an SCCS file, commands that modify SCCS files create a *lock-file,* called the *z-file,* whose name is formed by replacing the "s." of the SCCS file name with "z.". The *z-file* contains the process number of the command that creates it, and its existence is an indication to other commands that the SCCS file is being updated. Thus, other commands that modify SCCS files do not process an SCCS file if the corresponding *z-file* exists. The *z-file* is created with mode 444 (read-only) in the directory containing the SCCS file and is owned by the effective user. This file exists only for the duration of the execution of the command that creates it. In

general, users can ignore *x-files* and *z-files*. The files may be useful in the event of system crashes or similar situations.

The SCCS commands produce diagnostics (on the diagnostic output) of the form:

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

The code in parentheses may be used as an argument to the **help** command to obtain a further explanation of the diagnostic message. Detection of a fatal error during the processing of a file causes the SCCS command to terminate processing of that file and to proceed with the next file, in order, if more than one file has been named.

SCCS COMMANDS

This part describes the major features of all the SCCS commands. Detailed descriptions of the commands and of all their arguments are given in section 1 of the Runtime System manual and should be consulted for further information. The discussion below covers only the more common arguments of the various SCCS commands.

The commands follow in approximate order of importance. The following is a summary of all the SCCS commands and of their major functions:

get	Retrieves versions of SCCS files.
delta	Applies changes (deltas) to the text of SCCS files, i.e., creates new versions.
admin	Creates SCCS files and applies changes to parameters of SCCS files.
prs	Prints portions of an SCCS file in user specified format.

help	Gives explanations of diagnostic messages.
rmdel	Removes a delta from an SCCS file; allows the removal of deltas that were created by mistake.
cdc	Changes the commentary associated with a delta.
what	Searches any UNIX system file(s) for all occurrences of a special pattern and prints out what follows it; is useful in finding identifying information inserted by the **get** command.
sccsdiff	Shows the differences between any two versions of an SCCS file.
comb	Combines two or more consecutive deltas of an SCCS file into a single delta; often reduces the size of the SCCS file.
val	Validates an SCCS file.

C-7

A. The "get" Command

The **get** command creates a text file that contains a particular version of an SCCS file. The particular version is retrieved by beginning with the initial version and then applying deltas, in order, until the desired version is obtained. The created file is called the *g-file.* The *g-file* name is formed by removing the "s." from the SCCS file name. The *g-file* 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 invoked.

The most common invocation of **get** is

```
get s.abc
```

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

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

which indicates that

1. Version 1.3 of file "s.abc" was retrieved (1.3 is the latest trunk delta).

2. This version has 67 lines of text.

3. No ID keywords were substituted in the file.

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

In the case of several file arguments (or directory-name arguments), similar information is given for each file processed, but the SCCS file name precedes it. For example,

```
get s.abc s.def
```

produces

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

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

ID Keywords

In generating a *g-file* to be used for compilation, it is useful and informative 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. The 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 these ID keywords. The format of an ID keyword is an uppercase letter enclosed by percent signs (%). For example,

5.1

is defined as the ID keyword that is replaced by the SID of the retrieved version of a file. Similarly, **5/30/83** is defined as the ID keyword for the current date (in the form "mm/dd/yy"), and **sccs1** is defined as the name of the *g-file*. Thus, executing **get** on an SCCS file that contains the PL/I declaration,

```
DCL ID CHAR(100) VAR INIT('sccs1 5.1 5/30/83');
```

gives (for example) the following:

```
DCL ID CHAR(100) VAR INIT('MODNAME 2.3 07/07/77');
```

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 20 ID keywords provided, see **get**(1) in the *UNIX System User Reference Manual*.

Retrieval of Different Versions

Various keyletters are provided to allow the retrieval of other than the default version of an SCCS file. Normally, the default version is the most recent delta of the highest-numbered release on the trunk of the SCCS file tree. However, if the SCCS file being processed has a **d** (default SID) flag, the SID specified as the value of this flag is used as a default. The default SID is interpreted in exactly the same way as the value supplied with the **−r** keyletter of **get**.

The **−r** keyletter is used to specify a SID to be retrieved, in which case the **d** (default SID) flag (if any) is ignored. For example,

```
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,

```
get -r1.5.2.3 s.abc
```

which produces (for example) on the standard output

```
1.5.2.3
234 lines
```

When a 2- or 4-component SID is specified as a value for the **−r** keyletter (as above) and the particular version does not exist in the SCCS file, an error message results. Omission of the level number, as in

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

causes retrieval of the trunk delta with the highest level number within the given release if the given release exists. 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, assuming release 9 does not exist in file *s.abc* and that release 7 is actually the highest-numbered release below 9, execution of

```
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, omission of 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 it exists. (If the given branch does not exist, an error message results.) This might result in the following output:

```
4.3.2.8
89 lines
```

The **-t** keyletter is used to retrieve the latest (top) version in a particular release (i.e., when no **-r** keyletter is supplied or when its value is simply a release number). The latest version is defined as that delta which was produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in

release 3 is 3.5,

```
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

Specification of the **–e** keyletter to the **get** command is an indication of the intent to make a delta, and as such, its use is restricted. The presence of this keyletter causes **get** to check

1. The user list (a list of login names and/or group IDs of users allowed to make deltas) to determine if the login name or group ID of the user executing **get** is on that list. Note that a null (empty) user list behaves as if it contained all possible login names.

2. The release (R) of the version being retrieved satisfies the relation:

   floor is < or = to R which is
   < or = to ceiling

   to determine if the release being accessed is a protected release. The "floor" and "ceiling" are specified as flags in the SCCS file.

3. The R is not locked against editing. The "lock" is specified as a flag in the SCCS file.

4. Whether or not multiple concurrent edits are allowed for the SCCS file as specified 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, the **−e** keyletter 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* that already exists and is being edited for the purpose of making a delta.

Any ID keywords appearing in the *g-file* are not substituted by **get** (when the **−e** keyletter is specified) because the generated *g-file* is subsequently used to create another delta. Replacement of ID keywords cause them to be permanently changed within the SCCS file. In view of this, **get** does not need to check for the presence of ID keywords within the *g-file,* so the message

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

is never output when **get** is invoked with the **−e** keyletter.

In addition, the **−e** keyletter causes the creation (or updating) of a *p-file* which is used to pass information to the **delta** command.

The following is an example of the use of the **−e** keyletter:

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

which produces (for example) on the standard output

1.3
new delta 1.4
67 lines

If the **−r** and/or **−t** keyletters are used together with the **−e** keyletter, the version retrieved for editing is as specified by the **−r** and/or **−t** keyletters.

The keyletters **-i** and **-x** may be used to specify a list [see **get**(1) in the Runtime System manual for the syntax of such a list] of deltas to be included and excluded, respectively, by **get**.

Including a delta means forcing the changes that constitute the particular delta to be included in the retrieved version. This is useful if one wants to apply 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 (in the version of the SCCS file to be created) the effects of a previous delta. Whenever deltas are included or excluded, **get** checks for possible interference between such deltas and those deltas that are normally used in retrieving the particular version of the SCCS file. Two deltas can interfere, for example, when each one changes the same line of the retrieved *g-file*. Any interference is indicated by a warning that shows the range of lines within the retrieved *g-file* in which the problem may exist. The user is expected to examine the *g-file* to determine whether a problem actually exists and to take whatever corrective measures (if any) are deemed necessary (e.g., edit the file).

Warning: The −i and −x keyletters should be used with extreme care.

The **−k** keyletter is provided to facilitate regeneration of a *g-file* that may have been accidentally removed or ruined subsequent to the execution of **get** with the **−e** keyletter or to simply generate a *g-file* in which the replacement of ID keywords has been suppressed. Thus, a *g-file* generated by the **−k** keyletter is identical to one produced by **get** and executed with the **−e** keyletter. However, 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 a number of deltas to be "in progress" at any given time. This means that a number of **get** commands with the **−e** keyletter may be executed on the same file provided that no two executions retrieve the same version (unless multiple concurrent edits are allowed).

The *p-file* (created by the **get** command invoked with the **−e** keyletter) is named by replacing the "s." in the SCCS file name with "p.". It is created in the directory containing 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 that is 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 the *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 both checks succeed, the user is informed that other deltas are in progress and processing continues. If either check fails, an error message results. It is important to note that the various executions of **get** should be carried out from different directories. Otherwise, only the first execution succeeds since subsequent executions would attempt to overwrite a writable *g-file,* which is an SCCS error condition. In practice, such multiple executions are performed by different users so that this problem does not arise since each user normally has a different working directory. See "Protection" under the part "SCCS FILES" for a discussion of how different users are permitted to use SCCS commands on the same files.

Figure 4-4 shows, for the most useful cases, the version of an SCCS file retrieved by **get**, as well as the SID of the version to be eventually created by **delta**, as a function of the SID specified to **get**.

SID SPECIFIED*	–b KEY-LETTER USED†	OTHER CONDITIONS	SID RETRIEVED	SID OF DATA TO BE CREATED
none†	no	R default to mR	mRmL	mR(mL+1)
none‡	yes	R default to mR	mRmL	mRmL.(mB+1)
R	no	R > mR	mRmL	R.1§
R	no	R == mR	mRmL	mR.(mL+1)
R	yes	R > mR	mRmL	mR.mL.(mB+1).1
R	yes	R == mR	mR.mL	mR.mL.(mB+1).1
R	–	R<mR		
R	–	R< mR and does not exist	hR.mL**	hR.mL.(mB+1).1
R	–	Trunk successor in release > R and R exists	R.mL	R.mL.(mB+1).1

See footnotes on sheet 3 of 3.

Figure 4-4. Determination of New SID (Sheet 1 of 3)

SID SPECIFIED*	-b KEY-LETTER USED†	OTHER CONDITIONS	SID RETRIEVED	SID OF DATA TO BE CREATED
R.L.	no	No trunk successor	R.L	R.(L+1)
R.L.	yes	No trunks successor	R.L	R.L.(mB+1).1
R.L	-	Trunk 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	no	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 4-4. Determination of New SID (Sheet 2 of 3)

Footnotes:

* "R", "L", "B", and "S" are "release", "level", "branch", and "sequence" components of the SID, respectively; "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 (i.e., maximum branch number plus 1) of level L within release R". Also note that if the SID specified is of the form "R.L", "R.L.B", or "R.L.B.S", each of the specified components must exist.

† The **−b** keyletter is effective only if the **b** flag [see **admin(1)**] is present in the file. In this state, an entry of "-" means "irrelevant".

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

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

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

Figure 4-4. Determination of New SID (Sheet 3 of 3)

Concurrent Edits of Same SID

Under normal conditions, **get**s for editing (**−e** keyletter is specified) based on the same SID are not permitted to occur concurrently. That is, **delta** must be executed before a subsequent **get** for editing is executed at the same SID as the previous **get**. However, multiple concurrent edits (defined to be two or more successive executions of **get** for editing based on the same retrieved SID) are allowed if the **j** flag is set in the SCCS file.

Thus:

```
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 execution of **delta**. In this case, a **delta** command corresponding to the first **get** produces delta 1.2 [assuming 1.1 is the latest (most recent) trunk delta], and the **delta** command corresponding to the second **get** produces delta 1.1.1.1.

Keyletters That Affect Output

Specification of the **-p** keyletter causes **get** to write the retrieved text 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. This may be used, for example, to create *g-files* with arbitrary names.

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

The **-p** keyletter is particularly useful when used with the "!" or "$" arguments of the **send**(1C) command. For example,

```
send MOD=s.abc REL=3 compile
```

given that file *compile* contains

```
//plicomp job job-card-information
//step1 exec plickc
//pli.sysin dd *
 -s
 !get -p -rREL MOD
/*
//
```

will **send** the highest level of release 3 of file *s.abc.* Note that the line "~-s" (that causes **send** to make ID keyword substitutions before detecting and interpreting control lines) is necessary if **send** is to substitute "s.abc" for MOD and "3" for REL in the line "~!get -p -rREL MOD".

The **−s** keyletter suppresses all output that is normally directed to the standard output. Thus, the SID of the retrieved version, the number of lines retrieved, etc., are not output. This does not, however, affect messages to the diagnostic output. This keyletter is used to prevent nondiagnostic messages from appearing on the user's terminal and is often used in conjunction with the **−p** keyletter to "pipe" the output of **get**, as in

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

The **−g** keyletter is supplied to suppress the actual retrieval of the text of a version of the SCCS file. This may be useful in a number of ways. For example, to verify the existence of a particular SID in an SCCS file, one may execute

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

This outputs the given SID if it exists in the SCCS file or it generates an error message if it does not. Another use of the **−g** keyletter is in regenerating a *p-file* that may have been accidentally destroyed.

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

The **−l** keyletter causes the creation of an *l-file,* which is named by replacing the "s." of the SCCS file name with "l.". This file is

created in the current directory with mode 444 (read-only) and is owned by the real user. It contains a table [whose format is described in get(1) in the Runtime System manual] showing the deltas used in constructing a particular version of the SCCS file. For example,

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

generates an *l-file* showing the deltas applied to retrieve version 2.3 of the SCCS file. Specifying a value of "p" with the –l keyletter, as in

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

causes the generated output to be written to the standard output rather than to the *l-file*. The **–g** keyletter may be used with the **–l** keyletter to suppress the actual retrieval of the text.

The **–m** keyletter is of use in identifying, line by line, the changes applied to an SCCS file. Specification of this keyletter causes each line of the generated *g-file* to be preceded by the SID of the delta that caused that line to be inserted. The SID is separated from the text of the line by a tab character.

The **–n** keyletter causes each line of the generated *g-file* to be preceded by the value of the **sccs1** ID keyword and a tab character. The **–n** keyletter 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 the **–m** and **–n** keyletters are specified, each line of the generated *g-file* is preceded by the value of the **sccs1** ID keyword and a tab (this is the effect of the **–n** keyletter) and followed by the line in the format produced by the **–m** keyletter. Because use of the **–m** keyletter and/or the **–n** keyletter causes the contents of the *g-file* to be modified, such a *g-file* must *not* be used for creating a delta.

Therefore, neither the **−m** keyletter nor the **−n** keyletter may be specified together with the **−e** keyletter.

See **get**(1) in the Runtime System manual for a full description of additional **get** keyletters.

B. The "delta" Command

The **delta** command is used to incorporate the changes made to a *g-file* into the corresponding SCCS file, i.e., to create a delta, and therefore, a new version of the file.

Invocation of the **delta** command requires the existence of a *p-file*. The **delta** command 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. The **delta** command performs the same permission checks that **get** performs when invoked by the **−e** keyletter. 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* (its name is formed by replacing the "s." of the SCCS file name with "d.") and is obtained by performing an internal **get** at 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** with the **−e** keyletter more than once on the same SCCS file. The **−r** keyletter must then be used with **delta** to specify the SID that uniquely identifies the *p-file* entry. This entry is the one used to obtain the SID of the delta to be created.

In practice, the most common invocation of **delta** is

```
delta s.abc
```

which prompts on the standard output (but only if it is a terminal)

```
comments?
```

to which the user replies with a description of why the delta is being made, terminating 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. (Again, this prompt is printed only if the standard output is a terminal.) The standard input is then read for MR numbers, separated by blanks and/or tabs, terminated in the same manner as the response to the prompt "comments?". In a tightly controlled environment, it is expected that deltas are created only as a result of some trouble report, change request, trouble ticket, etc., collectively called [MRs]. It is desirable (or necessary) to record such MR number(s) within each delta.

The **−y** and/or **−m** keyletters may be used to supply the commentary (comments and MR numbers, respectively) on the command line rather than through the standard input.

```
delta -y" descriptive comment" -m" mrnum1 mrnum2" s.abc
```

In this case, the corresponding prompts are not printed, and the standard input is not read. The **-m** keyletter is allowed only if the SCCS file has a **v** flag. These keyletters are useful when **delta** is executed from within a shell procedure [see **sh**(1) in the Runtime System manual].

The commentary (comments and/or MR numbers), whether solicited by **delta** or supplied via keyletters, is recorded as part of the entry for the delta being created and applies to all SCCS files processed by the same invocation of **delta**. This implies that (if **delta** is invoked 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 this flag, then none of the files named may have

it. Any file that does not conform to these rules is not processed.

When processing is complete, **delta** outputs (on the standard output) the SID of the created delta (obtained from the *p-file* entry) and the counts of lines inserted, deleted, and left unchanged by the delta. Thus, a typical output might be

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

It is possible that the counts of lines reported as inserted, deleted, or unchanged by **delta** do not agree with the user's perception of the changes applied to the *g-file*. The reason for this is that there usually are a number of ways to describe a set of such changes, especially if lines are moved around in the *g-file,* and **delta** is likely to find a description that differs from the user's perception. However, the total number of lines of the new delta (the number inserted plus the number left unchanged) should agree with the number of lines in the edited *g-file.*

If (in the process of making a delta) **delta** 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 indicates 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 creating a delta from a *g-file* that was created by a **get** without the **-e** keyletter (recall that ID keywords are replaced by **get** in that case). This could also be caused by accidentally deleting or changing the ID keywords during the editing of the *g-file.* Another possibility is that the file had no ID keywords. In any case, it is left up to the user to determine what remedial action is necessary. However, the delta is made unless there is an **i** flag in the SCCS file indicating that this should be treated as a fatal error. In this last case, the delta is not 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* which is described in the part "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 the **−n** keyletter is specified. Thus:

```
delta -n s.abc
```

will keep the *g-file* upon completion of processing.

The **−s** (silent) keyletter suppresses all output that is normally directed to the standard output, other than the prompts "comments?" and "MRs?". Thus, use of the **−s** keyletter together with the **−y** keyletter (and possibly, the **−m** keyletter) causes **delta** neither to read the standard input nor to write the standard output.

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

C. The "admin" Command

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

Two keyletters are supplied for use in conjunction with detecting and correcting "corrupted" SCCS files (see "Auditing" in part "SCCS FILES"). Newly created SCCS files are given 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 upon that file.

Creation of SCCS Files

An SCCS file may be created by executing the command

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

in which the value "first" of the **−i** keyletter specifies 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 the value of the **−i** keyletter indicates that **admin** is to read the standard input for the text of the initial delta. Thus, 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 same invocation of 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 the **−i** keyletter.

When an SCCS file is created, the release number assigned to its first delta is normally "1", and its level number is always "1". Thus, the first delta of an SCCS file is normally "1.1". The **−r** keyletter is used to specify the release number to be assigned to the first delta. Thus:

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

indicates that the first delta should be named "3.1" rather than "1.1". Because this keyletter is only meaningful in creating the first delta, its use is only permitted with the **−i** keyletter.

Inserting Commentary for the Initial Delta

When an SCCS file is created, the user may choose to supply commentary stating the reason for creation of the file. This is done by supplying comments (**−y** keyletter) and/or MR numbers (**−m** keyletter) in exactly the same manner as for **delta**. The creation of an SCCS file may sometimes be the direct result of an MR. If comments (**−y** keyletter) are 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 (**−m** keyletter), the **v** flag must also be set (using the **−f** keyletter described below). The **v** flag simply determines whether or not MR numbers must be supplied when using any SCCS command that modifies a "delta commentary" [see **sccsfile**(4) in the *UNIX System User Reference Manual]* in the SCCS file. Thus:

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

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

Initialization and Modification of SCCS File Parameters

The portion of the SCCS file reserved for descriptive text may be initialized or changed through the use of the **−t** keyletter. The descriptive text is intended as a summary of the contents and purpose of the SCCS file.

When an SCCS file is being created and the **−t** keyletter is supplied, 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, the −t keyletter specifies that the descriptive text (if any) currently in the file is to be replaced with the text in the named file. Thus:

```
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 file name 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, changed, or deleted through the use of the **-f** and -d keyletters, respectively. The flags of an SCCS file are used to direct certain actions of the various commands. See **admin**(1) in the Runtime System manual for a description of all the flags. For example, the **i** flag specifies that the warning message (stating that there are no ID keywords contained in the SCCS file) should be treated as an error. Also the **d** (default SID) flag specifies the dfault version of the SCCS file to be retrieved by the **get** command. The -f keyletter is used to set a flag and, possibly, to set its value. For example,

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

sets the **i** flag and the **m** (module name) flag. The value "modname" specified for the **m** flag is the value that the **get** command will use to replace the **sccs2** ID keyword. (In the absence of the **m** flag, the name of the *g-file* is used as the replacement for the **sccs2** ID keyword.) Note that several **−f** keyletters may be supplied on a single invocation of **admin** and that **−f** keyletters may be supplied whether the command is creating a new SCCS file or processing an existing one.

The **−d** keyletter is used to delete a flag from an SCCS file and may only be specified when processing an existing file. As an example, the command

```
admin -dm s.abc
```

removes the **m** flag from the SCCS file. Several **−d** keyletters may be supplied on a single invocation of **admin** and may be intermixed with **−f** keyletters.

The SCCS files contain a list (user list) of login names and/or group IDs of users who are allowed to create deltas. This list is empty by default which implies that anyone may create deltas. To add login names and/or group IDs to the list, the **−a** keyletter is used. For example,

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

adds the login names "xyz" and "wql" and the group ID "1234" to the list. The **−a** keyletter may be used whether **admin** is creating a new SCCS file or processing an existing one and may appear several times. The **−e** keyletter is used in an analogous manner if one wishes to remove (erase) login names or group IDs from the list.

D. The "prs" Command

The **prs** command is used to print on the standard output all or parts of an SCCS file in a format, called the output "data specification," supplied by the user via the **−d** keyletter. The data specification is a string consisting 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,

```
:I:
```

is defined as the data keyword that is replaced by the SID of a specified delta. Similarly, **:F:** is defined as the data keyword for the

SCCS file name currently being processed, and :C: is defined as the comment line associated with a specified delta. All parts of an SCCS file have an associated data keyword. For a complete list of the data keywords, see **prs**(1) in the Runtime System manual.

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

```
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 SID of that delta using the **−r** keyletter. For example,

```
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 the **−r** keyletter 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 by specifying the −l or **−e** keyletters. The −e keyletter substitutes data keywords for the SID designated via the **−r** keyletter and all deltas created earlier. The −l keyletter substitutes data keywords for the SID designated via the **−r** keyletter 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 the **-e** and **-l** keyletters.

E. The "help" Command

The **help** command prints explanations of SCCS commands and of messages that these commands may print. Arguments to **help**, zero or more of which may be supplied, are simply the names of SCCS commands or the code numbers that appear in parentheses after SCCS messages. If no argument is given, **help** prompts for one. The **help** command has no concept of keyletter arguments or file arguments. Explanatory information related to an argument, if it exists, is printed on the standard output. If no information is found, an error message is printed. Note that each argument is processed independently, and an error resulting from one argument will not terminate the processing of the other arguments.

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

```
help ge5 rmdel
```

produces

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

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

F. The "rmdel" Command

The **rmdel** command is provided to allow removal of a delta from an SCCS file. Its use should be reserved for those cases in which incorrect global changes were made a part of the delta to be removed.

The delta to be removed must be a "leaf" delta. That is, it must be the latest (most recently created) delta on its branch or on the trunk of the SCCS file tree. In Figure 4-3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed; once they are removed, then deltas 1.3.2.1 and 2.1 can be removed, etc.

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 either be the one who created the delta being removed or be the owner of the SCCS file and its directory.

The **−r** keyletter, which is mandatory, is used to specify the complete SID of the delta to be removed (i.e., it must have two components for a trunk delta and four components for a branch delta). Thus:

```
rmdel -r2.3 s.abc
```

specifies the removal of (trunk) delta "2.3" of the SCCS file. Before removal of the delta, **rmdel** checks that the release number (R) of the given SID satisfies the relation.

floor <= R <= ceiling

The **rmdel** command also checks that the SID specified is not that of a version for 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 Runtime System manual], the release specified *must* not be contained in the list. If these conditions are not satisfied, processing is terminated, and the delta is not removed. After the specified delta has been removed, its type indicator in the "delta table" of the SCCS file is changed from "D" ("delta") to "R" ("removed").

G. The "cdc" Command

The **cdc** command is used to change a delta's commentary that was supplied when that delta was created. Its invocation is analogous to that of the **rmdel** command, except that the delta to be processed is not required to be a leaf delta. For example,

cdc -r3.4 s.abc

specifies that the commentary of delta "3.4" of the SCCS file is to be changed.

The new commentary is solicited by **cdc** in the same manner as that of **delta**. The old commentary associated with the specified delta is kept, but it is preceded by a comment line indicating that it has been changed (i.e., superseded), and the new commentary is entered ahead of this 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 deletion of selected MR numbers associated with the specified delta. This is specified by preceding the selected MR numbers by the character "!".

Thus:

```
cdc -r1.4 s.abc
MRs? mrnum3 !mrnum1
comments? deleted wrong MR number and inserted
 correct MR number
```

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

H. The "what" Command

The **what** command is used to find identifying information within any UNIX system file whose name is given as an argument to **what**. Directory names and a name of "-" (a lone minus sign) are not treated specially as they are by other SCCS commands and no keyletters are accepted by the command.

The **what** command searches the given file(s) for all occurrences of the string "@(#)", which is the replacement for the @(#) ID keyword [see **get**(1)], and prints (on the standard output) the balance following that string until the first double quote ("), greater than (>), backslash (\), newline, or (nonprinting) NUL character. For example, if the SCCS file *s.prog.c* (a C language program) contains the following line:

```
char id[] " @(#)sccs2:5.1";
```

and then the command

```
get -r3.4 s.prog.c
```

is executed, 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.

I. The "sccsdiff" Command

The **sccsdiff** command determines (and prints on the standard output) the differences between two specified versions of one or more SCCS files. The versions to be compared are specified by using the **−r** keyletter, whose format is the same as for the **get** command. The two versions must be specified as the first two arguments to this command in the order they were created, i.e., the older version is specified first. Any following keyletters are interpreted as arguments to the **pr**(1) command (which actually prints the differences) and must appear before any file names. The SCCS files to be processed are named last. Directory names and a name of "-" (a lone minus sign) are not acceptable to **sccsdiff**.

The differences are printed in the form generated by **diff**(1). The following is an example of the invocation of **sccsdiff**:

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

J. The "comb" Command

The **comb** command generates a "shell procedure" [see **sh**(1) in the Runtime System manual] which attempts to reconstruct the named SCCS files so that the reconstructed files are smaller than the originals. The generated shell procedure is written on the standard output. Named SCCS files are reconstructed by discarding unwanted deltas and combining other specified deltas. The SCCS files that contain deltas no longer useful should be discarded. It is not recommended that **comb** be used as a matter of routine; its use should be restricted to a very small number of times in the life of an SCCS file.

In the absence of any keyletters, **comb** preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the "shape" of the SCCS file tree. The effect of this is to eliminate middle deltas on the trunk and on all branches of the tree. Thus, in Figure 4-3, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated. Some of the keyletters are summarized as follows:

The **−p** keyletter specifies the oldest delta that is to be preserved in the reconstruction. All older deltas are discarded.

The -c keyletter specifies a list [see **get**(1) in the Runtime System manual for the syntax of such a list] of deltas to be preserved. All other deltas are discarded.

The **−s** keyletter causes the generation of a shell procedure, which when run, produces only a report summarizing the percentage space (if any) to be saved by reconstructing each named SCCS file. It is recommended that **comb** be run with this keyletter (in addition to any others desired) before any actual reconstructions.

C-7

It should be noted that the shell procedure generated by **comb** is not guaranteed to save space. In fact, it is possible for the reconstructed file to be larger than the original. Note, too, that the shape of the SCCS file tree may be altered by the reconstruction process.

K. The "val" Command

The **val** command is used to determine if a file is an SCCS file meeting the characteristics specified by an optional list of keyletter arguments. Any characteristics not met are considered errors.

The **val** command checks for the existence of a particular delta when the SID for that delta is explicitly specified via the **-r** keyletter. The string following the **-y** or **-m** keyletter is used to check the value set by the **t** or **m** flag, respectively [see **admin**(1) in the Runtime System manual for a description of the flags].

The **val** command treats the special argument "−" differently from other SCCS commands. This argument allows **val** to read the

argument list from the standard input as opposed to obtaining it from the command line. The standard input is read until end of file. This capability allows for one invocation of **val** with different values for the keyletter and file arguments. For example,

```
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 processing of the first file is completed, **val** then processes the remaining files, in this case, *s.xyz*, to determine if they meet the characteristics specified by the keyletter arguments associated with them.

The **val** command returns an 8-bit code; each bit set indicates the occurrence of a specific error [see **val**(1) for a description of possible errors and the codes]. In addition, an appropriate diagnostic is printed unless suppressed by the **−s** keyletter. A return code of "0" indicates all named files met the characteristics specified.

SCCS FILES

This part discusses several topics that must be considered before extensive use is made of SCCS. These topics deal with the protection mechanisms relied upon by SCCS, the format of SCCS files, and the recommended procedures for auditing SCCS files.

A. Protection

The SCCS relies on the capabilities of the UNIX software for most of the protection mechanisms required to prevent unauthorized changes to SCCS files (i.e., changes made by non-SCCS commands). The only protection features provided directly by SCCS are the "release lock" flag, the "release floor" and "ceiling" flags, and the "user list".

New SCCS files created by the **admin** command are given mode 444 (read-only). It is recommended that this mode *remain unchanged* as it prevents any direct modification of the files by non-SCCS

commands. It is further recommended that the directories containing SCCS files be given mode 755 which allows only the owner of the directory to modify its contents.

The SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies protection and auditing of SCCS files. The contents of directories should correspond to convenient logical groupings, e.g., subsystems of a large project.

The SCCS files must have only one link (name) because the commands that modify SCCS files do so by creating a copy of the file (the *x-file,* see "SCCS COMMAND CONVENTIONS"). Upon completion of processing, remove the old file and rename the *x-file.* If the old file has more than one link, this would break such additional links. Rather than process such files, SCCS commands produce an error message. All SCCS files *must* have names that begin with "s.".

When only one user 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.

However, in those situations in which several users with unique user IDs are assigned responsibility for one SCCS file (e.g., in large software development projects), one user (equivalently, one user ID) must be chosen as the "owner" of the SCCS files and be the one who will "administer" them (e.g., by using the **admin** command). This user is termed the "SCCS administrator" for that project. Because other users of SCCS 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 Runtime System manual]. This assures that the effective user ID is the user ID of the

administrator. This program invokes the desired SCCS command and causes it to inherit the privileges of the interface program for the duration of that command's execution. Thus, 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. Other users are thus able to modify the SCCS files only through the use of **delta** and, possibly, **rmdel** and **cdc**. The project-dependent interface program, as its name implies, must be custom-built for each project.

B. Formatting

The 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 this 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 various SCCS commands.
Descriptive Text	Arbitrary text provided by the user; usually a summary of the contents and purpose of the file.
Body	Actual text that is being administered by SCCS, intermixed with internal SCCS control lines.

Detailed information about the contents of the various sections of the file may be found in **sccsfile**(5). The checksum is the only portion of the file that is of interest below.

It is important to note that because SCCS files are ASCII files they may be processed by various UNIX software commands, such as **ed**(1), **grep**(1), and **cat**(1). This is very convenient in those instances in which an SCCS file must be modified manually (e.g., when the time and date of a delta was recorded incorrectly because the system clock was set incorrectly) or when it is desired to simply look at the file.

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

C. Auditing

On rare occasions, perhaps due to an operating system or hardware malfunction, an SCCS file or portions of it (i.e., one or more "blocks") can be destroyed. The SCCS commands (like most UNIX software commands) issue an error message when a file does not exist. In addition, SCCS commands use the checksum stored in the SCCS file to determine whether a 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 the **−h** or **−z** keyletters, as described below.

It is recommended that SCCS files be audited for possible corruptions on a regular basis. The simplest and fastest way to perform an audit is to execute the **admin** command with the **−h** keyletter on all SCCS files.

```
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. This process continues until all the files have been examined. When examining directories (as in the second

example above), the process just described will not detect missing files. A simple way to detect whether any files are missing from a directory is to periodically execute the **ls**(1) command on that directory and compare the outputs of the most current and the previous executions. Any file whose name appears in the previous output but not in the current one has been removed by some means.

Whenever a file has been corrupted, the manner in which the file is restored depends upon the extent of the corruption. If damage is extensive, the best solution is to contact the local UNIX system operations group and request that the file be restored from a backup copy. In the case of minor damage, repair through use of the editor **ed**(1) may be possible. In the latter case after such repair, the following command must be executed:

```
admin -z s.file
```

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

AN SCCS INTERFACE PROGRAM

A. General

In order to permit UNIX system users [with different user identification numbers (user IDs)] to use SCCS commands upon the same files, an SCCS interface program is provided. It temporarily grants the necessary file access permissions to these users. This part discusses the creation and use of such an interface program. The SCCS interface program may also be used as a preprocessor to SCCS commands since it can perform operations upon its arguments.

B. Function

When only one user uses SCCS, the real and effective user IDs are the same; and that user's ID owns the directories containing SCCS files. However, there are situations (e.g., in large software development projects) in which it is practical to allow more than one user to make changes to the same set of SCCS files. In these cases, one user must be chosen as the "owner" of the SCCS files and be the one who will "administer" them (e.g., by using the **admin** command). This user is termed the "SCCS administrator" for that project. Since other users of SCCS do not have the same privileges and permissions as the SCCS administrator, the other users 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, cdc,** and **unget** commands. Other SCCS commands either do not require write permission in the directory containing SCCS files or are (generally) reserved for use only by the administrator.

The interface program

- Must be owned by the SCCS administrator
- Must be executable by the new owner
- Must have the "set user on execution" bit "on" [see **chmod(1)** in the Runtime System manual].

Then when executed, the effective user ID is the user ID of the administrator. This program's function is to invoke the desired SCCS command and to cause it to inherit the privileges of the SCCS administrator for the duration of that command's execution. In this manner, the owner of an SCCS file (the administrator) can modify it at will. Other users whose login names are in the user list for that file (but who are not its owners) are given the necessary permissions only for the duration of the execution of the interface program. They are thus able to modify the SCCS files only through the use of **delta** and, possibly, **rmdel** and **cdc.**

C. Basic Program

When a UNIX system program is executed, the program is passed as argument 0, which is the name that invoked the program, and followed by any additional user-supplied arguments. Thus, if a program is given a number of links (names), the program may alter its processing depending upon which link invokes the program. This mechanism is used by an SCCS interface program to determine the SCCS command it should subsequently invoke [see exec(2) in the Software Development System manual.

A generic interface program (*inter.c*, written in C language) is shown in Figure 4-5. Note the reference to the (unsupplied) function "filearg". This is intended to demonstrate that the interface program may also be used as a preprocessor to SCCS commands. For example, function "filearg" could be used to modify file arguments to be passed to the SCCS command by supplying the full pathname of a file, thus avoiding extraneous typing by the user. Also, the program could supply any additional (default) keyletter arguments desired.

D. Linking and Use

In general, the following demonstrates the steps to be performed by the SCCS administrator to create the SCCS interface program. It is assumed, for the purposes of the discussion, that the interface program **inter.c** resides in directory "/x1/xyz/sccs". Thus, the command sequence

```
cd /x1/xyz/sccs
cc ... inter.c -o inter ...
```

compiles **inter.c** to produce the executable module **inter** (the "..." represents other arguments that may be required). The proper mode and the "set user ID on execution" bit are set by executing

```
chmod 4755 inter
```

For example, new links are created by

```
ln inter get
ln inter delta
ln inter rmdel
```

The names of the links may be arbitrary if the interface program is able to determine from them the names of SCCS commands to be invoked. Subsequently, any user whose shell parameter PATH [see **sh**(1) in the Runtime System manual] specifies directory "/xl/xyz/sccs" as the one to be searched first for executable commands may execute, e.g.

```
get -e /xl/xyz/sccs/s.abc
```

from any directory to invoke the interface program (via its link "get"). The interface program then executes "/usr/bin/get" (the actual SCCS **get** command) upon the named file. As previously mentioned, the interface program could be used to supply the pathname "/xl/xyz/sccs" so that the user would only have to specify

C-7

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

to achieve the same results.

Chapter 8

THE M4 MACRO PROCESSOR

GENERAL

The M4 macro processor is a front end for rational Fortran (Ratfor) and the C programming languages. The "#define" statement in C language and the analogous "define" in Ratfor are examples of the basic facility provided by any macro processor.

At the beginning of a program, a symbolic name or symbolic constant can be defined as a particular string of characters. The compiler will then replace later unquoted occurrences of the symbolic name with the corresponding string. Besides the straightforward replacement of one string of text by another, the M4 macro processor provides the following features:

- arguments
- arithmetic capabilities
- file manipulation
- conditional macro expansion
- string and substring functions.

The basic operation of M4 is to read every alphanumeric token (string of letters and digits) input and determine if the token is the name of a macro. The name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be rescanned. Macros may be called with arguments. The arguments are collected and substituted into the right places in the defining text before the defining text is rescanned.

The user also has the capability to define new macros. Built-ins and user-defined macros work exactly the same way except that some of the built-in macros have side effects on the state of the process. A

list of 21 built-in macros provided by the M4 macro processor can be found in Figure 5-1.

Macro Name	Function
changequote	Restores original characters or makes new quote characters the left and right brackets.
changescom	Changes left and right comment markers from the default # and new line.
deer	Returns the value of its argument decremented by 1.
define	Defines new macros.
defn	Returns the quoted definition of its argument(s).
divert	Diverts output to 1-out-of-10 diversions.

Figure 5-1. Built-in Macros (Sheet 1 of 4)

Macro Name	Function
divnum	Returns the number of the currently active diversion.
dnl	Reads and discards characters up to and including the next new line.
dumpdef	Dumps the current names and definitions of items named as arguments.
errprint	Prints its arguments on the standard error file.
eval	Prints arbitrary arithmetic on integers.
ifdef	Determines if a macro is currently defined.
ifelse	Performs arbitrary conditional testing.
include	Returns the contents of the file named in the argument. A fatal error occurs if the file name cannot be accessed.

Figure 5-1. Built-in Macros (Sheet 2 of 4)

Macro Name	Function
iner	Returns the value of its argument incremented by 1.
index	Returns the position where the second argument begins in the first argument pf index.
len	Returns the number of characters that makes its argument.
m4exit	Causes immediate exit from M4.
m4wrap	Pushes the exit code back at final EOF.
maketemp	Facilitates making unique file names.
popdef	Removes current definition of its argument(s) exposing any previous definitions.
pushdef	Defines new macros but saves any previous definition.

Figure 5-1. Built-in Macros (Sheet 3 of 4)

Macro Name	Function
shift	Returns all arguments of shift except the first argument.
sinclude	Returns the contents of the file named in the arguments. The macro remains silent and continues if the file is inaccessible.
substr	Produces substrings of strings.
syscmd	Executes the UNIX System command given in the first argument.
traceoff	Turns macro trace off.
traceon	Turns the macro trace on.
translit	Performs character transliteration.
undefine	Removes user-defined or built-in macro definitions.
undivert	Discards the diverted text.

Figure 5-1. Built-in Macros (Sheet 4 of 4)

To use the M4 macro processor, input the following command:

m4 [optional files]

Each argument file is processed in order. If there are no arguments or if an argument is "-", the standard input is read at that point. The processed text is written on the standard output which may be captured for subsequent processing with the following input:

```
m4 [files] >outputfile
```

DEFINING MACROS

The primary built-in function of M4 is **define**. **Define** is used to define new macros. The following input:

```
define(name, stuff)
```

causes the string *name* to be defined as *stuff*. All subsequent occurrences of *name* will be replaced by *stuff*. *Name* must be alphanumeric and must begin with a letter (the underscore counts as a letter). *Stuff* is any text that contains balanced parentheses. Use of a slash may stretch *stuff* over multiple lines. Thus, as a typical example,

```
define(N, 100)
...
if (i > N)
```

defines *N* to be 100 and uses the symbolic constant *N* in a later **if** statement.

The left parenthesis must immediately follow the word **define** to signal that **define** has arguments. If a user-defined macro or built-in name is not followed immediately by "(", it is assumed to have no arguments. Macro calls have the following general form:

```
name(arg1,arg2,...argn)
```

A macro name is only recognized as such if it appears surrounded by nonalphanumerics. Using the following example:

```
define(N, 100)
...
if (NNN > 100)
```

the variable *NNN* is absolutely unrelated to the defined macro *N* even though the variable contains a lot of *N*s.

Macros may be defined in terms of other names. For example,

```
define(N, 100)
define(M, N)
```

defines both *M* and *N* to be 100. If *N* is redefined and subsequently changes, *M* retains the value of 100 not *N*.

C-8

The M4 macro processor expands macro names into their defining text as soon as possible. The string *N* is immediately replaced by 100. Then the string *M* is also immediately replaced by 100. The overall result is the same as using the following input in the first place:

```
define(M, 100)
```

The order of the definitions can be interchanged as follows:

```
define(M, N)
define(N, 100)
```

Now *M* is defined to be the string *N*, so when the value of *M* is requested later, the result is the value of *N* at that time (because the *M* will be replaced by *N* which will be replaced by 100).

The more general solution is to delay the expansion of the arguments of **define** by quoting them. Any text surrounded by left and right single quotes is not expanded immediately but has the quotes stripped off. The value of a quoted string is the string stripped of the quotes. If the input is

```
define(N, 100)
define(M, 'N')
```

the quotes around the *N* are stripped off as the argument is being collected. The results of using quotes is to define *M* as the string *N*, not 100. The general rule is that M4 always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If the word **define** is to appear in the output, the word must be quoted in the input as follows:

```
'define' = 1;
```

Another example of using quotes is redefining *N*. To redefine *N*, the evaluation must be delayed by quoting

```
define(N, 100)
...
define('N', 200)
```

In M4, it is often wise to quote the first argument of a macro. The following example will not redefine *N*:

```
define(N, 100)
...
define(N, 200)
```

The *N* in the second definition is replaced by 100. The result is equivalent to the following statement:
define(100, 200)

This statement is ignored by M4 since only things that look like names can be defined.

If left and right single quotes are not convenient for some reason, the quote characters can be changed with the following built-in macro:

```
changequote([, ])
```

The built-in **changequote** makes the new quote characters the left and right brackets. The original characters can be restored by using **changequote** without arguments as follows:

```
changequote
```

There are two additional built-ins related to **define**. The **undefine** macro removes the definition of some macro or built-in as follows:

```
undefine('N')
```

The macro removes the definition of *N*. Built-ins can be removed with **undefine**, as follows:

```
undefine('define')
```

But once removed, the definition cannot be reused.

The built-in **ifdef** provides a way to determine if a macro is currently defined. Depending on the system, a definition appropriate for the particular machine can be made as follows:

```
ifdef('pdp11', 'define(wordsize,16)')
ifdef('u3b', 'define(wordsize,32)')
```

Remember to use the quotes.

The **ifdef** macro actually permits three arguments. If the first argument is defined, the value of **ifdef** is the second argument. If the first argument is not defined, the value of **ifdef** is the third argument. If there is no third argument, the value of **ifdef** is null. If the name is undefined, the value of **ifdef** is then the third argument, as in

```
ifdef('unix', on UNIX, not on UNIX)
```

ARGUMENTS

So far the simplest form of macro processing has been discussed which is replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**), any occurrence of **$n** is replaced by the **n**th argument when the macro is actually used. Thus, the macro **bump** defined as

```
define(bump, $1 = $1 + 1)
```

generates code to increment its argument by 1. The 'bump(x)' statement is equivalent to 'x = x + 1.'

A macro can have as many arguments as needed, but only the first nine are accessible (**$1** through **$9**). The macro name is **$0** although that is less commonly used. Arguments that are not supplied are replaced by null strings, so a macro can be defined which simply concatenates its arguments like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus, 'cat(x, y, z)' is equivalent to 'xyz'. Arguments **$4** through **$9** are null since no corresponding arguments were provided. Leading

unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus:

```
define(a,  b  c)
```

defines 'a' to be 'b c'.

Arguments are separated by commas; however, when commas are within parentheses, the argument is not terminated nor separated. For example,

```
define(a, (b,c))
```

has only two arguments. The first argument is **a**. The second is literally **(b,c)**. A bare comma or parenthesis can be inserted by quoting it.

ARITHMETIC BUILT-INS

The M4 provides three built-in functions for doing arithmetic on integers (only). The simplest is **incr** which increments its numeric argument by 1. The built-in **decr** decrements by 1. Thus to handle the common programming situation where a variable is to be defined as "one more than *N*", use the following:

```
define(N, 100)
define(N1, 'incr(N)')
```

Then *N1* is defined as one more than the current value of *N*.

The more general mechanism for arithmetic is a built-in called **eval** which is capable of arbitrary arithmetic on integers. The operators in decreasing order of precedence are

unary + and -
** or ^ (exponentiation)
* / % (modulus)
+ -
== != < <= > >=
! (not)
& or && (logical and)
| or || (logical or).

Parentheses may be used to group operations where needed. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1 and false is **0**. The precision in **eval** is 32 bits under the UNIX operating system.

As a simple example, define *M* to be "2==N+1" using **eval** as follows:

```
define(N, 3)
define(M, 'eval(2==N+1)')
```

The defining text for a macro should be quoted unless the text is very simple. Quoting the defining text usually gives the desired result and is a good habit to get into.

FILE MANIPULATION

A new file can be included in the input at any time by the built-in function **include**. For example,

```
include(filename)
```

inserts the contents of *filename* in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (**include**'s replacement text) is the contents of the file. If needed, the contents can be captured in definitions, etc.

A fatal error occurs if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used. The built-in **sinclude** (silent include) says nothing and continues if the file named cannot be accessed.

The output of M4 can be diverted to temporary files during processing, and the collected material can be output upon command. The M4 maintains nine of these diversions, numbered 1 through 9. If the built-in macro

```
divert(n)
```

is used, all subsequent output is put onto the end of a temporary file referred to as *n*. Diverting to this file is stopped by the **divert** or **divert(0)** command which resumes the normal output process.

Diverted text is normally output all at once at the end of processing with the diversions output in numerical order. Diversions can be brought back at any time by appending the new diversion to the current diversion. Output diverted to a stream other than 0 through 9 is discarded. The built-in **undivert** brings back all diversions in numerical order. The built-in **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted text (as does diverting) into a diversion whose number is not between 0 and 9, inclusive.

The value of **undivert** is *not* the diverted text. Furthermore, the diverted material is *not* rescanned for macros. The built-in **divnum** returns the number of the currently active diversion. The current output stream is zero during normal processing.

SYSTEM COMMAND

Any program in the local operating system can be run by using the **syscmd** built-in. For example,

```
syscmd(date)
```

on the UNIX system runs the **date** command. Normally, **syscmd** would be used to create a file for a subsequent **include**. To facilitate making unique file names, the built-in **maketemp** is provided with specifications identical to the system function *mktemp*. The **maketemp** macro fills in a string of XXXXX in the argument with the process id of the current process.

CONDITIONALS

Arbitrary conditional testing is performed via built-in **ifelse**. In the simplest form

```
ifelse(a, b, c, d)
```

compares the two strings *a* and *b*. If *a* and *b* are identical, **ifelse** returns the string *c*. Otherwise, string *d* is returned. Thus, a macro called **compare** can be defined as one which compares two strings and returns "yes" or "no" if they are the same or different as follows:

```
define(compare, 'ifelse($1, $2, yes, no)')
```

Note the quotes which prevent evaluation of **ifelse** occurring too early. If the fourth argument is missing, it is treated as empty.

The built-in **ifelse** can actually have any number of arguments and provides a limited form of multiway decision capability. In the input

```
ifelse(a, b, c, d, e, f, g)
```

if the string *a* matches the string *b*, the result is *c*. Otherwise, if *d* is the same as *e*, the result is *f*. Otherwise, the result is *g*. If the final argument is omitted, the result is null, so

```
ifelse(a, b, c)
```

is *c* if *a* matches *b*, and null otherwise.

STRING MANIPULATION

The built-in **len** returns the length of the string (number of characters) that makes up its argument. Thus:

```
len(abcdef)
```

is 6, and **len((a,b))** is 5.

The built-in **substr** can be used to produce substrings of strings. Using input, **substr(s, i, n)** returns the substring of *s* that starts at the *i*th position (origin zero) and is *n* characters long. If *n* is omitted, the rest of the string is returned. Inputting

```
substr('now is the time',1)
```

returns the following string:

```
ow is the time.
```

If *i* or *n* are out of range, various actions occur.

The built-in **index(s1, s2)** returns the index (position) in *s1* where the string *s2* occurs or -1 if it does not occur. As with **substr**, the origin for strings is 0.

The built-in **translit** performs character transliteration and has the general form

```
translit(s, f, t)
```

which modifies *s* by replacing any character found in *f* by the corresponding character of *t*. Using input

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If *t* is shorter than *f*, characters that do not have an entry in *t* are deleted. As a limiting case, if *t* is not present at all, characters from *f* are deleted from *s*. So

```
translit(s, aeiou)
```

would delete vowels from *s*.

There is also a built-in called **dnl** that deletes all characters that follow it up to and including the next new line. The **dnl** macro is useful mainly for throwing away empty lines that otherwise tend to clutter up M4 output. Using input

```
define(N, 100)
define(M, 200)
define(L, 300)
```

results in a new line at the end of each line that is not part of the definition. So the new line is copied into the output where it may not be wanted. If the built-in **dnl** is added to each of these lines, the newlines will disappear. Another method of achieving the same results is to input

```
divert(-1)
define(...)
...
divert.
```

PRINTING

The built-in **errprint** writes its arguments out on the standard error file. An example would be

```
errprint('fatal error')
```

The built-in **dumpdef** is a debugging aid that dumps the current names and definitions of items named as arguments. If no arguments are given, then all current names and definitions are printed. Do not forget to quote the names.

Chapter 9

THE LINK EDITOR

GENERAL

The link editor [*ld*(**1**)*] is a UNIX system support tool used on the VAX† processor and on all processors in the 3B‡ Computer family. The *ld* creates executable object files by combining object files, performing relocation, and resolving external references. The *ld* also processes symbolic debugging information. The inputs to *ld* are relocatable object files produced either by the compiler [*cc*(**1**)], the assembler [*as*(**1**)], or by a previous ld run. The *ld* combines these object files to form either a relocatable or an absolute (i.e., executable) object file.

The *ld* also supports a command language that allows users to control the *ld* process with great flexibility and precision. The UNIX system *ld* shares most of its source with other lds in use on other processors and operating systems. Therefore, the UNIX system ld provides many powerful features that may or may not be useful on a UNIX system.

C-9

Although the link edit process is controlled in detail through use of the *ld* command language described later, most users do *not* require this degree of flexibility, and the manual page obtained by typing

```
man ld
```

is sufficient instruction in the use of ld.

The command language (described later) supports the ability to

---

* See section 1 of the Runtime System manual

† Trademark of Digital Equipment Corporation

‡ Trademark of Western Electric Company

- Specify the memory configuration of the machine
- Combine object file sections in particular fashions
- Cause the files to be bound to specific addresses or within specific portions of memory
- Define or redefine global symbols at link edit time.

There are several concepts and definitions with which you should familiarize yourself before proceeding further.

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 RAM (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 the ld. *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.

Section

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 global symbols with an assignment statement in the *ld* command language.

C-9

Object File

Object files are produced both by the assembler (typically as a result of calling the compiler) and by the ld. The *ld* 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 the *ld* can also be absolute files.

Files produced from the compiler/assembler always contain three sections, called *.text*, *.data*, and *.bss*. The *.text* section contains the instruction text (for example, executable instructions), *.data* contains initialized data variables, and *.bss* contains uninitialized data variables. For example, if a C program contains the global (that is, not inside a function) declarations called *.text*, *.data* and *.bss*; the *.text* section contains the instruction text (e.g., executable instructions), *.data* contains initialized data variables, and *.bss* contains uninitialized data variables. For example, if a C program contained the global (i.e., not inside a function) declarations

```
int i = 100;
char abc[200];
```

and the assignment

```
abc[i] = 0;
```

then compiled code from the C assignment is stored in *.text*. The variable i is located in *.data*, and abc is located in *.bss*. There is an exception to the rule however; both initialized and uninitialized statics are allocated into the *.data* section. The value of an uninitialized static in a *.data* section is zero.

USING THE LINK EDITOR

The *ld* is called by the command

```
ld  [options]  filename1 filename2 . . .
```

Files passed to the *ld* must be object files, archive libraries containing object files, or text source files containing *ld* directives. The *ld* uses the "magic number" (in the first two bytes of the file) to determine which type of file is encountered. If the *ld* does not recognize the magic number, it assumes the file is a text file containing *ld* directives and attempts to parse it.

Input object files and archive libraries of object files are linked together to form an output object file. If there are no unresolved references, this file is executable *on the target machine*. An input file containing directives is referred to as an *ifile* in this document. Object files have the form "name.o" throughout the examples in this chapter. The names of actual input object files need not follow this convention.

If you merely want to link the object files file1.o and file2.o, the following command is sufficient.

```
ld  file1.o  file2.o
```

No directives to the *ld* are needed. If no errors are encountered during the link edit, the output is left on the default file a.out. The sections of the input files are combined in order. That is, if file1.o and file2.o each contain the standard sections *.text*, *.data*, and *.bss*, the output object file also contains these three sections. The output *.text* section is a concatenation of *.text* from file1.o and *.text* from file2.o. The *.data* and *.bss* sections are formed similarly. The output *.text* section is then bound (with the exception of 3B 5 Computers) at address 0X000000. The output *.data* and *.bss* sections are link edited together into contiguous addresses (the particular address depending on the particular processor).

Instead of entering the names of files to be link edited (as well as *ld* options on the *ld* command line), this information can be placed into an ifile, and just the ifile passed to ld. For example, if you are going to frequently link the object files file1.o, file2.o, and file3.o with the same options f1 and f2, then enter the command

```
ld -f1  -f2  file1.o  file2.o  file3.o
```

each time it is necessary to invoke ld. Alternatively, an ifile containing the statements

```
-f1
-f2
file1.o
file2.o
file3.o
```

could be created, and then the following UNIX system command would serve:

```
ld  ifilename
```

Note that it is perfectly permissible to specify some of the object files to be link edited in the ifile and others on the command line—as well as some *options* in the ifile and others on the command line. Input

object files are link edited in the order they are encountered, whether this occurs on the command line or in an ifile. As an example, if a command line were

```
ld  file1.o  ifile  file2.o
```

and the ifile contained

```
file3.o
file4.o
```

then the order of link editing would be: file1.o, file3.o, file4.o, and file2.o. Note from this example that an ifile is read *and processed* immediately upon being encountered in the command line.

Options may be interspersed with file names both on the command line and in an ifile. The ordering of options is not significant, except for the "l" and "L" options for specifying libraries. The "l" option is a shorthand notation for specifying an archive library, and an archive library is just a collection of object files. Thus, as is the case with any object file, libraries are searched as they are encountered. The "L" specifies an alternative directory for searching for libraries. Therefore, to be effective, a "-L" option must appear before any "-l" options.

All options for *ld* must be preceded by a hyphen (-) whether in the ifile or on the *ld* command line. Options that have an argument (except for the "-l" and "-L" options) are separated from the argument by white space (blanks or tabs). The following options (in alphabetical order) are supported, though not all options are available on each processor.

-a Produces an absolute, executable file. Messages are issued when undefined symbols are found, and several special symbols (such as "_end") are defined. Unless overridden by the "-r" option, relocation information is stripped from the output file. If neither "-r" nor "-a" is specified, "-a" is assumed. This flag applies only to the 3B 5 Computers.

-e ss Defines the primary entry point of the output file to be the symbol given by the argument "ss". See " Changing the Entry Point" in "NOTES AND SPECIAL CONSIDERATIONS" for a discussion of how the option is used.

-f bb Sets the default fill value. This value is used to fill "holes" formed within output sections. Also, it is used to initialize input *.bss* sections when they are combined with other non—*.bss* input sections. The argument "bb" is a 2-byte constant. If the "-f" option is not used, the default fill value is zero.

-lx Specifies a UNIX system archive library file as *ld* input. The argument is a character string (less than 10 characters) immediately following the "-l" without any intervening white space. As an example, -lc refers to libc.a, -lC to libC.a, etc. The given archive library must contain valid object files as its members.

-m Produces a map or listing of the input/output sections (including "holes") on the standard output.

-o name Names the output object file. The argument "name" is the name of the UNIX system file to be used as the output file. The default output object file name is "a.out". The "name" can be a full or partial UNIX system pathname.

-r Retains relocation entries in the output object file. Relocation entries must be saved if the output file is to be used as an input file in a subsequent *ld* call. If the -r option is used, unresolved references do not prevent the creation of an output object file.

-s Strips line number entries and symbol table information from the output object file. Relocation entries ("-r" option) are meaningless without the symbol table, hence use of "-s" precludes the use of "-r". All symbols are stripped, including global and undefined symbols.

-t Disables checking that all instances of a multiply-defined symbol are the same size.

-u sym Introduces an unresolved external symbol into the output file's symbol table. The argument "sym" is the name of the symbol. This is useful for linking entirely from a library, since initially the symbol table is empty and an unresolved reference is needed to force the linking of an initial routine from the library.

-x Does not preserve any local (non-global) symbols in the output symbol table; enter external and static symbols only. This option saves some space in the output file.

-z Configures memory so that nothing may be placed at address zero. This is used to catch references through null pointers. This option is overridden if any section or memory directives are used. This option applies only to the 3B 5 Computers.

-F The magic number 0413 is stored in the UNIX system header indicating that the file should be paged. This option applies only to 3B 5 Computers.

-H Changes the type of all global symbols to "static". This option can be used to "hide" symbols since static symbols have different accessing rules from global symbols.

-Ldir Changes the algorithm for searching for libraries to look in *dir* before looking in the default location. This option is for *ld* libraries as the -I option is for compiler #include files. The "-L" option is useful for finding libraries that are not in the standard library directory. To be useful, this option must appear before the "-l" option.

-M Prints a warning message for all external variables that are multiply-defined.

-N Places the data section immediately following the text section in memory and stores the magic number 0407 in the UNIX system header. This prevents the text from being shared (the default).

-S Requests a "silent" *ld* run. All error messages resulting from errors that do not immediately stop the *ld* run are suppressed.

-V Prints on the standard error output a "version id" identifying the *ld* being run.

-VS num Takes *num* as a decimal version number identifying the *a.out* file that is produced. The version stamp is stored in the UNIX system header.

LINK EDITOR COMMAND LANGUAGE

Expressions

Expressions may contain global symbols, constants, and most of the basic C language operators. (See Figure 7-2, "SYNTAX DIAGRAM FOR INPUT DIRECTIVES".) Constants are as in C with a number recognized as decimal unless preceded with "0" for octal or "0x" for hexadecimal. All numbers are treated as long ints. 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. The *ld* 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.

The *ld* 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:

ALIGN	DSECT	MEMORY	PHY	SECTIONS
ASSIGN	GROUP	NOLOAD	RANGE	SPARE
BLOCK	LENGTH	ORIGIN	REGION	TV

align	group	length	origin	spare
assign	l	o	phy	
block	len	org	range	

The operators that are supported, in order of precedence from high to low, are shown in Figure 7-1.

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

Figure 7-1. Symbols and Functions of Operators

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

Assignment Statements

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

symbol = expression;

or

symbol op= expression;

where *op* is one of the operators *+, –, *, or /.*

Assignment statements must be terminated by a semicolon.

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

Assignment statements are normally placed outside the scope of section-definition directive (see " Section Definition Directive" under " LINK EDITOR COMMAND LANGUAGE"). However, there exists a special symbol, called ".", that can occur only *within* a section-definition directive. This symbol refers to the *current R address of the ld's location counter. Thus, assignment expressions involving "." are evaluated during the allocation phase of ld.* Assigning a value to the "." symbol within a section-definition directive increments/resets ld'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)
```

Link editor expressions may have either an absolute or a relocatable value. When the *ld* 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 value is in relation to the section of the referenced symbol.

- All other expressions have absolute values.

Specifying a Memory Configuration

MEMORY directives are used to specify

a. The total size of the virtual space of the target machine.

b. The configured and unconfigured areas of the virtual space.

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

By means of MEMORY directives, an arbitrary name of up to eight characters is assigned to a virtual address range. Output sections can then be forced to be bound to virtual addresses within specifically *named* memory areas. Memory names may contain uppercase or lowercase letters, digits, and the special characters '$', '.', or '_'. Names of memory ranges are used by *ld* 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 the ld's allocation process, and hence nothing can be link edited, bound, or assigned to any address within unconfigured memory.

As an option on the MEMORY directive, *attributes* may be associated with a named memory area. This restricts the memory areas (with specific attributes) to which an output section can be bound. The attributes assigned to output sections in this manner are recorded in the appropriate section headers in the output file to allow for possible error checking in the future. For example, putting a text section into writable memory is one potential error condition. Currently, error checking of this type is not implemented.

The attributes currently accepted are

a. R : readable memory.

b. W : writable memory.

c. X : executable, i.e. instructions may reside in this memory.

d. 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 W, R, I, and X.

The syntax of the MEMORY directive is

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

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

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

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

Section Definition Directives

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

In the default case where no SECTIONS directives are given, all input sections of the same name appear in an output section of that name. For example, if a number of object files from the compiler are linked, each containing the three sections *.text*, *.data*, and *.bss*, the output object file also contains three sections, *.text*, *.data*, and *.bss*. If two object files are linked (one that contains 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.

If a file name appears with no sections listed, then *all* sections from the file are linked into the current output section. For example,

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

The order in which the input sections appears in the output section "outsec1" is given by

a. Section sec1 from file file1.o

b. All sections from file2.o, in the order they appear in the file

c. Section sec1 from file file3.o, and then section sec2 from file file3.o

If there are any additional input files that contained 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 file1.0 or file3.0, they will be placed in output sections with the same names as the input sections.

Load a Section at a Specified Address

Bonding of an output section to a specific virtual address is accomplished by an *ld* option as shown on 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 issues an appropriate error message.

So 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* in any particular order.

The *ld* 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. The *ld* 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, the ld 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 linked to 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

a. Links all input *.text* sections together into one output section. This output section is called *.text* and is bound to an address of 0x0.

b. Links all input *.data* sections together into one output section. This output section is called *.data* and (with the exception of 3B 5 Computers) is bound to an address aligned to a machine dependent constant.

c. Links all input *.bss* sections together into one output section. This output section is called *.bss* and is allocated so as to immediately follow the output section *.data*. Note that the output section *.bss* is *not given any particular address alignment.*

Specifying any SECTIONS directives results in this default allocation *not* being performed.

The default allocation of *ld* is equivalent to supplying the following directive:

```
SECTIONS
{
        .text  : { }
        GROUP ALIGN( align_value ) :
        {
                .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 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 the ld *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 the ld's location counter to be incremented or reset and a "hole" left in the output section. "Holes" built into output sections in this manner take up physical space in the output file and are initialized using a fill character (either the default fill character (0x00) or a supplied fill character). See the definition of the "-f" option in "USING THE LINK EDITOR" and the discussion of filling holes in "Initialized Section Holes or .bss Sections" under "LINK EDITOR COMMAND LANGUAGE".

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:

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

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

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

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


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

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

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

Creating and Defining Symbols at Link-Edit Time

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

a. Use of "." to adjust ld's location counter during allocation

b. Use of "." to assign an allocation-dependent value to a symbol

c. Assigning an allocation-independent value to a symbol.

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

Case b) 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. The *ld* issues warning messages for each defined symbol that is being redefined within an ifile. However, assignments of absolute values to new symbols are safe because there are no references or initialized data associated with the symbol.

Allocating a Section Into Named Memory

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

For example,

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

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

This directs *ld* to place "outsec1" anywhere within the memory area named "mem1" (i.e., somewhere within the address range 0x0-0xFFFF or 0x120000-0x123FF). 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 "LINK EDITOR COMMAND LANGUAGE"), the *ld* 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. The *ld* will use the default fill value to initialize all *.bss* sections in the output section.

Consider the following *ld* 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 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 the *ld* 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 the ld 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 typically consisting of the standard three sections: *.text, .data,* and *.bss.* Archive libraries are created through the use of the UNIX system "ar" command from object files generated by running the **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 the *ld* prior to the actual scanning of the library.

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

a. Specific members of a library cannot be referenced explicitly in an ifile.

b. The default rules for the placement of members and sections cannot be overridden when they apply to archive library members.

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

The ordering of archive libraries is important since for a member to be extracted from the library it must satisfy a reference *that is*

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

Consider the following example:

a. The input files file1.o and file2.o each contain a reference to the external function FCN.

b. Input file1.o contains a reference to symbol ABC.

c. Input file2.o contains a reference to symbol XYZ.

d. Library liba.a, member 0, contains a definition of XYZ.

e. Library libc.a, member 0, contains a definition of ABC.

f. Both libraries have a member 1 that defines FCN.

If the *ld* 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 (since the library liba.a is searched before file2.o is specified). If the *ld* 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* command were entered as

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

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

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

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

creates an undefined symbol called "rout1" in the ld'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 "trigger" to cause ld 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, . . . fn.o each contain the standard 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* may do allocation. For example,

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

Allocation Algorithm

An output section is formed either as a result of a SECTIONS directive or by combining input sections of the same name. An output section can have zero or more input sections comprising it. After the composition of an output section is determined, it must then be allocated into configured virtual memory. Ld 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:

a. Any output sections for which explicit bonding addresses were specified are allocated.

b. 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.

c. 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 the ld, normally *.text, .data, .bss.* Otherwise, output sections are allocated in the order they were defined or made known to the ld into the first available space they fit.

Incremental Link Editing

As previously mentioned, the output of the ld can be used as an input file to subsequent *ld* 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

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

Step 2:

```
ld -r  -o outfile2  ifile2

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

Step 3:

```
ld  -a  -m  -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 programs are recompiled.

To apply this technique, there are two simple rules

a. 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.

b. All allocation and memory directives, as well as any assignment statements, are included only in the final ld call.

DSECT, COPY, and NOLOAD 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 }

}
```

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

a. 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 (the "-m" option) generated by the ld.

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

c. 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 in the DSECT or as a DSECT).

d. 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.

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.

Output File Blocking

The BLOCK option (applied to any output section or GROUP directive) is used to direct *ld* 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* 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,..., etc. in the file).

Nonrelocatable Input Files

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

When the *ld* detects an input file (that does not have relocation or symbol table information), a warning message is given. Such information can be removed by the *ld* (see the "-a" and "-s" options in the part USING THE LINK EDITOR) or by the *strip*(**1**) program. However, *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.

a. Each input file must have no unresolved external references.

b. Each input file must be bound to the exact same virtual address as it was bound to in the *ld* run that created it.

Note that 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 the ld.

ERROR MESSAGES

Corrupt Input Files

The following error messages indicate that the input file is corrupt, nonexistent, or unreadable. The user should check that the file is in the correct directory with the correct permissions. If the object file is corrupt, try recompiling or reassembling it.

- Can't open *name*
- Can't read archive header from archive *name*
- Can't read file header of archive *name*
- Can't read 1st word of file *name*
- Can't seek to the beginning of file *name*
- Fail to read file header of *name*
- Fail to read lnno of section *sect* of file *name*
- Fail to read magic number of file *name*

- Fail to read section headers of file *name*
- Fail to read section headers of library *name* member *number*
- Fail to read symbol table of file *name*
- Fail to read symbol table when searching libraries
- Fail to read the aux entry of file *name*
- Fail to read the field to be relocated
- Fail to seek to symbol table of file *name*
- Fail to seek to symbol table when searching libraries
- Fail to seek to the end of library *name* member *number*
- Fail to skip aux entries when searching libraries
- Fail to skip the mem of struct of *name*
- Illegal relocation type
- No reloc entry found for symbol
- Reloc entries out of order in section *sect* of file *name*
- Seek to *name* section *sect* failed
- Seek to *name* section *sect* lnno failed
- Seek to *name* section *sect* reloc entries failed
- Seek to relocation entries for section *sect* in file *name* failed.

Errors During Output

These errors occur because the *ld* cannot write to the output file. This usually indicates that the file system is out of space.

- Cannot complete output file *name*. Write error.
- Fail to copy the rest of section *num* of file *name*
- Fail to copy the bytes that need no reloc of section *num* of file
- *name* I/O error on output file *name*.

Internal Errors

These messages indicate that something is wrong with the *ld* internally. There is probably nothing the user can do except get help.

- Attempt to free nonallocated memory
- Attempt to reinitialize the SDP aux space
- Attempt to reinitialize the SDP slot space
- Default allocation did not put *.data* and *.bss* into the same region
- Failed to close SDP symbol space
- Failure dumping an AIDFNxxx data structure
- Failure in closing SDP aux space
- Failure to initialize the SDP aux space
- Failure to initialize the SDP slot space
- Internal error: audit_groups, address mismatch
- Internal error: audit_group, finds a node failure
- Internal error: fail to seek to the member of *name*
- Internal error: in allocate lists, list confusion (*num num*)
- Internal error: invalid aux table id

- Internal error: invalid symbol table id
- Internal error: negative aux table *ld*
- Internal error: negative symbol table id
- Internal error: no symtab entry for DOT
- Internal error: split_scns, size of *sect* exceeds its new displacement.

Allocation Errors

These error messages appear during the allocation phase of the link edit. They generally appear if a section or group does not fit at a certain address or if the given MEMORY or SECTION directives in some way conflict. If you are using an ifile, check that MEMORY and SECTION directives allow enough room for the sections to ensure that nothing overlaps and that nothing is being placed in unconfigured memory. For more information, see "LINK EDITOR COMMAND LANGUAGE" and "NOTES AND SPECIAL CONSIDERATIONS".

- Bond address *address* for *sect* is not in configured memory
- Bond address *address* for *sect* overlays previously allocated section *sect* at *address*
- Can't allocate output section *sect*, of size *num*
- Can't allocate section *sect* into owner *mem*
- Default allocation failed: *name* is too large
- GROUP containing section *sect* is too big
- Memory types *name1* and *name2* overlap
- Output section *sect* not allocated into a region
- *Sect* at *address* overlays previously allocated section *sect* at *address*

- *Sect,* bonded at *address,* won't fit into configured memory
- *Sect* enters unconfigured memory at *address*
- Section *sect* in file *name* is too big.

Misuse of Link Editor Directives

These errors arise from the misuse of an input directive. Please review the appropriate section in the manual.

- Adding *name(sect)* to multiple output sections.

The input section is mentioned twice in the SECTION directive.

- Bad attribute value in MEMORY directive: *c.*

An attribute must be one of "R", "W", "X", or "I".

- Bad flag value in SECTIONS directive, *option.*

Only the "-l" option is allowed inside of a SECTIONS directive

- Bad fill value.

The fill value must be a 2-byte constant.

- Bonding excludes alignment.

The section will be bound at the given address regardless of the alignment of that address.

- Cannot align a section within a group
- Cannot bond a section within a group

- Cannot specify an owner for sections within a group.

The entire group is treated as one unit, so the group may be aligned or bound to an address, but the sections making up the group may not be handled individually.

- DSECT *sect* can't be given an owner
- DSECT *sect* can't be linked to an attribute.

Since dummy sections do not participate in the memory allocation, it is meaningless for a dummy section to be given an owner or an attribute.

- Region commands not allowed.

The UNIX system link editor does not accept the REGION commands.

- Section *sect* not built.

The most likely cause of this is a syntax error in the SECTIONS directive.

- Semicolon required after expression
- Statement ignored.

Caused by a syntax error in an expression.

- Usage of unimplemented syntax.

The UNIX system *ld* does not accept all possible ld commands.

Misuse of Expressions

These errors arise from the misuse of an input expression. Please review the appropriate section in the manual.

- Absolute symbol *name* being redefined.

An absolute symbol may not be redefined.

- ALIGN illegal in this context.

Alignment of a symbol may only be done within a SECTIONS directive.

- Attempt to decrement DOT
- Illegal assignment of physical address to DOT.
- Illegal operator in expression
- Misuse of DOT symbol in assignment instruction.

The DOT symbol (".") cannot be used in assignment statements that are outside SECTIONS directives.

- Symbol *name* is undefined.

All symbols referenced in an assignment statement must be defined.

- Symbol *name* from file *name* being redefined.

A defined symbol may not be redefined in an assignment statement.

- Undefined symbol in expression.

Misuse of Options

These errors arise from the misuse of options. Please review the appropriate section of the manual.

- Both -r and -s flags are set; -s flag turned off.

Further relocation requires a symbol table.

- Can't find library lib*x*.a
- -L path too long (*string*)
- -o file name too large (>128 char), truncated to (*string*)
- Too many -L options, seven allowed.

Some options require white space before the argument, some do not; see " USING THE LINK EDITOR". Including extra white space or not including the required white space is the most likely cause of the following messages.

- *option* flag does not specify a number
- *option* is an invalid flag
- -e flag does not specify a legal symbol name *name*
- -f flag does not specify a 2-byte number
- No directory given with -L
- -o flag does not specify a valid file name: *string*
- the -l flag (specifying a default library) is not supported
- -u flag does not specify a legal symbol name: *name.*

Space Restraints

The following error messages may occur if the *ld* attempts to allocate more space than is available. The user should attempt to decrease the amount of space used by the ld. This may be accomplished by making the ifile less complicated or by using the "-r" option to create intermediate files.

- Fail to allocate *num* bytes for slotvec table
- Internal error: aux table overflow
- Internal error: symbol table overflow
- Memory allocation failure on *num*-byte 'calloc' call
- Memory allocation failure on realloc call
- Run is too large and complex.

Miscellaneous Errors

These errors occur for many reasons. Refer to the error message for an indication of where to look in the manual.

- Archive symbol table is empty in archive *name*, execute 'ar ts *name*' to restore archive symbol table.

On systems with a random access archive capability, the link editor requires that all archives have a symbol table. This symbol table may have been removed by strip.

- Cannot create output file *name*.

The user may not have write permission in the directory where the output file is to be written.

- File **name** has no relocation information.

See " NOTES AND SPECIAL CONSIDERATIONS" .

- File *name* is of unknown type, magic number = *num*
- Ifile nesting limit exceeded with file *name.*

Ifiles may be nested 16 deep.

- Library *name,* member has no relocation information.
- Line nbr entry (*num num*) found for nonrelocatable symbol:

Section *sect,* file *name*

This is generally caused by an interaction of *yacc*(**1**) and *cc*(**1**). Re-*yacc* the offending file with the " -l" option of *yacc.*

See the part " NOTES AND SPECIAL CONSIDERATIONS" .

- Multiply defined symbol *sym,* in *name* has more than one size.

A multiply defined symbol may not have been defined in the same manner in all files.

- *name*(*sect*) not found.

An input section specified in a SECTIONS directive was not found in the input file.

- Section *sect* starts on an odd byte boundary!

This will happen only if the user specifically binds a section at an odd boundary.

- Sections *.text, .data,* or *.bss* not found. Optional header may be useless.

The UNIX system a.out header uses values found in the *.text*, *.data*, and *.bss* section headers.

- Undefined symbol *sym* first referenced in file *name* .

Unless the -r option is used, the *ld* requires that all referenced symbols are defined.

- Unexpected EOF (End Of File).

Syntax error in the ifile.

SYNTAX DIAGRAM FOR INPUT DIRECTIVES

A syntax diagram for input directives is found in Figure 7-2.

directives	->	expanded directives
<file>	->	{ <cmd> }
<cmd>	->	<memory>
	->	<sections>
	->	<assignment>
	->	<filename>
	->	<flags>
<memory>	->	**MEMORY** { <memory_spec> { [,] <memory_spec> } }
<memory spec>	->	<name> [<attributes>] : <origin_spec> [,] <length_spec>
<attributes>	->	({ **R** ! **W** ! **X** ! **I** })
<origin_spec>	->	<origin> = <long>
<lenth_spec>	->	<length> = <long>
<origin>	->	**ORIGIN** ! **o** ! **org** ! **origin**
<length>	->	**LENGTH** ! **l** ! **len** ! **length**
<sections>	->	**SECTIONS** { { <sec_or_group> } }
<sec_or_group>	->	<section> ! <group> ! <library>
<group>	->	**GROUP** <group_options> : { <section_list> } [<mem_spec>]
<section_list>	->	<section> { [,] <section> }

Figure 7-2. Syntax Diagram for Input Directives (Sheet 1 of 4)

directives	->	expanded directives
<section>	->	<name> <sec_options> : { <statement_list> } [<fill>] [<mem_spec>]
<group_options>	->	[<addr>] | <align_option>]
<sec_options>	->	[<addr>] [<align_option>] [<block_option>] [<type_option>]
<addr>	->	<long>
<align_option>	->	<align> (<long>)
<align>	->	**ALIGN | align**
<block_option>	->	<block> (<long>)
<block>	->	**BLOCK | block**
<type_option>	->	**(DSECT) | (NOLOAD) | (COPY)**
<fill>	->	= <long>
<mem_spec>	->	> <name>
	->	> <attributes>
<statement>	->	<file_name> [(<name_list>)] [<fill>]
	->	<library>
	->	<assignment>
<name_list>	->	<name> { [,] <name> }
<library>	->	-l<name>
<assignment>	->	<lside> <assign_op> <expr> <end>
<lside>	->	<name> | .
<assign_op>	->	= | += | -= | *= | /=
<end>	->	; | ,
<expr>	->	<expr> <binary_op> <expr>
	->	<term>
<binary_op>	->	* | / | %
	->	+ | -
	->	>> | <<

Figure 7-2. Syntax Diagram for Input Directives (Sheet 2 of 4)

directives	->	expanded directives
	->	== | != | > | < | <= | >=
	->	&
	->	|
	->	&&
	->	||
<term>	->	<long>
	->	<name>
	->	<align> (<term>)
	->	(<expr)
	->	<unary_op> <term>
<unary_op>	->	! | -
<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
	->	-F
	->	-L<pathname>
	->	-M
	->	-N
	->	-S
	->	-V
	->	-VS<wht_space><long>
	->	-a
	->	-x

Figure 7-2. Syntax Diagram for Input Directives (Sheet 3 of 4)

directives	->	expanded directives
<name>	->	Any valid symbol name
<long>	->	Any valid long integer constant
<wht_space>	->	Blanks, tabs, and newlines
<filename>	->	Any valid UNIX operating system filename. This may include a full or partial pathname.
<pathname>	->	Any valid UNIX operating system pathname (full or partial)

Figure 7-2. Syntax Diagram for Input Directives (Sheet 4 of 4)

Chapter 10

THE COMMON OBJECT FILE FORMAT

GENERAL

This chapter describes the Common Object File Format (COFF) used on several processors and operating systems, including the Western Electric 3B Computer family and the UNIX operating system. The COFF is simple enough to be easily incorporated into existing projects, yet flexible enough to meet the needs of most projects. The COFF is the output file produced on some UNIX systems by the assembler (*as*) and the link editor (*ld*). This format is also used by other operating systems; hence, the word *common* is both descriptive and widely recognized. Currently, this object file format is used for the Western Electric 3B Computers family, including the 3B 20D, the 3B 20S, and the 3B 5 Computers, and on the VAX*-11/780 and 11/750 UNIX operating systems. Some key features of COFF are

- Applications may add system-dependent information to the object file without causing access utilities to become obsolete.
- Space is provided for symbolic information used by debuggers and other applications
- Users may make some modifications in the object file construction at compile time.

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

* Trademark of Digital Equipment Corporation

- A table of section headers
- Data corresponding to the section header
- Relocation information
- Line numbers
- A symbol table
- A string table.

Figure 8-1 shows the overall structure.

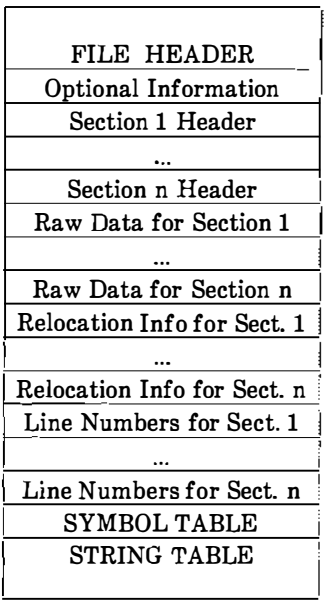


Figure 8-1. Object File Format

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 UNIX system link editor 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 compiler *(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 can be executed on the target machine.

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 default case, there are three sections named **.text**, **.data**, and **.bss**. Additional sections accommodate multiple text or data segments, shared data segments, or user-specified sections. However, the UNIX operating system loads only the *.text*, *.data*, and *.bss* into memory when the file is executed.

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 the 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 heading contains two address fields, a physical address, and a virtual address; but in all

versions of COFF on *UNIX* systems, the *physical address* is equivalent to the *virtual address.*

FILE HEADER

The file header contains the 20 bytes of information shown in Figure 8-2. The last 2 bytes are flags that are used by *ld* and object file utilities.

Bytes	Declaration	Name	Description
0-1	unsigned short	f_magic	Magic number, see Figure 8-3.
2-3	unsigned short	f_nscns	Number of section headers (equals the number of sections)
4-7	long int	f_timdat	Time and date stamp indicating when the file was created relative to the number of elapsed seconds since 00:00:00 GMT, January 1, 1970.

Figure 8-2. File Header Contents (Sheet 1 of 2)

Bytes	Declaration	Name	Description
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 8-4)

Figure 8-2. File Header Contents (Sheet 2 of 2)

The size of optional header information (**f_opthdr**) is used by all referencing programs that seek to the beginning of the section header table. This enables the same utility programs to work correctly on files targeted for different systems.

Magic Numbers

The magic number specifies the target machine on which the object file is executable. The currently defined magic numbers are in Figure 8-3.

Mnemonic	Magic Number	System
N3B MAGIC	0550	3B 20S Computers
FBOMAGIC	0560	WE*-32 (Forward Byte Ordering)
RBOMAGIC	0565	WE-32 (Reverse Byte Ordering)
VAXWRMAGIC	0570	VAX-11/750 and VAX-11/780 (writable text segments)
VAXROMAGIC	0575	VAX-11/750 and VAX-11780 (writable text segments)
U370WRMAGIC	0530	IBM 370 (writable text segments)
U370ROMAGIC	0535	IBM 370 (read-only sharable text segments)

* Trademark of AT&T Technologies

Figure 8-3. Magic Numbers

Flags

The last 2 bytes of the file header are flags that describe the type of the object file. The currently defined flags are given in Figure 8-4.

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 UNIX
F_UPDATE	00040	Not used by UNIX
F_SWABD	00100	Not used by UNIX
F_AR16WR	00200	File has the byte ordering used by the PDP*-11/70 processor.

* Trademark of Digital Equipment Corporation

Figure 8-4. File Header Flags (Sheet 1 of 2)

Mnemonic	Flag	Meaning
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 3B 20S computers (i.e., 32 bits per word, most significant byte first).
F_PATCH	02000	Not used by UNIX

Figure 8-4. File Header Flags (Sheet 2 of 2)

File Header Declaration

The C structure declaration for the file header is given in Figure 8-5. 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 data 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 */
};
```

Figure 8-5. File Header Declaration

OPTIONAL HEADER INFORMATION

The template for optional information varies among different systems that use the 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 **f_opthdr**.

Standard UNIX System a.out Header

By default, files produced by the link editor for a *UNIX* system always have a standard *UNIX* System **a.out** header in the optional header field. The UNIX system a.out header is 28 bytes. There is one exception; files produced for the 3B 20S Computers have an optional header of 36 bytes. The extra 8 bytes represent unused fields that are present for historical reasons. Therefore, the two formats contain functionally equivalent information. The fields of the optional header are described in Figure 8-6 and 8-7.

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	dum1	Unused dummy field
20-23	long int	dum2	Unused dummy field
24-27	long int	entry	Entry point
27-31	long int	text_start	Base address of text
32-35	long int	data_start	Base address of data

Figure 8-6. Optional Header Contents (3B 20S Computers Only)

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
24-37	long int	data_start	Base address of data

Figure 8-7. Optional Header Contents (VAX-11/780 Processor)

The magic number in the optional header supplies operating system dependent information about the object file. 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 UNIX operating system are given in Figure 8-8.

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.

Figure 8-8. UNIX Magic Numbers

Optional Header Declaration

The C language structure declaration currently used for the *UNIX* system **a.out** file header is given in Figure 8-9. 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 */
#if   u3b
      long    dum1;        /* unused dummy field */
      long    dum2;        /* unused dummy field */
#endif
      long    entry;       /* entry point */
      long    text_start;  /* base of text for this file */
      long    data_start   /* base of data for this file */
} AOUTHDR;
```

Figure 8-9. 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 8-10.

Bytes	Declaration	Name	Description
0-7	char	s_name	8-char 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 ptr to relocation entries
28-31	long int	s_lnnoptr	File ptr to line number entries
32-33	unsigned short	s_nreloc	Number of entries
34-35	unsigned short	s_nlnno	Number of line number entries
36-39	long int	s_flags	Flags (see Figure 8-11)

Figure 8-10. Section Header Contents

The size of a section is padded to a multiple of 4 bytes.

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

Flags

The lower 4 bits of the flag field indicate a section type. The flags are described in Figure 8-11.

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)

Figure 8-11. Section Header Flags (Sheet 1 of 2)

Mnemonic	Flag	Meaning
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

Figure 8-11. Section Header Flags (Sheet 2 of 2)

Section Header Declaration

The C structure declaration for the section headers is described in Figure 8-12. This declaration may be found in the header file *scuhdr.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 8-12. 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 zero.

SECTIONS

Figure 8-1 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 full word boundary in the file.

Files produced by the cc and the as always contain three sections, called **.text**, **.data**, and **.bss**. The **.text** section contains the instruction text (i.e., executable code), the **.data** section contains initialized data variables, and the **.bss** section contains uninitialized data variables.

The link editor "SECTIONS directives" (see Chapter 7) allows users to

- Describe how input sections are to be combined.
- Direct the placement of output sections.
- Rename output sections.

If no SECTIONS directives are given, each input section appears in an output section of the same name. For example, if a number of object files from the "cc" are linked together (each containing the three sections **.text**, **.data**, and **.bss**), the output object file contains three sections, **.text**, **.data**, and **.bss**.

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 8-13.

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

Figure 8-13. Relocation Section Contents

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

As the link editor reads each input section and performs relocation, the relocation entries are read. They direct how references found within the input section are treated.

The currently recognized relocation types are given in Figures 8-14 through 8-16.

Mnemonic	Flag	Meaning
R_ABS	0	Reference is absolute; no relocation is necessary. The entry will be ignored.
R_DIR24	04	Direct 24-bit reference to the symbol's virtual address.
R_REL24	05	A "PC-relative" 24-bit reference to the symbol's virtual address. Actual address is calculated by adding a constant to the PC value.

Figure 8-14. 3B 20S Computers Relocation Types

Mnemonic	Flag	Meaning
R_BS	0	Reference is absolute; no relocation is necessary. The entry will be ignored.
R_DIR32	06	Direct 32-bit reference to the symbol's virtual address
R_DIR32S	012	Direct 32-bit reference to the symbol's virtual address, with the 32-bit value stored in the reverse order in the object file.

Figure 8-15. 3B 5 Relocation Types

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 8-16. VAX Relocation Types

On the VAX processors, relocation of a symbol index of -1 indicates that the amount by which the section is being relocated is added to the relocatable address.

The *as* automatically generates relocation entries which are then used by the link editor. The link editor uses this information to resolve external references in the file.

Relocation Entry Declaration

The structure declaration for relocation entries is given in Figure 8-17. 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 8-17. Relocation Entry Declaration

LINE NUMBERS

When invoked with the -g option, UNIX system ccs (*cc*, *f77*) generates an entry in the object file for every C language 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 8-18.

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

Figure 8-18. 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 appear in increasing order of address.

Line Number Declaration

The structure declaration currently used for line number entries is given in Figure 8-19.

```
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 8-19. 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 8-20.

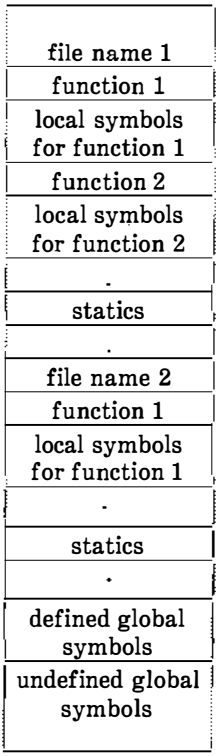


Figure 8-20. COFF Global Symbol Table

The word "statics" in Figure 8-20 means symbols defined in 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 the "cc", "as", and other tools. These symbols are given in Figure 8-21.

Symbol	Meaning
.file	file name
.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 enumeran

Figure 8-21. Special Symbols in the Symbol Table (Sheet 1 of 2)

Symbol	Meaning
.eos	end of members of structure, union, or enumeration
_etext,etext	next available address after the end of the output section .text
_edata,edata	next available address after the end of the output section .data
_end,end	next available address after the end of the output section .bss.

Figure 8-21. Special Symbols in the Symbol Table (Sheet 2 of 2)

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; and a *.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 " cc" 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 entry.

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 8-22.

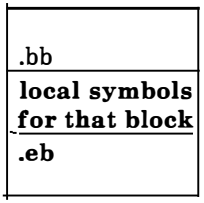


Figure 8-22. Special Symbols (.bb and .eb)

Because inner blocks can be nested by several levels, the **.bb-.eb** pairs and associated symbols may also be nested. See Figure 8-23.

```
{                           /* 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 8-23. Nested blocks

The symbol table would look like Figure 8-24.

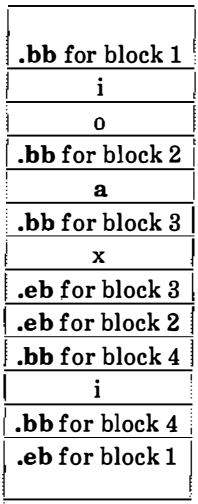


Figure 8-24. Example of the Symbol Table

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 8-25.

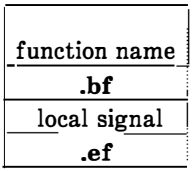


Figure 8-25. Symbols for Functions

If the return value of the function is a structure or union, a special symbol **.target** is put between the function name and the **.bf**. The sequence is shown in Figure 8-26.

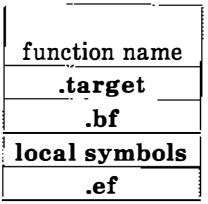


Figure 8-26. The Special Symbol .Target

The **cc** invents **.target** to store the function-return structure or union. The symbol **.target** is an automatic variable with "pointer" type. Its value field in the symbol is always 0.

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 the 18 bytes of information. The meaning of each of the fields in the symbol table entry is described in Figure 8-27.

It should be noted that indices for symbol table entries begin at zero and count upward. Each auxiliary entry also counts as one symbol.

Bytes	Declaration	Name	Description
0-7	(see text below)	_n	These eight bytes contain either the name of a pointer or the name of 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 8-27. Symbol Table Entry Format

Symbol Names

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

bytes serve to distinguish a symbol table entry with an offset from one with a name in the first 8 bytes as shown in Figure 8-28.

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 8-28. Name Field

Some special symbols are generated by the "cc" and link editor as discussed in "special symbols". The VAX "cc" prepends an underscore ('_') to all the user defined symbols it generates.

Storage Classes

The storage class field has one of the values described in Figure 8-29. These " defines" may be found in the header file *storclass.h*.

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

Figure 8-29. Storage Classes (Sheet 1 of 2)

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

Figure 8-29. Storage Classes (Sheet 2 of 2)

All of these storage classes except for C_ALIAS and C-HIDDEN are generated by the " cc" or " as". The compress utility, **cprs**, generates the C_ALIAS mnemonic. This utility removes duplicated structure, union, and enumeration definitions and puts ALIAS entries in their places. The storage class C-HIDDEN is not used by any UNIX system tools. [See **cprs**(1) in the Runtime System manual.]

Some of these storage classes are used only internally by the " cc" and the " as". 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 8-30.

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 8-30. Storage Class by Special Symbols

Also some storage classes are used only for certain special symbols. They are summarized in Figure 8-31.

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

Figure 8-31. 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 8-32.

Storage Class	Meaning
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 8-32. 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 1-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 8-33.

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 was defined

Figure 8-33. 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. The **.text**, **.data**, and **.bss** symbols default to section numbers 1, 2, and 3, respectively.

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, the link editor combines *all* the input symbols 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 8-34.

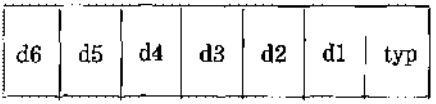
Storage Class	Section Number
C_AUTO	N_ABS
C_EXT	N_ABS, N_UNDEF, N_SCNUM
C_STAT	N_SCNUM
C_REG	N_ABS
C_LABEL	N_UNDEF, N_SCNUM
C_MOS	N_ABS
C_ARG	N_ABS
C_STRTAG	N_DEBUG
C_MOU	N_ABS
C_UNTAG	N_DEBUG
C_TPDEF	N_DEBUG
C_ENTAG	N_DEBUG
C_MOE	N_ABS
C_REGPARM	N_ABS
C_FIELD	N_ABS
C_BLOCK	N_SCNUM
C_FCN	N_SCNUM
C_EOS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

Figure 8-34. 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 "cc". The VAX "cc" generates this information 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



Bits 0 through 3, called "typ", indicate one of the fundamental types given in Figure 8-35.

Mnemonic	Value	Type
T_NULL	0	type not assigned
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 8-35. 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 8-36.

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

Figure 8-36. 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 "function that returns a pointer to a character."

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

Here *tabptr* is a 3-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 "3-dimensional array of pointers to short integers."

Type Entries and Storage Classes

Figure 8-37 shows the type entries that are legal for each storage class.

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

Figure 8-37. Type Entries by Storage Class (Sheet 1 of 2)

Storage Class	----------"d" entry----------			"typ" entry
	Function?	Array?	Pointer?	Basic Type
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 8-37. Type Entries by Storage Class (Sheet 2 of 2)

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

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

Structure for Symbol Table Entries

The C language structure declaration for the symbol table entry is given in Figure 8-38. 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;
       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 8-38. Symbol Table Entry Declaration

Auxiliary Table Entries

Currently, there is at most one auxiliary entry per symbol. The auxiliary table entry 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 8-39.

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

Notes:
1. Any except T_MOE.
2. C_AUTO, C_STAT, C_MOS, C_MOU, C_TPDEF.

Figure 8-39. Auxiliary Symbol Table Entries

In Figure 8-39, "tagname" means any symbol name including the special symbol .x**fake**, and "fcname" and "arrname" represent any symbol name.

Any symbol that satisfies more than one condition in Figure 8-39 should have a union format in its auxiliary entry. Symbols that do not satisfy any of the above conditions should **NOT** have any auxiliary entry.

File Names

Each of the auxiliary table entries for a file name contains a 14-character file name in bytes 0 through 13. The remaining bytes are 0, regardless of the size of the entry.

Sections

The auxiliary table entries for sections have the format as shown in Figure 8-40.

Bytes	Declaration	Name	Description
0-3	long int	x_scnlen	section length
4-6	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 8-40. Format for Auxiliary Table Entries

Tag Names

The auxiliary table entries for tag names have the format shown in Figure 8-41.

Bytes	Declaration	Name	Description
0-5	-	-	unused (filled with zeros)
6-7	unsigned short	x_size	size of strucrt, 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 8-41. Tag Names Table Entries

End of Structures

The auxiliary table entries for the end of structures have the format shown in Figure 8-42.

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 struct, union, or enumeration
8-17	-	-	unused (filled with zeroes)

Figure 8-42. Table Entries for End of Structures

Functions

The auxiliary table entries for functions have the format shown in Figure 8-43:

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

Figure 8-43. Table Entries for Functions

Arrays

The auxiliary table entries for arrays have the format shown in Figure 8-44:

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 8-44. 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 8-45:

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

Figure 8-45. 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 8-46:

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 8-46. Format for Beginning of Block and Function

Names Related to Structures, Unions, and Enumerations

The auxiliary table entries for structure, union, and enumerations symbols have the format shown in Figure 8-47:

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 numeration
8-17	-	-	unused (filled with zeroes)

Figure 8-47. Entries for Structures, Unions, and Numerations

Names 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.

Auxiliary Entry Declaration

The C language structure declaration for an auxiliary symbol table entry is given in Figure 8-48. 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;
        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 8-48. Auxiliary Symbol Table Entry

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 four.

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 8-49:

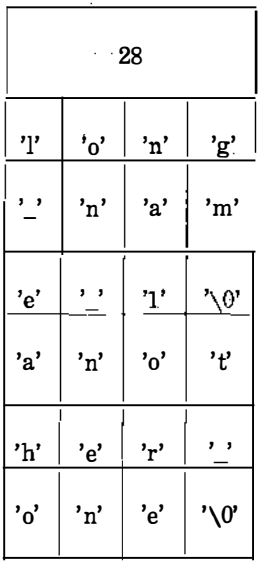


Figure 8-49. String Table

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

ACCESS ROUTINES

Supplied with every standard *UNIX* system release is 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. In this way, you can concern yourself with the section you are interested in without knowing all the object file details.

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 3X of the Software Development System manual. A summary of what is available can be found under LDFCN(4).

Chapter 11

SYMBOLIC DEBUGGING PROGRAM—"sdb"

GENERAL

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

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 provided formatted printout of structured data.

C-11

USAGE

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

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

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

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

In the above example, **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 which 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 name of the directory containing the source of the program being debugged. The **sdb** program currently requires all source to reside in a single directory. The default is the working directory. In the example, the second and third arguments defaulted to the correct values, so only the first was specified.

It is possible that the error occurred in a function which was not compiled with the **-g** option. In this case, **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*. The **sdb** program will print an error message if *main* was not compiled with the **-g** option, but debugging can continue for those routines compiled with the **-g** option. Figure 7-1 shows a typical example of **sdb** usage.

Printing a Stack Trace

It is often useful to obtain a listing of the function calls which 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=0x7fffff54,envp=0x7fffff5c)[prgm.c:15]
```

This indicates that the error occurred 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 the **shell** 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*. F77 users can specify a common block variable in the same manner.

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 lengths 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 numeric length specifier may be used for the **s** or **a** commands. These commands normally print characters until either a null is reached or 128 characters are printed. The number specifies 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.

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

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

The only restriction is that array subscripts must be numbers. Depending on your machine, accessing arrays may be limited to 1-dimensional arrays. Note that as a special case:

```
*psym->/d
```

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

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

```
*1024/
```

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

```
*02000/
```

and

```
*0x400/
```

It is possible to mix numbers and variables so that

```
*1000.x/
```

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

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

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

The address of a variable is printed with the =, so

```
*i=
```

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

```
*./
```

which redisplays the last variable typed.

SOURCE FILE DISPLAY AND MANIPULATION

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

Displaying the Source File

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

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

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

Changing the Current Source File or Function

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

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

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

Changing the Current Line in the Source File

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

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

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

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

The + and - commands may be used to move the current line forwards or backwards by a specified number of lines. Typing a

new-line advances the current line by one, and typing a number causes that line to become the current line in the file. These commands may be combined with the display commands so that

```
*+15z
```

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

A CONTROLLED ENVIRONMENT FOR PROGRAM TESTING

One very useful feature of **sdb** is breakpoint debugging. After entering **sdb**, certain lines in the source program may be specified to be *breakpoints*. The program is then started with a **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. The **sdb** program 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 which 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 which contains executable code. 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 commands

```
*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 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 **sdb**.

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 which continues but passes the signal which stopped the

program back to the program. This is useful for testing user-written signal handlers. Execution may be continued at a specified line with the **g** command. For example:

```
*17 g
```

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

The **s** command is used to run the program for a single line. 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 which stopped the program. Its uses are similar to the **s** command. There is also an **I** command which causes the program to execute one machine level instruction at a time, but also passes the signal which 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 function which prints structured data in a nice way. 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 values of variables which 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 which formats data directly from a core dump which is being debugged.

MACHINE LANGUAGE DEBUGGING

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

Displaying Machine Language Statements

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

```
*main:25?
```

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

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

```
*0x1024:?
```

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

```
*0x1024?
```

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

```
*0x1024:b
```

sets a breakpoint at address *0x1024*.

Manipulating Registers

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

```
*r3%
```

displays the value of register *r3*.

OTHER COMMANDS

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

The ! command is identical to that in **ed**(1) and is used to have the **shell** execute a command.

It is possible to change the values of variables when the program is stopped at a breakpoint. This is done with the command

```
*variable!value
```

which sets the variable 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.

```
$ cat testdiv2.c
main(argc, argv, envp)
char **argv, **envp; {
   int i;
   i = div2(-1);
   printf(" -1/2  =  % d\n", i);
}
div2(i) {
   int j;
   j = i>>1;
   return(j);
}
$ cc -g  testdiv2.c
$ a.out
-1/2 = -1
$ sdb
No core image      # Warning message from sdb
*/^div2            # Search for function " div2"
7: div2(i) {       # It starts on line 7
*z                 # Print the next few lines
7: div2(i) {
8:    int j;
9:    j = i>>1;
10:   return(j);
11: }
*div2:b      # Place breakpoint at beginning of " div2"
div2:9 b     # Sdb echoes proc name and line number
*r           # Run the function
a.out        # Sdb echoes command line executed
Breakpoint at # Executions stops just before line 9
div2:9:    j = i>>1;
*t         # Print trace of subroutine calls
div2(i=-1)    [testdiv2.c:9]
main(argc=1,argv=0x7fffff50,envp=0x7fffff58)[testdiv2.c:4]
*i/        # Print i
-1
*s         # Single step
div2:10: return(j);   # Execution stops before line 10
*j/        # Print j
-1
*9d        # Delete the breakpoint
*div2(1)/ # Try running " div2"  with different arguments
0
```

```
*div2(-2)/
-1
*div2(-3)/
-2
*q
$
```

Figure 7-1. EXAMPLE OF sdb USAGE

Chapter 12

FORTRAN 77

This chapter describes the compiler and run-time system for Fortran 77 as implemented on the UNIX system. This chapter also describes the interfaces between procedures and the file formats assumed by the I/O system.

USAGE

The command to run the compiler is

f77 options file

The **f77**(1) command is a general purpose command for compiling and loading Fortran and Fortran-related files into an executable module. EFL (compiler) and Ratfor (preprocessor) source files will be translated into Fortran before being presented to the Fortran compiler. The **f77** command invokes the C compiler to translate C source files and invokes the assembler to translate assembler source files. Object files will be link edited. [The **f77**(1) and cc(1) commands have slightly different link editing sequences. Fortran programs need two extra libraries (libI77.a) and an additional startup routine.] The following file name suffixes are understood:

.f	Fortran source file
.e	EFL source file
.r	Ratfor source file
.c	C language source file
.s	Assembler source file
.o	Object file.

C-12

LANGUAGE EXTENSIONS

Fortran 77 includes almost all of Fortran 66 as a subset. The most important additions are a character string data type, file-oriented input/output statements, and random access I/O. Also, the language has been cleaned up considerably.

In addition to implementing the language specified in the Fortran 77 American National Standard, this compiler implements a few extensions. Most are useful additions to the language. The remainder are extensions to make it easier to communicate with C language procedures or to permit compilation of old (1966 Standard Fortran) programs.

Double Complex Data Type

The data type double complex is added. Each datum is represented by a pair of double-precision real variables. A double complex version of every complex built-in function is provided.

Internal Files

The Fortran 77 American National Standard introduces *internal files* (memory arrays) but restricts their use to formatted sequential I/O statements. This I/O system also permits internal files to be used in direct and unformatted reads and writes.

Implicit Undefined Statement

Fortran has a rule that the type of a variable that does not appear in a type statement is integer if its first letter is *i*, *j*, *k*, *l*, *m* or *n*. Otherwise, it is real. Fortran 77 has an implicit statement for overriding this rule. An additional type statement, *undefined*, is permitted. The statement

```
implicit undefined(a-z)
```

turns off the automatic data typing mechanism, and the compiler will issue a diagnostic for each variable that is used but does not appear in a type statement. Specifying the **-u** compiler option is equivalent to beginning each procedure with this statement.

Recursion

Procedures may call themselves directly or through a chain of other procedures.

Automatic Storage

Two new keywords recognized are **static** and **automatic.** These keywords may appear as "types" in type statements and in **implicit** statements. Local variables are static by default; there is exactly one copy of the datum, and its value is retained between calls. There is one copy of each variable declared **automatic** for each invocation of the procedure. Automatic variables may not appear in **equivalence, data,** or **save** statements.

Variable Length Input Lines

The Fortran 77 American National Standard expects input to the compiler to be in a 72-column format: except in comment lines, the first five characters are the statement number, the next is the continuation character, and the next 66 are the body of the line. (If there are fewer than 72 characters on a line, the compiler pads it with blanks; characters after the first 72 are ignored.) In order to make it easier to type Fortran programs, this compiler also accepts input in variable length lines. An ampersand (**&**) in the first position of a line indicates a continuation line; the remaining characters form the body of the line. A tab character in one of the first six positions of a line signals the end of the statement number and continuation part of the line; the remaining characters form the body of the line. A tab elsewhere on the line is treated as another kind of blank by the compiler.

In the Fortran 77 Standard, there are only 26 letters

— Fortran is a one-case language. Consistent with ordinary system usage, the new compiler expects lowercase input. By default, the compiler converts all uppercase characters to lowercase except those inside character constants. However, if the **-U** compiler option is specified, uppercase letters are not transformed. In this mode, it is possible to specify external names with uppercase letters in them and to have distinct variables differing only in case. Regardless of the setting of the option, keywords will only be recognized in lowercase.

Include Statement

The statement

```
include " stuff"
```

is replaced by the contents of the file *stuff*. Includes may be nested to a reasonable depth, currently ten.

Binary Initialization Constants

A **logical, real,** or **integer** variable may be initialized in a data statement by a binary constant, denoted by a letter followed by a quoted string. If the letter is *b*, the string is binary, and only zeroes and ones are permitted. If the letter is *o*, the string is octal with digits *zero* through *seven*. If the letter is z or x, the string is hexadecimal with digits *zero* through *nine*, *a* through *f*. Thus, the statements

```
integer a(3)
data a/b'1010',o'12',z'a'/
```

initialize all three elements of a to ten.

Character Strings

For compatibility with C language usage, the following backslash escapes are recognized:

\n	New-line
\t	Tab
\b	Backspace
\f	Form feed

\0	Null
\'	Apostrophe (does not terminate a string)
\"	Quotationmark (does not terminate a string)
\\	\
\x	Where **x** is any other character.

Fortran 77 only has one quoting character — the apostrophe ('). This compiler and I/O system recognize both the apostrophe and the double quote ("). If a string begins with one variety of quote mark, the other may be embedded within it without using the repeated quote or backslash escapes.

Every unequivalenced scalar local character variable and every character string constant is aligned on an integer word boundary. Each character string constant appearing outside a data statement is followed by a null character to ease communication with C language routines.

Hollerith

Fortran 77 does not have the old Hollerith **(nh)** notation though the new Standard recommends implementing the old Hollerith feature in order to improve compatibility with old programs. In this compiler, Hollerith data may be used in place of character string constants and may also be used to initialize non character variables in data statements.

Equivalence Statements

This compiler permits single subscripts in **equivalence** statements under the interpretation that all missing subscripts are equal to 1. A warning message is printed for each such incomplete subscript.

One-Trip DO Loops

The Fortran 77 American National Standard requires that the range of a **do** loop not be performed if the initial value is already past the limit value, as in

```
do 10 i = 2, 1
```

The 1966 Standard stated that the effect of such a statement was undefined, but it was common practice that the range of a **do** loop would be performed at least once. In order to accommodate old programs though they were in violation of the 1966 Standard, the **-onetrip** compiler option causes nonstandard loops to be generated.

Commas in Formatted Input

The I/O system attempts to be more lenient than the Fortran 77 American National Standard when it seems worthwhile. When doing a formatted read of non-character variables, commas may be used as value separators in the input record overriding the field lengths given in the format statement. Thus, the format

```
(i10, f20.10, i4)
```

will read the record

```
-345,.05e-3,12
```

correctly.

Short Integers

On machines that support half word integers, the compiler accepts declarations of type **integer*2.** (Ordinary integers follow the Fortran rules about occupying the same space as a REAL variable; they are assumed to be of C language type **long int;** half word integers are of C language type **short int.**) An expression involving only objects of type **integer*2** is of that type. Generic functions return short or long integers depending on the actual types of their arguments. If a procedure is compiled using the **-I2** flag, all small

integer constants will be of type **integer*2.** If the precision of an integer-valued intrinsic function is not determined by the generic function rules, one will be chosen that returns the prevailing length (**integer*2** when the **-I2** command flag is in effect). When the **-I2** option is in effect, all quantities of type **logical** will be short. Note that these short integer and logical quantities do not obey the standard rules for storage association.

Additional Intrinsic Functions

This compiler supports all of the intrinsic functions specified in the Fortran 77 Standard. In addition, there are functions for performing bitwise Boolean operations (**or, and, xor,** and **not**) and for accessing the command arguments (**getarg** and **iargc**).

The following lists the Fortran intrinsic function library plus some additional functions. These functions are automatically available to the Fortran programmer and require no special invocation of the compiler. The asterisk (*) beside some of the commands indicate they are not part of standard F77. In parenthesis beside each function description listed below is the location for the command in the *UNIX System Programmer Reference Manual*. These functions are as follows:

```
abort*  Terminate program (ABORT(3F))
abs     Absolute value (MAX(3F)
acos    Arccosine (ACOS(3F))
aimag   Imaginary part of complex argument (AIMAG(3F))
aint    Integer part (AINT(3F))
alog    Natural logarithm (LOG(3F))
alog10  Common logarithm (ALOG10(3F))
amax0   Maximum value (MAX(3F))
amax1   Maximum value (MAX(3F))
amin0   Minimum value (MIN(3F))
amin1   Minimum value (MIN(3F))
amod    Remaindering (MOD(3F))
and*    Bitwise boolean (BOOL(3F))
anint   Nearest integer (ROUND(3F))
asin    Arcsine (ASIN(3F))
atan    Arctangent (ATAN(3F))
atan2   Arctangent (ATAN2(3F))
```

cabs	Complex absolute value (ABS(3F))
ccos	Complex cosine (COS(3F))
cexp	Complex exponential (EXP(3F))
char	Explicit type conversion (FTYPE(3F))
clog	Complex natural logarithm (LOG(3F))
cmplx	Explicit type conversion (FTYPE(3F))
conjg	Complex conjugate (CONJG(3F))
cos	Cosine (COS(3F))
cosh	Hyperbolic cosine (COSH(3F))
csin	Complex sine (SIN(3F))
csqrt	Complex square root (SQRT(3F))
dabs	Absolute value (ABS(3F))
dacos	Arccosine (ACOS(3F))
dasin	Arcsine (ASIN(3F))
datan	Arctangent (ATAN(3F))
datan2	Double precision arctangent (ATAN2(3F))
dble	Explicit type conversion (FTYPE(3F))
dcmplx*	Explicit type conversion (FTYPE(3F))
dconjg*	Complex conjugate (CONJG(3F))
dcos	Cosine (DCOS(3F))
dcosh	Hyperbolic cosine (COSH(3F))
ddim	Positive difference (DIM(3F))
dexp	Exponential (EXP(3F))
dim	Positive difference (DIM(3F))
dimag*	Imaginary part of complex argument ((AIMAG(3F))
dint	Integer part (AINT(3F))
dlog	Natural logarithm (LOG(3F))
dlog10	Common logarithm (LOG10(3F))
dmax1	Maximum value (MAX(3F))
dmin1	Minimum value (MIN(3F))
dmod	Remaindering (DMOD(3F))
dnint	Nearest integer (ROUND(3F))
dprod	Double precision product (DPROD(3F))
dsign	Transfer of sign (SIGN(3F))
dsin	Sine (SIN(3F))
dsinh	Hyperbolic sine (SINH(3F))
dsqrt	Square root (SQRT(3F))
dtan	Tangent (TAN(3F))
dtanh	Hyperbolic tangent (TANH(3F))
exp	Exponential (EXP(3F))
float	Explicit type conversion (FTYPE(3F))
getarg*	Return command-line argument (GETARG(3F))
getenv*	Return environment variable (GETENV(3F))

iabs	Absolute value (ABS(3F))
iargc	Return number of arguments (IARGC(3F))
ichar	Explicit type conversion (FTYPE(3F))
idim	Positive difference (DIM(3F))
idint	Explicit type conversion (FTYPE(3F))
idnint	Nearest integer (ROUND(3F))
ifix	Explicit type conversion (FTYPE(3F))
index	Return location of substring (INDEX(3F))
int	Explicit type conversion (FTYPE(3F))
irand*	Random number generator
isign	Transfer of sign (SIGN(3F))
len	Return location of string (LEN(3F))
lge	String comparison (STRCMP(3F))
lgt	String comparison (STRCMP(3F))
lle	String comparison (STRCMP(3F))
llt	String comparison (STRCMP(3F))
log	Natural logarithm (LOG(3F))
log10	Common logarithm (LOG10(3F))
lshift*	Bitwise boolean (BOOL(3F))
max	Maximum value (MAX(3F))
max0	Maximum value (MAX(3F))
max1	Maximum value (MAX(3F))
mclock*	Return Fortran time accounting (MCLOCK(3F))
min	Minimum value (MIN(3F))
min0	Minimum value (MIN(3F))
min1	Minimum value (MIN(3F))
mod	Remaindering (MOD(3F))
nint	Nearest integer (BOOL(3F))
not*	Bitwise boolean (BOOL(3F))
or*	Bitwise boolean (BOOL(3F))
rand*	Random number generator (RAND(3F))
real	Explicit type conversion (FTYPE(3F))
rshift*	Bitwise boolean (BOOL(3F))
sign	Transfer of sign (SIGN(3F))
signal*	Specify action on receipt of system signal (SIGNAL(3F))
sin	Sine (SINE(3F))
sinh	Hyperbolic sine (SINH(3F))
sngl	Explicit type conversion (FTYPE(3F))
sqrt	Square root (SQRT(3F))
srand*	Random number generator (RAND(3F))
system*	Issue a shell command (SYSTEM(3F))

```
tan     Tangent (TAN(3F))
tanh    Hyperbolic tangent (TANH(3F))
xor*    Bitwise boolean (BOOL(3F))
zabs*   Complex absolute value (ABS(3F)).
```

For more information on the Fortran intrinsic function commands, see the *UNIX System Programmer Reference Manual.*

VIOLATIONS OF THE STANDARD

The following paragraphs describe only three known ways in which the UNIX system implementation of Fortran 77 violates the new American National Standard.

Double Precision Alignment

The Fortran 77 American National Standard permits **common** or **equivalence** statements to force a double precision quantity onto an odd word boundary, as in the following example:

```
real a(4)
double precision b,c
equivalence (a(1),b), (a(4),c)
```

Some machines require that double precision quantities be on double word boundaries; other machines run inefficiently if this alignment rule is not observed. It is possible to tell which equivalenced and common variables suffer from a forced odd alignment, but every double-precision argument would have to be assumed on a bad boundary. To load such a quantity on some machines, it would be necessary to use two separate operations. The first operation would be to move the upper and lower halves into the halves of an aligned temporary. The second would be to load that double-precision temporary. The reverse would be needed to store a result. All double-precision real and complex quantities are required to fall on even word boundaries on machines with corresponding hardware requirements and to issue a diagnostic if the source code demands a violation of the rule.

Dummy Procedure Arguments

If any argument of a procedure is of type character, all dummy procedure arguments of that procedure must be declared in an **external** statement. This requirement arises as a subtle corollary of the way we represent character string arguments. A warning is printed if a dummy procedure is not declared **external.** Code is correct if there are no **character** arguments.

T and TL Formats

The implementation of the **t** (absolute tab) and **tl** (leftward tab) format codes is defective. These codes allow rereading or rewriting part of the record which has already been processed. The implementation uses "seeks"; so if the unit is not one which allows seeks (such as a terminal) the program is in error. A benefit of the implementation chosen is that there is no upper limit on the length of a record nor is it necessary to predeclare any record lengths except where specifically required by Fortran or the operating system.

INTERPROCEDURE INTERFACE

To be able to write C language procedures that call or are called by Fortran procedures, it is necessary to know the conventions for procedure names, data representation, return values, and argument lists that the compiled code obeys.

Procedure Names

On UNIX systems, the name of a common block or a Fortran procedure has an underscore appended to it by the compiler to distinguish it from a C language procedure or external variable with the same user-assigned name. Fortran library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

Data Representations

The following is a table of corresponding Fortran and C language declarations:

Fortran	C Language
`integer*2 x`	`short int x;`
`integer x`	`long int x;`
`logical x`	`long int x;`
`real x`	`float x;`
`double precision x`	`double x;`
`complex x`	`struct { float r, i; } x;`
`double complex x`	`struct { double dr, di; } x;`
`character*6 x`	`char x[6];`

By the rules of Fortran, **integer, logical,** and **real** data occupy the same amount of memory.

Return Values

A function of type **integer, logical, real,** or **double precision** declared as a C language function returns the corresponding type. A **complex** or **double complex** function is equivalent to a C language routine with an additional initial argument that points to the place where the return value is to be stored. Thus, the following:

```
complex function f( . . . )
```

is equivalent to

```
struct { float r, i; } temp;
f_(&temp, . . .)
. . .
```

A character-valued function is equivalent to a C language routine with two extra initial arguments - a data address and a length. Thus,

```
character*15 function g( . . . )
```

is equivalent to

```
char result[ ];
long int length;
g_(result, length, . . .)
. . .
```

and could be invoked in C language by

```
char chars[15];
. . .
g_(chars, 15L, . . . );
```

Subroutines are invoked as if they were **integer**-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function but are used to do an indexed branch in the calling procedure. (If the subroutine has no entry points with alternate return arguments, the returned value is undefined.) The statement

```
call nret(*1, *2, *3)
```

is treated exactly as if it were the computed **goto**

```
goto (1, 2, 3), nret( )
```

Argument Lists

All Fortran arguments are passed by address. In addition, for every argument that is of type character or that is a dummy procedure, an argument giving the length of the value is passed. (The string lengths are **long int** quantities passed by value.) The order of arguments is then:

Extra arguments for complex and character functions
Address for each datum or function
A **long int** for each character or procedure argument

Thus, the call in

```
external f
character*7 s
integer b(3)
...
call sam(f, b(2), s)
```

is equivalent to that in

```
int f();
char s[7];
long int b[3];
...
sam_(f, &b[1], s, 0L, 7L);
```

Note that the first element of a C language array always has subscript 0, but Fortran arrays begin at 1 by default. Fortran arrays are stored in column-major order; C language arrays are stored in row-major order.

FILE FORMATS

Structure of Fortran Files

Fortran requires four kinds of external files: *sequential formatted* and *unformatted*, and *direct formatted* and *unformatted*. On UNIX systems, these are all implemented as ordinary files which are assumed to have the proper internal structure.

Fortran I/O is based on "records." When a direct file is opened in a Fortran program, the record length of the records must be given; and this is used by the Fortran I/O system to make the file look as if it is made up of records of the given length. In the special case that the record length is given as 1, the files are not considered to be divided into records but are treated as byte-addressable byte strings; i.e., as ordinary files on the UNIX system. (A read or write request on such a file keeps consuming bytes until satisfied rather than being restricted to a single record.)

The peculiar requirements on sequential unformatted files make it unlikely that they will ever be read or written by any means except Fortran I/O statements. Each record is preceded and followed by an integer containing the record's length in bytes.

The Fortran I/O system breaks sequential formatted files into records while reading by using each new-line as a record separator. The result of reading off the end of a record is undefined according to the Fortran 77 American National Standard. The I/O system is permissive and treats the record as being extended by blanks. On output, the I/O system will write a new-line at the end of each record. It is also possible for programs to write new-lines for themselves. This is an error, but the only effect will be that the single record the user thought was written will be treated as more than one record when being read or backspaced over.

Preconnected Files and File Positions

Units 5, 6, and 0 are preconnected when the program starts. Unit 5 is connected to the standard input, unit 6 is connected to the standard output, and unit 0 is connected to the standard error unit. All are connected for sequential formatted I/O.

All the other units are also preconnected when execution begins. Unit *n* is connected to a file named **fort.***n*. These files need not exist nor will they be created unless their units are used without first executing an **open.** The default connection is for sequential formatted I/O.

The Fortran 77 Standard does not specify where a file which has been explicitly **opened** for sequential I/O is initially positioned. In fact, the I/O system attempts to position the file at the end. A **write** will append to the file and a **read** will result in an "end of file" indication. To position a file to its beginning, use a **rewind** statement. The preconnected units 0, 5, and 6 are positioned as they come from the parent process.

Chapter 13

RATFOR

GENERAL

This chapter describes the Ratfor(1) preprocessor. It is assumed that the user is familiar with the current implementation of Fortran 77 on the UNIX system.

The Ratfor language allows users to write Fortran programs in a fashion similar to C language. The Ratfor program is implemented as a preprocessor that translates this "simplified" language into Fortran. The facilities provided by Ratfor are:

- Statement grouping
- if-else and switch for decision making
- while, for, do, and repeat-until for looping
- break and next for controlling loop exits
- Free form input such as multiple statements/lines and automatic continuation
- Simple comment convention
- Translation of >, >=, etc., into .gt., .ge., etc.
- return statement for functions
- define statement for symbolic parameters
- include statement for including source files.

USAGE

The Ratfor program takes either a list of file names or the standard input and writes Fortran on the standard output. Options include **-6x**, which uses x as a continuation character in column 6 (the UNIX system uses *&* in column 1), **-h**, which causes quoted strings to be turned into *n*H constructs and **-C**, which causes Ratfor comments to be copied into the generated Fortran.

STATEMENT GROUPING

The Ratfor language provides a statement grouping facility. A group of statements can be treated as a unit by enclosing them in the braces { and }. For example, the Ratfor code

```
if (x > 100)
    { call error(" x>100" ); err = 1; return }
```

will be translated by the Ratfor preprocessor into **Fortran** equivalent to

```
      if (x .le. 100) goto 10
          call error(5hx>100)
          err = 1
          return
10    ...
```

which should simplify programming effort. By using { and }, a group of statements can be used instead of a single statement.

Also note in the previous Ratfor example that the character > was used instead of **.GT.** in the **if** statement. The Ratfor preprocessor translates this C language type operator to the appropriate Fortran operator. More on relationship operators later.

In addition, many Fortran compilers permit character strings in quotes (like " *x>100*"). But others, like ANSI Fortran 66, do not. Ratfor converts it into the right number of *H*s.

The Ratfor language is free form. Statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. The previous example could also be written as

```
if (x > 100) {
    call error(" x>100" )
    err = 1
    return
}
```

which shows grouped statements spread over several lines. In this case, no semicolon is needed at the end of each line because Ratfor assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the **if** is a single statement, no braces are needed.

THE "if-else" CONSTRUCTION

The Ratfor language provides an **else** statement. The syntax of the **if-else** construction is:

if (*legal Fortran condition*)
    ratfor statement
else
    ratfor statement

where the **else** part is optional. The legal Fortran condition is anything that can legally go into a Fortran Logical **IF** statement. The Ratfor preprocessor does not check this clause since it does not know enough Fortran to know what is permitted. The "ratfor" statement is any Ratfor or Fortran statement or any collection of them in braces. For example:

```
if (a <= b)
    { sw = 0; write(6, 1) a, b }
else
    { sw = 1; write(6, 1) b, a }
```

is a valid Ratfor **if-else** construction. This writes out the smaller of a and b, then the larger, and sets sw appropriately.

As before, if the statement following an **if** or an **else** is a single statement, no braces are needed.

Nested "if" Statements

The statement that follows an **if** or an **else** can be any Ratfor statement including another **if** or **else** statement. In general, the structure

```
if (condition) action
else if (condition) action
else action
```

provides a way to write a multibranch in Ratfor. (The Ratfor language also provides a **switch** statement which could be used instead, under certain conditions.) The last **else** handles the "default" condition. If there is no default action, this final **else** can be omitted. Thus, only the actions associated with the valid condition are performed. For example:

```
if (x < 0)
    x = 0
else if (x > 100)
    x = 100
```

will ensure that x is not less than 0 and not greater than 100.

Nested **if** and **else** statements could result in ambiguous code. In Ratfor when there are more **if** statements than **else** statements, **else** statements are associated with the closest previous **if** statement that currently does not have an associated **else** statement. For example:

```
if (x > 0)
if (y > 0)
write(6,1) x, y
else
write(6,2) y
```

is interpreted by the Ratfor preprocessor as

```
if (x > 0) {
    if (y > 0)
        write(6, 1) x, y
    else
        write(6, 2) y
}
```

in which the braces are assumed. If the other association is desired it must be written as

```
if (x > 0) {
    if (y > 0)
        write(6, 1) x, y
}
else
    write(6, 2) y
```

with the braces specified.

THE "switch" STATEMENT

The **switch** statement provides a way to express multiway branches which branch on the value of some *integer*-valued expression. The syntax is

```
switch (expression) {
    case expr1 :
    statements
    case expr2, expr3 :
    statements
    ...
    default:
    statements
}
```

where each **case** is followed by an integer expression (or several integer expressions separated by commas). The **switch** *expression* is compared to each **case** *expr* until a match is found. Then the *statements* following that **case** are executed. If no **cases** match

expression, then the *statements* following **default** are executed. The **default** section of a **switch** is optional.

When the *statements* associated with a **case** are executed, the entire **switch** is exited immediately. This is different from C language.

THE "do" STATEMENT

The **do** statement in Ratfor is quite similar to the **DO** statement in Fortran except that it uses no statement number (braces are used to mark the end of the **do** instead of a statement number). The syntax of the **ratfor do** statement is

```
do legal-Fortran-DO-text {
  ratfor statements
}
```

The *legal-Fortran-DO-text* must be something that can legally be used in a Fortran **DO** statement. Thus if a local version of Fortran allows **DO** limits to be expressions (which is not currently permitted in ANSI Fortran 66), they can be used in a **ratfor do** statement. The *ratfor statements* are enclosed in braces; but as with the **if**, a single statement need not have braces around it. For example, the following code sets an array to zero:

```
do i = 1, n
    x(i) = 0.0
```

and the code

```
do i = 1, n
  do j = 1, n
    m(i, j) = 0
```

sets the entire array *m* to zero.

THE "break" AND "next" STATEMENTS

The Ratfor **break** and **next** statements provide a means for leaving a loop early and one for beginning the next iteration. The **break** causes an immediate exit from the **do**; in effect, it is a branch to the statement *after* the **do**. The **next** is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this code skips over negative values in an array

```
do i = 1, n {
  if (x(i) < 0.0)
    next
  process positive element
}
```

The **break** and **next** statements will also work in the other Ratfor looping constructions and will be discussed with each looping construction.

The **break** and **next** can be followed by an integer to indicate breaking or iterating that level of enclosing loop. For example:

```
break 2
```

exits from two levels of enclosing loops, and

```
break 1
```

is equivalent to **break**. The

```
next 2
```

iterates the second enclosing loop.

THE "while" STATEMENT

The Ratfor language provides a **while** statement. The syntax of the **while** statement is

while (*legal-Fortran-condition*)
  ratfor statement

As with the **if**, legal-Fortran-condition is something that can go into a Fortran Logical **IF**, and ratfor statement is a single statement which may be multiple statements enclosed in braces.

For example, suppose nextch is a function which returns the next input character both as a function value and in its argument. Then a **while** loop to find the first nonblank character could be

```
while (nextch(ich) == iblank)
    ;
```

where a semicolon by itself is a null statement (which is necessary here to mark the end of the **while**). If the semicolon were not present, the **while** would control the next statement. When the loop is exited, ich contains the first nonblank.

THE "for" STATEMENT

The **for** statement is another Ratfor loop. A **for** statement allows explicit initialization and increment steps as part of the statement.

The syntax of the **for** statement is

for (*init* ; *condition* ; *increment*)
  ratfor statement

where *init* is any single Fortran statement which is executed once before the loop begins. The increment is any single Fortran statement that is executed at the end of each pass through the loop before the test. The *condition* is again anything that is legal in a

Fortran Logical **IF**. Any of init, condition, and increment may be omitted although the semicolons must always be present. A nonexistent condition is treated as always true, so

```
for (;;)
```

is an infinite loop.

For example, a Fortran **DO** loop could be written as

```
for (i = 1; i <= n; i = i + 1) ...
```

which is equivalent to

```
i = 1
while (i <= n) {
    ...
    i = i + 1
}
```

The initialization and increment of *i* have been moved into the **for** statement.

The **for, do,** and **while** versions have the advantage that they will be done zero times if *n* is less than 1. In addition, the **break** and **next** statements work in a **for** loop.

The *increment* in a **for** need not be an arithmetic progression. The program

```
sum = 0.0
for (i = first; i > 0; i = ptr(i))
    sum = sum + value(i)
```

steps through a list (stored in an integer array *ptr*) until a zero pointer is found while adding up elements from a parallel array of values. Notice that the code also works correctly if the list is empty.

THE "repeat-until" STATEMENT

There are times when a test needs to be performed at the bottom of a loop after one pass through. This facility is provided by the **repeat-until** statement. The syntax for the **repeat-until** statement is

repeat
  ratfor statement
until (*legal-Fortran-condition*)

where ratfor-statement is done once, then the *condition* is evaluated. If it is true, the loop is exited; if it is false, another pass is made.

The **until** part is optional, so a **repeat** by itself is an infinite loop. A **repeat-until** loop can be exited by the use of a **stop**, **return**, or **break** statement or an implicit stop such as running out of input with a **READ** statement.

As stated before, a **break** statement causes an immediate exit from the enclosing **repeat-until** loop. A **next** statement will cause a skip to the bottom of a **repeat-until** loop (i.e., to the **until** part).

THE "return" STATEMENT

The standard Fortran mechanism for returning a value from a routine uses the name of the routine as a variable. This variable can be assigned a value. The last value stored in it is the value returned by the function. For example, in a Fortran routine named *equal*, the statements

```
equal = 0
return
```

cause *equal* to return zero.

The Ratfor language provides a **return** statement similar to the C language **return** statement. In order to return a value from any routine, the **return** statement has the syntax

return (*expression*)

where *expression* is the value to be returned.

If there is no parenthesized expression after **return**, no value is returned.

THE "define" STATEMENT

The Ratfor language provides a **define** statement similar to the C language version. Any string of alphanumeric characters can be defined as a name. Whenever that name occurs in the input (delimited by nonalphanumerics), it is replaced by the rest of the definition line. (Comments and trailing white spaces are stripped off.) A defined name can be arbitrarily long and must begin with a letter.

Usually the **define** statement is used for symbolic parameters. The syntax of the **define** statement is

define name value

where *name* is a symbolic name that represents the quantity of *value*. For example:

```
define ROWS 100
define CLOS 50
dimension a(ROWS), b(ROWS, COLS)
    if (i > ROWS | j > COLS) ...
```

causes the preprocessor to replace the name *ROWS* with the value *100* and the name *COLS* with the value *50*. Alternately, definitions may be written as

```
define(ROWS, 100)
```

in which case the defining text is everything after the comma up to the right parenthesis. This allows multiple-line definitions.

THE "include" STATEMENT

The Ratfor language provides an **include** statement similar to the **#include <...>** statement in C language. The syntax for this statement is

include *file*

which inserts the contents of the named file into the Ratfor input file in place of the **include** statement. The standard usage is to place **COMMON** blocks on a file and use the **include** statement to include the common code whenever needed.

FREE-FORM INPUT

In Ratfor, statements can be placed anywhere on a line. Long statements are continued automatically as are long conditions in **if**, **for**, and **until** statements. Blank lines are ignored. Multiple statements may appear on one line if they are separated by semicolons. No semicolon is needed at the end of a line if Ratfor can make some reasonable guess about whether the statement ends there. Lines ending with any of the characters

```
= + - * , | & ( _
```

are assumed to be continued on the next line. Underscores are discarded wherever they occur. All other characters remain as part of the statement.

Any statement that begins with an all-numeric field is assumed to be a Fortran label and placed in columns 1 through 5 upon output. Thus:

```
write(6, 100); 100 format(" hello" )
```

is converted into

```
      write(6, 100)
100   format(5hhello)
```

TRANSLATIONS

When the **-h** option is chosen, text enclosed in matching single or double quotes is converted to *nH...* but is otherwise unaltered (except for formatting — it may get split across card boundaries during the reformatting process). Within quoted strings, the backslash (\) serves as an escape character; i.e., the next character is taken literally. This provides a way to get quotes and the backslash itself into quoted strings. For example:

```
"\'"
```

is a string containing a backslash and an apostrophe. (This is not the standard convention of doubled quotes, but it is easier to use and more general.)

Any line that begins with the character % is left absolutely unaltered except for stripping off the % and moving the line one position to the left. This is useful for inserting control cards and other things that should not be preprocessed (like an existing Fortran program). Use % only for ordinary statements not for the condition parts of **if**, **while**, etc., or the output may come out in an unexpected place.

The following character translations are made (except within single or double quotes or on a line beginning with a %):

== .eq.

!= .ne.

> .gt.

>= .ge.

< .lt.

<= .le.

& .and.

| .or.

! .not.

In addition, the following translations are provided for input devices with restricted character sets:

[{

] }

$({

$) }

WARNINGS

The Ratfor preprocessor catches certain syntax errors (such as missing braces), **else** statements without **if** statements, and most errors involving missing parentheses in statements.

All other errors are reported by the Fortran compiler. Unfortunately, the Fortran compiler prints messages in terms of generated Fortran code and not in terms of the Ratfor code. This makes it difficult to locate Ratfor statements that contain errors.

The keywords are deserved. Using **if, else, while,** etc., as variable names will cause considerable problems. Likewise, spaces within keywords and use of the Arithmetic **IF** will cause problems.

The Fortran *n*H convention is not recognized by Ratfor. Use quotes instead.

EXAMPLE OF RATFOR CONVERSION

As an example of how to use the Ratfor program, the following program **prog.r** (where the **.r** indicates a Ratfor source program), is written in the Ratfor language:

```
      ICNT=0
   10 WRITE(6,31)
   31 FORMAT(" INPUT FIRST NUMBER" )
      READ(5,32) A
   32 FORMAT(F10.2)
      WRITE(6,33)
   33 FORMAT(" INPUT SECOND NUMBER" )
      READ(5,34) B
   34 FORMAT(F10.2)
      IF(A<B)
        WRITE(6,36) A,B
      ELSE WRITE(6,37)A,B
   36 FORMAT(F10.2," < ",F10.2)
   37 FORMAT(F10.2," >= ",F10.2)
      ICNT=ICNT+1
      IF(ICNT.EQ.5)
        GOTO 100
      GOTO 10
  100 END
```

The command

```
ratfor prog.r > prog.f
```

causes the Fortran translation program **prog.f** to be produced. (The Ratfor program **prog.r** remains intact.) The Fortran program **prog.f** follows:

```
      icnt=0
10    write(6,31)
31    format(" INPUT FIRST NUMBER" )
      read(5,32) a
32    format(f10.2)
      write(6,33)
33    format(" INPUT SECOND NUMBER" )
      read(5,34) b
34    format(f10.2)
      if(.not.(a.lt.b))goto 23000
      write(6,36) a,b
      goto 23001
23000 continue
      write(6,37)a,b
23001 continue
36    format(f10.2," < ",f10.2)
37    format(f10.2," >= ",f10.2)
      icnt=icnt+1
      if(.not.(icnt.eq.5))goto 23002
      goto 100
23002 continue
      goto 10
100   end
```

The Fortran program **prog.f** is compiled using the command

```
f77 prog.f
```

An object program file **prog.o** and a final output file **a.out** are produced. Since the output file **a.out** is an executable file, the command

```
a.out
```

causes the program to run.

The Ratfor program **prog.r** can also be translated and compiled with the single command

```
f77 prog.r
```

where the **.r** indicates a Ratfor source program. An object file **prog.o** and a final output file **a.out** are produced.

Chapter 14

THE PROGRAMMING LANGUAGE EFL

INTRODUCTION

EFL is a clean, general purpose computer language intended to encourage portable programming. It has a uniform and readable syntax and good data and control flow structuring. EFL programs can be translated into efficient Fortran code, so the EFL programmer can take advantage of the ubiquity of Fortran, the valuable libraries of software written in that language, and the portability that comes with the use of a standardized language, without suffering from Fortran's many failings as a language. It is especially useful for numeric programs. Thus, the EFL language permits the programmer to express complicated ideas in a comprehensible way, while permitting access to the power of the Fortran environment.

The name EFL originally stood for "Extended Fortran Language." The current compiler is much more than a simple preprocessor: it attempts to diagnose all syntax errors, to provide readable Fortran output, and to avoid a number of niggling restrictions.

In examples and syntax specifications, **boldface** type is used to indicate literal words and punctuation, such as **while**. Words in *italic* type indicate an item in a category, such as an *expression*. A construct surrounded by double brackets represents a list of one or more of those items, separated by commas. Thus, the notation

[[*item*]]

could refer to any of the following:

item
item, *item*
item, *item*, *item*

The reader should have a fair degree of familiarity with some procedural language. There will be occasional references to Ratfor and to Fortran which may be ignored if the reader is unfamiliar with those languages.

LEXICAL FORM

Character Set

The following characters are legal in an EFL program:

letters	**a b c d e f g h i j k l m**
	n o p q r s t u v w x y z
digits	**0 1 2 3 4 5 6 7 8 9**
white space	*blank tab*
quotes	' "
sharp	#
continuation	_
braces	{ }
parentheses	( )
other	, ; : . + - * /
	= < > & ~ ! $

Letter case (upper or lower) is ignored except within strings, so "a" and "A" are treated as the same character. All of the examples below are printed in lower case. An exclamation mark ("!") may be used in place of a tilde ("~"). Square brackets ("[" and "]") may be used in place of braces ("{" and "}").

Lines

EFL is a line-oriented language. Except in special cases (discussed below), the end of a line marks the end of a token and the end of a statement. The trailing portion of a line may be used for a comment. There is a mechanism for diverting input from one source file to another, so a single line in the program may be replaced by a number of lines from the other file. Diagnostic messages are labeled with the line number of the file on which they are detected.

White Space

Outside of a character string or comment, any sequence of one or more spaces or tab characters acts as a single space. Such a space terminates a token.

Comments

A comment may appear at the end of any line. It is introduced by a sharp (#) character, and continues to the end of the line. (A sharp inside of a quoted string does not mark a comment.) The sharp and succeeding characters on the line are discarded. A blank line is also a comment. Comments have no effect on execution.

Include Files

It is possible to insert the contents of a file at a point in the source text, by referencing it in a line like

include joe

No statement or comment may follow an **include** on a line. In effect, the **include** line is replaced by the lines in the named file, but diagnostics refer to the line number in the included file. **Includes** may be nested at least ten deep.

Continuation

Lines may be continued explicitly by using the underscore (_) character. If the last character of a line (after comments and trailing white space have been stripped) is an underscore, the end of a line and the initial blanks on the next line are ignored. Underscores are ignored in other contexts (except inside of quoted strings). Thus

```
1_000_000_
000
```

equals 10^9.

There are also rules for continuing lines automatically: the end of line is ignored whenever it is obvious that the statement is not complete. To be specific, a statement is continued if the last token on a line is an operator, comma, left brace, or left parenthesis. (A statement is not continued just because of unbalanced braces or parentheses.) Some compound statements are also continued automatically; these points are noted in the sections on executable statements.

Multiple Statements on a Line

A semicolon terminates the current statement. Thus, it is possible to write more than one statement on a line. A line consisting only of a semicolon, or a semicolon following a semicolon, forms a null statement.

Tokens

A program is made up of a sequence of tokens. Each token is a sequence of characters. A blank terminates any token other than a quoted string. End of line also terminates a token unless explicit continuation (see above) is signaled by an underscore.

Identifiers

An identifier is a letter or a letter followed by letters or digits. The following is a list of the reserved words that have special meaning in EFL. They will be discussed later.

array	**exit**	**precision**
automatic	**external**	**procedure**
break	**false**	**read**
call	**field**	**readbin**
case	**for**	**real**
character	**function**	**repeat**
common	**go**	**return**
complex	**goto**	**select**
continue	**if**	**short**
debug	**implicit**	**sizeof**
default	**include**	**static**
define	**initial**	**struct**
dimension	**integer**	**subroutine**
do	**internal**	**true**
double	**lengthof**	**until**
doubleprecision	**logical**	**value**
else	**long**	**while**
end	**next**	**write**
equivalence	**option**	**writebin**

The use of these words is discussed below. These words may not be used for any other purpose.

Strings

A character string is a sequence of characters surrounded by quotation marks. If the string is bounded by single-quote marks ('), it may contain double quote marks ("), and vice versa. A quoted string may not be broken across a line boundary.

'hello there'
" ain't misbehavin'"

Integer Constants

An integer constant is a sequence of one or more digits.

0
57
123456

Floating Point Constants

A floating point constant contains a dot and/or an exponent field. An *exponent field* is a letter **d** or **e** followed by an optionally signed integer constant. If *I* and *J* are integer constants and *E* is an exponent field, then a floating constant has one of the following forms:

.J
I.
I.J
IE
I.E
.IE
I.JE

Punctuation

Certain characters are used to group or separate objects in the language. These are

parentheses	( )
braces	{ }
comma	,
semicolon	;
colon	:
end-of-line	

The end-of-line is a token (statement separator) when the line is neither blank nor continued.

Operators

The EFL operators are written as sequences of one or more non-alphanumeric characters.

```
+  -  *  /  **
<  <=  >  >=  ==  ~=
&&  ||  &  |
+=  -=  /=  **=
&&=  ||=  &=  |=
->  .  $
```

A dot (".") is an operator when it qualifies a structure element name, but not when it acts as a decimal point in a numeric constant. There is a special mode (see " ATAVISMS") in which some of the operators may be represented by a string consisting of a dot, an identifier, and a dot (*e.g.*, **.lt.**).

Macros

EFL has a simple macro substitution facility. An identifier may be defined to be equal to a string of tokens; whenever that name appears as a token in the program, the string replaces it. A macro name is given a value in a **define** statement like

```
define count    n += 1
```

Any time the name **count** appears in the program, it is replaced by the statement

```
n += 1
```

A **define** statement must appear alone on a line; the form is

define *name rest-of-line*

Trailing comments are part of the string.

PROGRAM FORM

Files

A *file* is a sequence of lines. A file is compiled as a single unit. It may contain one or more procedures. Declarations and options that appear outside of a procedure affect the succeeding procedures on that file.

Procedures

Procedures are the largest grouping of statements in EFL. Each procedure has a name by which it is invoked. (The first procedure invoked during execution, known as the *main* procedure, has the null name.) Procedure calls and argument passing are discussed in " PROCEDURES."

Blocks

Statements may be formed into groups inside of a procedure. To describe the scope of names, it is convenient to introduce the ideas of *block* and of *nesting level.* The beginning of a program file is at nesting level zero. Any options, macro definitions, or variable declarations are also at level zero. The text immediately following a **procedure** statement is at level 1. After the declarations, a left brace marks the beginning of a new block and increases the nesting level by 1; a right brace drops the level by 1. (Braces inside declarations do not mark blocks.) (See " Blocks" under " EXECUTABLE STATEMENTS.") An **end** statement marks the end of the procedure, level 1, and the return to level 0. A name (variable or macro) that is defined at level K is defined throughout that block

and in all deeper nested levels in which that name is not redefined or redeclared. Thus, a procedure might look like the following:

```
#  block 0
procedure george
real x
x = 2
...
if(x > 2)
        {       # new block
        integer x # a different variable
        do x = 1,7
                write(,x)
        ...
        }       # end of block
end     # end of procedure, return to block 0
```

Statements

A statement is terminated by end of line or by a semicolon. Statements are of the following types:

Option
Include
Define

Procedure
End

Declarative
Executable

The **option** statement is described in " COMPILER OPTIONS". The **include, define,** and **end** statements have been described above; they may not be followed by another statement on a line. Each procedure begins with a **procedure** statement and finishes with an **end** statement; these are discussed in " PROCEDURES". Declarations describe types and values of variables and procedures. Executable statements cause specific actions to be taken. A block is an example of an executable statement; it is made up of declarative and executable statements.

Labels

An executable statement may have a *label* which may be used in a branch statement. A label is an identifier followed by a colon, as in

```
          read(, x)
          if(x < 3) goto error
          ...
error:    fatal(" bad input" )
```

DATA TYPES AND VARIABLES

EFL supports a small number of basic (scalar) types. The programmer may define objects made up of variables of basic type; other aggregates may then be defined in terms of previously defined aggregates.

Basic Types

The basic types are

logical
integer
field(*m*:*n*)
real
complex
long real
long complex
character(*n*)

A logical quantity may take on the two values *true* and *false*. An integer may take on any whole number value in some machine-dependent range. A field quantity is an integer restricted to a particular closed interval ([*m*:*n*]). A "real" quantity is a floating point approximation to a real or rational number. A long real is a more precise approximation to a rational. (Real quantities are represented as single precision floating point numbers; long reals are double precision floating point numbers.) A complex quantity is an approximation to a complex number, and is represented as a pair of reals. A character quantity is a fixed-length string of *n* characters.

Constants

There is a notation for a constant of each basic type.

A logical may take on the two values

true
false

An integer or field constant is a fixed point constant, optionally preceded by a plus or minus sign, as in

17
−94
+6
0

A long real ("double precision") constant is a floating point constant containing an exponent field that begins with the letter **d.** A real ("single precision") constant is any other floating point constant. A real or long real constant may be preceded by a plus or minus sign. The following are valid **real** constants:

17.3
−.4
7.9e−6 (= 7.9×10^{-6})
14e9 (= 1.4×10^{10})

The following are valid **long real** constants

7.9d−6 (= 7.9×10^{-6})
5d3

A character constant is a quoted string.

Variables

A variable is a quantity with a name and a location. At any particular time the variable may also have a value. (A variable is said to be *undefined* before it is initialized or assigned its first value, and after certain indefinite operations are performed.) Each variable has certain attributes:

Storage Class

The association of a name and a location is either transitory or permanent. Transitory association is achieved when arguments are passed to procedures. Other associations are permanent (static). (A future extension of EFL may include dynamically allocated variables.)

Scope of Names

The names of common areas are global, as are procedure names: these names may be used anywhere in the program. All other names are local to the block in which they are declared.

Precision

Floating point variables are either of normal or **long** precision. This attribute may be stated independently of the basic type.

Arrays

It is possible to declare rectangular arrays (of any dimension) of values of the same type. The index set is always a cross-product of intervals of integers. The lower and upper bounds of the intervals must be constants for arrays that are local or **common.** A formal argument array may have intervals that are of length equal to one of the other formal arguments. An element of an array is denoted by the array name followed by a parenthesized comma-separated list of integer values, each of which must lie within the corresponding interval. (The intervals may include negative numbers.) Entire arrays may be passed as procedure arguments or in input/output lists, or they may be initialized; all other array references must be to individual elements.

Structures

It is possible to define new types which are made up of elements of other types. The compound object is known as a *structure;* its constituents are called *members* of the structure. The structure may be given a name, which acts as a type name in the remaining statements within the scope of its declaration. The elements of a structure may be of any type (including previously defined structures), or they may be arrays of such objects. Entire structures may be passed to procedures or be used in input/output lists; individual elements of structures may be referenced. The uses of structures will be detailed below. The following structure might represent a symbol table:

```
struct tableentry
        {
        character(8) name
        integer hashvalue
        integer numberofelements
        field(0:1) initialized, used, set
        field(0:10) type
        }
```

EXPRESSIONS

Expressions are syntactic forms that yield a value. An expression may have any of the following forms, recursively applied:

primary
(*expression*)
unary-operator expression
expression binary-operator expression

In the following table of operators, all operators on a line have equal precedence and have higher precedence than operators on later lines. The meanings of these operators are described in " Unary Operators" and " Binary Operators" under " EXPRESSIONS".

C-14

```
->  .
**
*  /     unary +  -  ++  --
+  -
<  <=  >  >=  ==  ~=
&  &&
|  ||
$
=  +=  -=  *=  /=  **=    &=  |=  &&=  ||=
```

Examples of expressions are

```
a<b && b<c
-(a + sin(x)) / (5+cos(x))**2
```

Primaries

Primaries are the basic elements of expressions. They include constants, variables, array elements, structure members, procedure invocations, input/output expressions, coercions, and sizes.

Constants

Constants are described in "Constants" under "DATA TYPES AND VARIABLES".

Variables

Scalar variable names are primaries. They may appear on the left or the right side of an assignment. Unqualified names of aggregates (structures or arrays) may appear only as procedure arguments and in input/output lists.

Array Elements

An element of an array is denoted by the array name followed by a parenthesized list of subscripts, one integer value for each declared dimension:

```
a(5)
b(6,-3,4)
```

Structure Members

A structure name followed by a dot followed by the name of a member of that structure constitutes a reference to that element. If that element is itself a structure, the reference may be further qualified.

```
a.b
x(3).y(4).z(5)
```

Procedure Invocations

A procedure is invoked by an expression of one of the forms

procedurename ( )
procedurename (*expression*)
procedurename (*expression-1*, ..., *expression-n*)

The *procedurename* is either the name of a variable declared **external** or it is the name of a function known to the EFL compiler (see " Known Functions" under " PROCEDURES"), or it is the actual name of a procedure, as it appears in a **procedure** statement. If a *procedurename* is declared **external** and is an argument of the current procedure, it is associated with the procedure name passed as actual argument; otherwise it is the actual name of a procedure. Each *expression* in the above is called an *actual argument*. Examples of procedure invocations are

```
f(x)
work()
g(x, y+3, 'xx')
```

When one of these procedure invocations is to be performed, each of the actual argument expressions is first evaluated. The types, precisions, and bounds of actual and formal arguments should agree. If an actual argument is a variable name, array element, or structure member, the called procedure is permitted to use the corresponding

formal argument as the left side of an assignment or in an input list; otherwise it may only use the value. After the formal and actual arguments are associated, control is passed to the first executable statement of the procedure. When a **return** statement is executed in that procedure, or when control reaches the **end** statement of that procedure, the function value is made available as the value of the procedure invocation. The type of the value is determined by the attributes of the *procedurename* that are declared or implied in the calling procedure, which must agree with the attributes declared for the function in its procedure. In the special case of a generic function, the type of the result is also affected by the type of the argument. See " PROCEDURES" .

Input/Output Expressions

The EFL input/output syntactic forms may be used as integer primaries that have a non-zero value if an error occurs during the input or output. See " Input/Output Statements" under " EXECUTABLE STATEMENTS" .

Coercions

An expression of one precision or type may be converted to another by an expression of the form

attributes (*expression*)

At present, the only *attributes* permitted are precision and basic types. Attributes are separated by white space. An arithmetic value of one type may be coerced to any other arithmetic type; a character expression of one length may be coerced to a character expression of another length; logical expressions may not be coerced to a nonlogical type. As a special case, a quantity of **complex** or **long complex** type may be constructed from two integer or real quantities by passing two expressions (separated by a comma) in the coercion. Examples and equivalent values are

integer(5.3) = 5
long real(5) = 5.0d0
complex(5,3) = 5+3i

Most conversions are done implicitly, since most binary operators permit operands of different arithmetic types. Explicit coercions are of most use when it is necessary to convert the type of an actual argument to match that of the corresponding formal parameter in a procedure call.

Sizes

There is a notation which yields the amount of memory required to store a datum or an item of specified type:

sizeof (*leftside*)
sizeof (*attributes*)

In the first case, *leftside* can denote a variable, array, array element, or structure member. The value of **sizeof** is an integer, which gives the size in arbitrary units. If the size is needed in terms of the size of some specific unit, this can be computed by division:

sizeof(x) / sizeof(integer)

yields the size of the variable x in integer words.

The distance between consecutive elements of an array may not equal **sizeof** because certain data types require final padding on some machines. The **lengthof** operator gives this larger value, again in arbitrary units. The syntax is

lengthof (*leftside*)
lengthof (*attributes*)

Parentheses

An expression surrounded by parentheses is itself an expression. A parenthesized expression must be evaluated before an expression of which it is a part is evaluated.

Unary Operators

All of the unary operators in EFL are prefix operators. The result of a unary operator has the same type as its operand.

Arithmetic

Unary + has no effect. A unary − yields the negative of its operand.

The prefix operator ++ adds one to its operand. The prefix operator −− subtracts one from its operand. The value of either expression is the result of the addition or subtraction. For these two operators, the operand must be a scalar, array element, or structure member of arithmetic type. (As a side effect, the operand value is changed.)

Logical

The only logical unary operator is complement (~). This operator is defined by the equations

~ true = false
~ false = true

Binary Operators

Most EFL operators have two operands, separated by the operator. Because the character set must be limited, some of the operators are denoted by strings of two or three special characters. All binary operators except exponentiation are left associative.

Arithmetic

The binary arithmetic operators are

+	addition
−	subtraction
*	multiplication
/	division
**	exponentiation

Exponentiation is right associative: a**b**c = a**(b**c) = $a^{(b^c)}$ The operations have the conventional meanings: 8+2 = 10, 8-2 = 6, 8*2 = 16, 8/2 = 4, 8**2 = 8^2 = 64.

The type of the result of a binary operation *A op B* is determined by the types of its operands:

	Type of B				
Type of A	i	r	l r	c	l c
i	i	r	l r	c	l c
r	r	r	l r	c	l c
l r	l r	l r	l r	l c	l c
c	c	c	l c	c	l c
l c	l c	l c	l c	l c	l c

i = integer
r = real
l r = long real
c = complex
l c = long complex

If the type of an operand differs from the type of the result, the calculation is done as if the operand were first coerced to the type of the result. If both operands are integers, the result is of type integer, and is computed exactly. (Quotients are truncated toward zero, so 8/3=2.)

Logical

The two binary logical operations in EFL, **and** and **or,** are defined by the truth tables:

A	B	A and B	A or B
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Each of these operators comes in two forms. In one form, the order of evaluation is specified. The expression

a && b

is evaluated by first evaluating **a**; if it is false then the expression is false and **b** is not evaluated; otherwise, the expression has the value of **b.** The expression

a || b

is evaluated by first evaluating **a**; if it is true then the expression is true and **b** is not evaluated; otherwise, the expression has the value of **b.** The other forms of the operators (**&** for **and** and **|** for **or**) do not imply an order of evaluation. With the latter operators, the compiler may speed up the code by evaluating the operands in any order.

Relational Operators

There are six relations between arithmetic quantities. These operators are not associative.

EFL Operator		Meaning
<	$<$	less than
<=	$\leq$	less than or equal to
==	$=$	equal to
~=	$\neq$	not equal to
>	$>$	greater than
>=	$\geq$	greater than or equal

Since the complex numbers are not ordered, the only relational operators that may take complex operands are == and ~= . The character collating sequence is not defined.

Assignment Operators

All of the assignment operators are right associative. The simple form of assignment is

basic-left-side = expression

A *basic-left-side* is a scalar variable name, array element, or structure member of basic type. This statement computes the expression on the right side, and stores that value (possibly after coercing the value to the type of the left side) in the location named by the left side. The value of the assignment expression is the value assigned to the left side after coercion.

There is also an assignment operator corresponding to each binary arithmetic and logical operator. In each case, *a op= b* is equivalent to *a = a op b*. (The operator and equal sign must not be separated by blanks.) Thus, **n+=2** adds 2 to n. The location of the left side is evaluated only once.

Dynamic Structures

EFL does not have an address (pointer, reference) type. However, there is a notation for dynamic structures,

leftside -> structurename

This expression is a structure with the shape implied by *structurename* but starting at the location of *leftside*. In effect, this overlays the structure template at the specified location. The *leftside* must be a variable, array, array element, or structure member. The type of the *leftside* must be one of the types in the structure declaration. An element of such a structure is denoted in the usual way using the dot operator. Thus,

place(i) -> st.elt

refers to the **elt** member of the **st** structure starting at the i^{th} element of the array **place.**

Repetition Operator

Inside of a list, an element of the form

integer-constant-expression $ *constant-expression*

is equivalent to the appearance of the *expression* a number of times equal to the first expression. Thus,

(3, 3$4, 5)

is equivalent to

(3, 4, 4, 4, 5)

Constant Expressions

If an expression is built up out of operators (other than functions) and constants, the value of the expression is a constant, and may be used anywhere a constant is required.

DECLARATIONS

Declarations statement describe the meaning, shape, and size of named objects in the EFL language.

Syntax

A declaration statement is made up of attributes and variables. Declaration statements are of two forms:

attributes variable-list
attributes { *declarations* }

In the first case, each name in the *variable-list* has the specified attributes. In the second, each name in the declarations also has the specified attributes. A variable name may appear in more than one variable list, so long as the attributes are not contradictory. Each

name of a nonargument variable may be accompanied by an initial value specification. The *declarations* inside the braces are one or more declaration statements. Examples of declarations are

```
integer k=2

long real b(7,3)

common(cname)
    {
    integer i
    long real array(5,0:3) x, y
    character(7) ch
    }
```

Attributes

Basic Types

The following are basic types in declarations

logical
integer
field(*m*:*n*)
character(*k*)
real
complex

In the above, the quantities k, m, and n denote integer constant expressions with the properties $k>0$ and $n>m$.

Arrays

The dimensionality may be declared by an **array** attribute

array(*b*$_1$,...,*b*$_n$)

Each of the b_i may either be a single integer expression or a pair of integer expressions separated by a colon. The pair of expressions form a lower and an upper bound; the single expression is an upper bound with an implied lower bound of 1. The number of dimensions is equal to n, the number of bounds. All of the integer expressions

must be constants. An exception is permitted only if all of the variables associated with an array declarator are formal arguments of the procedure; in this case, each bound must have the property that *upper-lower* +1 is equal to a formal argument of the procedure. (The compiler has limited ability to simplify expressions, but it will recognize important cases such as **(0:n−1)**. The upper bound for the last dimension (b_n) may be marked by an asterisk (*) if the size of the array is not known. The following are legal **array** attributes:

```
array(5)
array(5, 1:5, −3:0)
array(5, *)
array(0:m−1, m)
```

Structures

A structure declaration is of the form

struct *structname* { *declaration statements* }

The *structname* is optional; if it is present, it acts as if it were the name of a type in the rest of its scope. Each name that appears inside the *declarations* is a *member* of the structure, and has a special meaning when used to qualify any variable declared with the structure type. A name may appear as a member of any number of structures, and may also be the name of an ordinary variable, since a structure member name is used only in contexts where the parent type is known. The following are valid structure attributes

```
struct xx
      {
      integer a, b
      real x(5)
      }

struct { xx z(3); character(5) y }
```

The last line defines a structure containing an array of three **xx**'*s* and a character string.

Precision

Variables of floating point (**real** or **complex**) type may be declared to be **long** to ensure they have higher precision than ordinary floating point variables. The default precision is **short.**

Common

Certain objects called *common areas* have external scope, and may be referenced by any procedure that has a declaration for the name using a

common (*commonareaname* **)**

attribute. All of the variables declared with a particular **common** attribute are in the same block; the order in which they are declared is significant. Declarations for the same block in differing procedures must have the variables in the same order and with the same types, precision, and shapes, though not necessarily with the same names.

External

If a name is used as the procedure name in a procedure invocation, it is implicitly declared to have the **external** attribute. If a procedure name is to be passed as an argument, it is necessary to declare it in a statement of the form

external [[*name* **]]**

If a name has the external attribute and it is a formal argument of the procedure, then it is associated with a procedure identifier passed as an actual argument at each call. If the name is not a formal argument, then that name is the actual name of a procedure, as it appears in the corresponding **procedure** statement.

Variable List

The elements of a variable list in a declaration consist of a name, an optional dimension specification, and an optional initial value specification. The name follows the usual rules. The dimension specification is the same form and meaning as the parenthesized list in an **array** attribute. The initial value specification is an equal sign (=) followed by a constant expression. If the name is an array, the right side of the equal sign may be a parenthesized list of constant expressions, or repeated elements or lists; the total number of elements in the list must not exceed the number of elements of the array, which are filled in column-major order.

The Initial Statement

An initial value may also be specified for a simple variable, array, array element, or member of a structure using a statement of the form

initial [[*var* = *val*]]

The *var* may be a variable name, array element specification, or member of structure. The right side follows the same rules as for an initial value specification in other declaration statements.

EXECUTABLE STATEMENTS

Every useful EFL program contains executable statements, otherwise it would not do anything and would not need to be run. Statements are frequently made up of other statements. Blocks are the most obvious case, but many other forms contain statements as constituents.

To increase the legibility of EFL programs, some of the statement forms can be broken without an explicit continuation. A square (□) in the syntax represents a point where the end of a line will be ignored.

Expression Statements

Subroutine Call

A procedure invocation that returns no value is known as a subroutine call. Such an invocation is a statement. Examples are

```
work(in, out)
run()
```

Input/output statements (see "Input/Output Statements" under "EXECUTABLE STATEMENTS") resemble procedure invocations but do not yield a value. If an error occurs the program stops.

Assignment Statements

An expression that is a simple assignment (=) or a compound assignment (+= etc.) is a statement:

```
a = b
a = sin(x)/6
x *= y
```

Blocks

A block is a compound statement that acts as a statement. A block begins with a left brace, optionally followed by declarations, optionally followed by executable statements, followed by a right brace. A block may be used anywhere a statement is permitted. A block is not an expression and does not have a value. An example of a block is

```
{
integer i  # this variable is unknown
        # outside the braces
big = 0
do i = 1,n
   if(big < a(i))
        big = a(i)
}
```

Test Statements

Test statements permit execution of certain statements conditional on the truth of a predicate.

If Statement

The simplest of the test statements is the **if** statement, of form

if (*logical-expression*) ☐ *statement*

The logical expression is evaluated; if it is true, then the *statement* is executed.

If-Else

A more general statement is of the form

if (*logical-expression*) ☐ *statement-1* ☐
else ☐ *statement-2*

If the expression is **true** then *statement-1* is executed, otherwise, *statement-2* is executed. Either of the consequent statements may itself be an **if-else** so a completely nested test sequence is possible:

```
if(x<y)
   if(a<b)
      k = 1
   else
      k = 2
else
   if(a<b)
      m = 1
   else
      m = 2
```

An **else** applies to the nearest preceding un-**else**d **if**. A more common use is as a sequential test:

```
if(x==1)
   k = 1
else if(x==3 | x==5)
   k = 2
else
   k = 3
```

Select Statement

A multiway test on the value of a quantity is succinctly stated as a **select** statement, which has the general form

select(*expression* **)** □ *block*

Inside the block two special types of labels are recognized. A prefix of the form

case [[*constant*]] **:**

marks the statement to which control is passed if the expression in the select has a value equal to one of the case constants. If the expression equals none of these constants, but there is a label **default** inside the select, a branch is taken to that point; otherwise the statement following the right brace is executed. Once execution begins at a **case** or **default** label, it continues until the next **case** or **default** is encountered. The **else-if** example above is better written as

```
select(x)
   {
   case 1:
      k = 1
   case 3,5:
      k = 2
   default:
      k = 3
   }
```

Note that control does not "fall through" to the next case.

Loops

The loop forms provide the best way of repeating a statement or sequence of operations. The simplest (**while**) form is theoretically sufficient, but it is very convenient to have the more general loops available, since each expresses a mode of control that arises frequently in practice.

While Statement

This construct has the form

while (*logical-expression*) □ *statement*

The expression is evaluated; if it is true, the statement is executed, and then the test is performed again. If the expression is false, execution proceeds to the next statement.

For Statement

The **for** statement is a more elaborate looping construct. It has the form

for (*initial-statement* , □ *logical-expression* ,
□ *iteration-statement*) □ *body-statement*

Except for the behavior of the **next** statement (see " Branch Statement" under " EXECUTABLE STATEMENTS"), this construct is equivalent to

initial-statement
while (*logical-expression*)
{
body-statement
iteration-statement
}

This form is useful for general arithmetic iterations, and for various pointer-type operations. The sum of the integers from 1 to 100 can be computed by the fragment

```
n = 0
for(i = 1, i <= 100, i += 1)
    n += i
```

Alternatively, the computation could be done by the single statement

```
for( { n = 0 ; i = 1 } , i<=100 , { n += i ; ++i } )
    ;
```

Note that the body of the **for** loop is a null statement in this case. An example of following a linked list will be given later.

Repeat Statement

The statement

repeat □ *statement*

executes the statement, then does it again, without any termination test. Obviously, a test inside the *statement* is needed to stop the loop.

Repeat ... Until Statement

The **while** loop performs a test before each iteration. The statement

repeat □ *statement* □ **until** (*logical-expression*)

executes the *statement,* then evaluates the logical; if the logical is true the loop is complete; otherwise, control returns to the *statement.*

Thus, the body is always executed at least once. The **until** refers to the nearest preceding **repeat** that has not been paired with an **until.** In practice, this appears to be the least frequently used looping construct.

Do Loop

The simple arithmetic progression is a very common one in numerical applications. EFL has a special loop form for ranging over an ascending arithmetic sequence

do *variable = expression-1, expression-2, expression-3*
statement

The variable is first given the value *expression-1.* The statement is executed, then *expression-3* is added to the variable. The loop is repeated until the variable exceeds *expression-2.* If *expression-3* and the preceding comma are omitted, the increment is taken to be 1. The loop above is equivalent to

```
t2 = expression-2
t3 = expression-3
for(variable=expression-1, variable<=t2, variable+=t3)
    statement
```

(The compiler translates EFL **do** statements into Fortran DO statements, which are in turn usually compiled into excellent code.) The **do** *variable* may not be changed inside of the loop, and *expression-1* must not exceed *expression-2.* The sum of the first hundred positive integers could be computed by

```
n = 0
do i = 1, 100
        n += i
```

Branch Statements

Most of the need for branch statements in programs can be averted by using the loop and test constructs, but there are programs where they are very useful.

Goto Statement

The most general, and most dangerous, branching statement is the simple unconditional

goto *label*

After executing this statement, the next statement performed is the one following the given label. Inside of a **select** the case labels of that block may be used as labels, as in the following example:

```
select(k)
        {
        case 1:
                error(7)

        case 2:
                k = 2
                goto case 4

        case 3:
                k = 5
                goto case 4

        case 4:
                fixup(k)
                goto default

        default:
                prmsg(" ouch" )
        }
```

(If two **select** statements are nested, the case labels of the outer **select** are not accessible from the inner one.)

Break Statement

A safer statement is one which transfers control to the statement following the current **select** or loop form. A statement of this sort is almost always needed in a **repeat** loop:

```
repeat
    {
    do a computation
    if( finished )
        break
    }
```

More general forms permit controlling a branch out of more than one construct.

```
break 3
```

transfers control to the statement following the third loop and/or **select** surrounding the statement. It is possible to specify which type of construct (**for**, **while**, **repeat**, **do**, or **select**) is to be counted. The statement

```
break while
```

breaks out of the first surrounding **while** statement. Either of the statements

```
break 3 for
break for 3
```

will transfer to the statement after the third enclosing **for** loop.

Next Statement

The **next** statement causes the first surrounding loop statement to go on to the next iteration: the next operation performed is the test of a **while,** the *iteration-statement* of a **for,** the body of a **repeat,** the test of a **repeat...until,** or the increment of a **do.** Elaborations similar to those for **break** are available:

```
next
next 3
next 3 for
next for 3
```

A **next** statement ignores **select** statements.

Return

The last statement of a procedure is followed by a return of control to the caller. If it is desired to effect such a return from any other point in the procedure, a

```
return
```

statement may be executed. Inside a function procedure, the function value is specified as an argument of the statement:

return (*expression* **)**

Input/Output Statements

EFL has two input statements (**read** and **readbin**), two output statements (**write** and **writebin**), and three control statements (**endfile, rewind,** and **backspace**). These forms may be used either as a primary with a **integer** value or as a statement. If an exception occurs when one of these forms is used as a statement, the result is undefined but will probably be treated as a fatal error. If they are used in a context where they return a value, they return zero if no exception occurs. For the input forms, a negative value indicates end-of-file and a positive value an error. The input/output part of EFL very strongly reflects the facilities of Fortran.

Input/Output Units

Each I/O statement refers to a "unit," identified by a small positive integer. Two special units are defined by EFL, the *standard input unit* and the *standard output unit.* These particular units are assumed if no unit is specified in an I/O transmission statement.

The data on the unit are organized into *records.* These records may be read or written in a fixed sequence, and each transmission moves an integral number of records. Transmission proceeds from the first record until the *end of file.*

Binary Input/Output

The **readbin** and **writebin** statements transmit data in a machine-dependent but swift manner. The statements are of the form

writebin(*unit* **,** *binary-output-list* **)**
readbin(*unit* **,** *binary-input-list* **)**

Each statement moves one unformatted record between storage and the device. The *unit* is an integer expression. A *binary-output-list* is an *iolist* (see below) without any format specifiers. A *binary-input-list* is an *iolist* without format specifiers in which each of the expressions is a variable name, array element, or structure member.

Formatted Input/Output

The **read** and **write** statements transmit data in the form of lines of characters. Each statement moves one or more records (lines). Numbers are translated into decimal notation. The exact form of the lines is determined by format specifications, whether provided explicitly in the statement or implicitly. The syntax of the statements is

write(*unit* **,** *formatted-output-list* **)**
read(*unit* **,** *formatted-input-list* **)**

The lists are of the same form as for binary I/O, except that the lists may include format specifications. If the *unit* is omitted, the standard input or output unit is used.

Iolists

An *iolist* specifies a set of values to be written or a set of variables into which values are to be read. An *iolist* is a list of one or more *ioexpressions* of the form

expression
{ *iolist* }
do-specification { *iolist* }

For formatted I/O, an *ioexpression* may also have the forms

ioexpression : *format-specifier*
: *format-specifier*

A *do-specification* looks just like a **do** statement, and has a similar effect: the values in the braces are transmitted repeatedly until the **do** execution is complete.

Formats

The following are permissible *format-specifiers*. The quantities w, d, and k must be integer constant expressions.

i(w)	integer with w digits
f(w,d)	floating point number of w characters, d of them to the right of the decimal point.
e(w,d)	floating point number of w characters, d of them to the right of the decimal point, with the exponent field marked with the letter **e**
l(w)	logical field of width w characters, the first of which is **t** or **f** (the rest are blank on output, ignored on input) standing for **true** and **false** respectively

c	character string of width equal to the length of the datum
c(*w***)**	character string of width *w*
s(*k***)**	skip *k* lines
x(*k***)**	skip *k* spaces
" ... "	use the characters inside the string as a Fortran format

If no format is specified for an item in a formatted input/output statement, a default form is chosen.

If an item in a list is an array name, then the entire array is transmitted as a sequence of elements, each with its own format. The elements are transmitted in column-major order, the same order used for array initializations.

Manipulation Statements

The three input/output statements

backspace(*unit***)**
rewind(*unit***)**
endfile(*unit***)**

look like ordinary procedure calls, but may be used either as statements or as integer expressions which yield non-zero if an error is detected. **backspace** causes the specified unit to back up, so that the next read will re-read the previous record, and the next write will over-write it. **rewind** moves the device to its beginning, so that the next input statement will read the first record. **endfile** causes the file to be marked so that the record most recently written will be the last record on the file, and any attempt to read past is an error.

PROCEDURES

Procedures are the basic unit of an EFL program, and provide the means of segmenting a program into separately compilable and named parts.

Procedures Statement

Each procedure begins with a statement of one of the forms

procedure
attributes **procedure** *procedurename*
attributes **procedure** *procedurename* ( )
attributes **procedure** *procedurename* ([[*name*]])

The first case specifies the main procedure, where execution begins. In the two other cases, the *attributes* may specify precision and type, or they may be omitted entirely. The precision and type of the procedure may be declared in an ordinary declaration statement. If no type is declared, then the procedure is called a *subroutine* and no value may be returned for it. Otherwise, the procedure is a function and a value of the declared type is returned for each call. Each *name* inside the parentheses in the last form above is called a *formal argument* of the procedure.

End Statement

Each procedure terminates with a statement

end

Argument Association

When a procedure is invoked, the actual arguments are evaluated. If an actual argument is the name of a variable, an array element, or a structure member, that entity becomes associated with the formal argument, and the procedure may reference the values in the object, and assign to it. Otherwise, the value of the actual is associated with the formal argument, but the procedure may not attempt to change the value of that formal argument.

If the value of one of the arguments is changed in the procedure, it is not permitted that the corresponding actual argument be associated with another formal argument or with a **common** element that is referenced in the procedure.

Execution and Return Values

After actual and formal arguments have been associated, control passes to the first executable statement of the procedure. Control returns to the invoker either when the **end** statement of the procedure is reached or when a **return** statement is executed. If the procedure is a function (has a declared type), and a **return**(*value*) is executed, the value is coerced to the correct type and precision and returned.

Known Functions

A number of functions are known to EFL, and need not be declared. The compiler knows the types of these functions. Some of them are *generic;* i.e., they name a family of functions that differ in the types of their arguments and return values. The compiler chooses which element of the set to invoke based upon the attributes of the actual arguments.

Minimum and Maximum Functions

The generic functions are **min** and **max.** The **min** calls return the value of their smallest argument; the **max** calls return the value of their largest argument. These are the only functions that may take different numbers of arguments in different calls. If any of the arguments are **long real** then the result is **long real.** Otherwise, if any of the arguments are **real** then the result is **real;** otherwise all the arguments and the result must be **integer.** Examples are

```
min(5, x, −3.20)
max(i, z)
```

Absolute Value

The **abs** function is a generic function that returns the magnitude of its argument. For integer and real arguments the type of the result is identical to the type of the argument; for complex arguments the type of the result is the real of the same precision.

Elementary Functions

The following generic functions take arguments of **real, long real,** or **complex** type and return a result of the same type:

sin	**sine function**
cos	**cosine function**
exp	**exponential function (e^x).**
log	**natural (base e) logarithm**
log10	**common (base 10) logarithm**
sqrt	**square root function ($\sqrt{x}$).**

In addition, the following functions accept only **real** or **long real** arguments:

atan	$atan(x)=\tan^{-1}x$
atan2	$atan2(x,y)=\tan^{-1}\frac{x}{y}$

Other Generic Functions

The **sign** functions takes two arguments of identical type; **sign**$(x,y) = sgn(y)\,|x|$. The **mod** function yields the remainder of its first argument when divided by its second. These functions accept integer and real arguments.

ATAVISMS

Certain facilities are included in the EFL language to ease the conversion of old Fortran or Ratfor programs to EFL.

Escape Lines

In order to make use of nonstandard features of the local Fortran compiler, it is occasionally necessary to pass a particular line through to the EFL compiler output. A line that begins with a percent sign ("%") is copied through to the output, with the percent sign removed but no other change. Inside of a procedure, each escape line is treated as an executable statement. If a sequence of lines constitute a continued Fortran statement, they should be enclosed in braces.

Call Statement

A subroutine call may be preceded by the keyword **call.**

```
call joe
call work(17)
```

Obsolete Keywords

The following keywords are recognized as synonyms of EFL keywords:

Fortran	EFL
double precision	**long real**
function	**procedure**
subroutine	**procedure** *(untyped)*

Numeric Labels

Standard statement labels are identifiers. A numeric (positive integer constant) label is also permitted; the colon is optional following a numeric label.

Implicit Declarations

If a name is used but does not appear in a declaration, the EFL compiler gives a warning and assumes a declaration for it. If it is used in the context of a procedure invocation, it is assumed to be a procedure name; otherwise it is assumed to be a local variable defined at nesting level 1 in the current procedure. The assumed type is determined by the first letter of the name. The association of letters and types may be given in an **implicit** statement, with syntax

implicit (*letter-list*) *type*

where a *letter-list* is a list of individual letters or ranges (pair of letters separated by a minus sign). If no **implicit** statement appears, the following rules are assumed:

```
implicit (a–h, o–z) real
implicit (i–n) integer
```

Computed Goto

Fortran contains an indexed multi-way branch; this facility may be used in EFL by the computed GOTO:

goto ([[*label*]]), *expression*

The expression must be of type integer and be positive but be no larger than the number of labels in the list. Control is passed to the statement marked by the label whose position in the list is equal to the expression.

Goto Statement

In unconditional and computed **goto** statements, it is permissible to separate the **go** and **to** words, as in

go to xyz

Dot Names

Fortran uses a restricted character set, and represents certain operators by multi-character sequences. There is an option (**dots=on**; see " COMPILER OPTIONS") which forces the compiler to recognize the forms in the second column below:

<	.lt.
<=	.le.
>	.gt.
>=	.ge.
==	.eq.
~=	.ne.
&	.and.
|	.or.
&&	.andand.
||	.oror.
~	.not.
true	.true.
false	.false.

In this mode, no structure element may be named **lt, le,** etc. The readable forms in the left column are always recognized.

Complex Constants

A complex constant may be written as a parenthesized list of real quantities, such as

(1.5, 3.0)

The preferred notation is by a type coercion,

complex(1.5, 3.0)

Function Values

The preferred way to return a value from a function in EFL is the **return(***value***)** construct. However, the name of the function acts as a variable to which values may be assigned; an ordinary **return** statement returns the last value assigned to that name as the function value.

Equivalence

A statement of the form

equivalence v_1, v_2, ..., v_n

declares that each of the v_i starts at the same memory location. Each of the v_i may be a variable name, array element name, or structure member.

Minimum and Maximum Functions

There are a number of non-generic functions in this category, which differ in the required types of the arguments and the type of the return value. They may also have variable numbers of arguments, but all the arguments must have the same type.

Function	*Argument Type*	*Result Type*
amin0	**integer**	**real**
amin1	**real**	**real**
min0	**integer**	**integer**
min1	**real**	**integer**
dmin1	**long real**	**long real**
amax0	**integer**	**real**
amax1	**real**	**real**
max0	**integer**	**integer**
max1	**real**	**integer**
dmax1	**long real**	**long real**

COMPILER OPTIONS

A number of options can be used to control the output and to tailor it for various compilers and systems. The defaults chosen are conservative, but it is sometimes necessary to change the output to match peculiarities of the target environment.

Options are set with statements of the form

option [[*opt*]]

where each *opt* is of one of the forms

optionname
optionname = *optionvalue*

The *optionvalue* is either a constant (numeric or string) or a name associated with that option. The two names **yes** and **no** apply to a number of options.

Default Options

Each option has a default setting. It is possible to change the whole set of defaults to those appropriate for a particular environment by using the **system** option. At present, the only valid values are **system=unix** and **system=gcos.**

Input Language Options

The **dots** option determines whether the compiler recognizes **.lt.** and similar forms. The default setting is **no.**

Input/Output Error Handling

The **ioerror** option can be given three values: **none** means that none of the I/O statements may be used in expressions, since there is no way to detect errors. The implementation of the **ibm** form uses ERR= and END= clauses. The implementation of the **fortran77** form uses IOSTAT= clauses.

Continuation Conventions

By default, continued Fortran statements are indicated by a character in column 6 (Standard Fortran). The option **continue=column1** puts an ampersand (**&**) in the first column of the continued lines instead.

Default Formats

If no format is specified for a datum in an iolist for a **read** or **write** statement, a default is provided. The default formats can be changed by setting certain options

Option	*Type*
iformat	integer
rformat	real
dformat	long real
zformat	complex
zdformat	long complex
lformat	logical

The associated value must be a Fortran format, such as

option rformat=f22.6

Alignments and Sizes

In order to implement **character** variables, structures, and the **sizeof** and **lengthof** operators, it is necessary to know how much space various Fortran data types require, and what boundary alignment properties they demand. The relevant options are

Fortran Type	*Size Option*	*Alignment Option*
integer	**isize**	**ialign**
real	**rsize**	**ralign**
long real	**dsize**	**dalign**
complex	**zsize**	**zalign**
logical	**lsize**	**lalign**

The sizes are given in terms of an arbitrary unit; the alignment is given in the same units. The option **charperint** gives the number of characters per **integer** variable.

Default Input/Output Units

The options **ftnin** and **ftnout** are the numbers of the standard input and output units. The default values are **ftnin=5** and **ftnout=6.**

Miscellaneous Output Control Options

Each Fortran procedure generated by the compiler will be preceded by the value of the **procheader** option.

No Hollerith strings will be passed as subroutine arguments if **hollincall=no** is specified.

The Fortran statement numbers normally start at 1 and increase by 1. It is possible to change the increment value by using the **deltastno** option.

EXAMPLES

In order to show the flavor or programming in EFL, we present a few examples. They are short, but show some of the convenience of the language.

File Copying

The following short program copies the standard input to the standard output, provided that the input is a formatted file containing lines no longer than a hundred characters.

```
procedure  # main program
character(100) line

while( read( , line) == 0 )
                    write( , line)
end
```

Since **read** returns zero until the end of file (or a read error), this program keeps reading and writing until the input is exhausted.

Matrix Multiplication

The following procedure multiplies the $m \times n$ matrix a by the $n \times p$ matrix b to give the $m \times p$ matrix c. The calculation obeys the formula $c_{ij} = \sum a_{ik} b_{kj}$.

```
procedure matmul(a,b,c, m,n,p)
integer i, j, k, m, n, p
long real a(m,n), b(n,p), c(m,p)

do i = 1,m
do j = 1,p
        {
        c(i,j) = 0
        do k = 1,n
                c(i,j) += a(i,k) * b(k,j)
        }
end
```

Searching a Linked List

Assume we have a list of pairs of numbers (x,y). The list is stored as a linked list sorted in ascending order of x values. The following procedure searches this list for a particular value of x and returns the corresponding y value.

```
define LAST       0
define NOTFOUND          -1

integer procedure val(list, first, x)

#  list is an array of structures.
#  Each structure contains a thread index value,
#  an x, and a y value.
struct
        {
        integer nextindex
        integer x, y
        } list(*)
integer first, p, arg

for(p = first , p~=LAST && list(p).x<=x ,
   p = list(p).nextindex)
        if(list(p).x == x)
                return( list(p).y )

return(NOTFOUND)
end
```

The search is a single **for** loop that begins with the head of the list and examines items until either the list is exhausted (p==LAST) or until it is known that the specified value is not on the list (list(p).x > x). The two tests in the conjunction must be performed in the specified order to avoid using an invalid subscript in the **list(p)** reference. Therefore, the **&&** operator is used. The next element in the chain is found by the iteration statement **p=list(p).nextindex**.

Walking a Tree

As an example of a more complicated problem, let us imagine we have an expression tree stored in a common area, and that we want to print out an infix form of the tree. Each node is either a leaf (containing a numeric value) or it is a binary operator, pointing to a

left and a right descendant. In a recursive language, such a tree walk would be implement by the following simple pseudocode:

```
if this node is a leaf
        print its value
otherwise
        print a left parenthesis
        print the left node
        print the operator
        print the right node
        print a right parenthesis
```

In a nonrecursive language like EFL, it is necessary to maintain an explicit stack to keep track of the current state of the computation. The following procedure calls a procedure **outch** to print a single character and a procedure **outval** to print a value.

```
procedure walk(first)       # print an expression tree

integer first           # index of root node
integer currentnode
integer stackdepth
common(nodes) struct
        {
        character(1) op
        integer leftp, rightp
        real val
        } tree(100)     # array of structures

struct
        {
        integer nextstate
        integer nodep
        } stackframe(100)

define NODEtree(currentnode)
define STACK        stackframe(stackdepth)

#  nextstate values
define DOWN         1
define LEFT 2
define RIGHT        3
```

```
#  initialize stack with root node
stackdepth = 1
STACK.nextstate = DOWN
STACK.nodep = first

while( stackdepth > 0 )
        {
        currentnode = STACK.nodep
        select(STACK.nextstate)
                {
                case DOWN:
                        if(NODE.op == " ")  # a leaf
                                {
                                outval( NODE.val )
                                stackdepth -= 1
                                }
                        else    { # a binary operator node
                                outch( "(" )
                                STACK.nextstate = LEFT
                                stackdepth += 1
                                STACK.nextstate = DOWN
                                STACK.nodep = NODE.leftp
                                }

                case LEFT:
                        outch( NODE.op )
                        STACK.nextstate = RIGHT
                        stackdepth += 1
                        STACK.nextstate = DOWN
                        STACK.nodep = NODE.rightp

                case RIGHT:
                        outch( ")" )
                        stackdepth -= 1
                }
        }
end
```

PORTABILITY

One of the major goals of the EFL language is to make it easy to write portable programs. The output of the EFL compiler is intended to be acceptable to any Standard Fortran compiler (unless the **fortran77** option is specified).

Primitives

Certain EFL operations cannot be implemented in portable Fortran, so a few machine-dependent procedures must be provided in each environment.

Character String Copying

The subroutine **eflasc** is called to copy one character string to another. If the target string is shorter than the source, the final characters are not copied. If the target string is longer, its end is padded with blanks. The calling sequence is

```
subroutine eflasc(a, la, b, lb)
integer a(*), la, b(*), lb
```

and it must copy the first **lb** characters from **b** to the first **la** characters of **a.**

Character String Comparisons

The function **eflcmc** is invoked to determine the order of two character strings. The declaration is

```
integer function eflcmc(a, la, b, lb)
integer a(*), la, b(*), lb
```

The function returns a negative value if the string **a** of length **la** precedes the string **b** of length **lb.** It returns zero if the strings are equal, and a positive value otherwise. If the strings are of differing length, the comparison is carried out as if the end of the shorter string were padded with blanks.

DIFFERENCES BETWEEN RATFOR AND EFL

There are a number of differences between Ratfor and EFL, since EFL is a defined language while Ratfor is the union of the special control structures and the language accepted by the underlying Fortran compiler. Ratfor running over Standard Fortran is almost a subset of EFL. Most of the features described in the "ATAVISMS" are present to ease the conversion of Ratfor programs to EFL.

There are a few incompatibilities: The syntax of the **for** statement is slightly different in the two languages: the three clauses are separated by semicolons in Ratfor, but by commas in EFL. (The initial and iteration statements may be compound statements in EFL because of this change). The input/output syntax is quite different in the two languages, and there is no FORMAT statement in EFL. There are no ASSIGN or assigned GOTO statements in EFL.

The major linguistic additions are character data, factored declaration syntax, block structure, assignment and sequential test operators, generic functions, and data structures. EFL permits more general forms for expressions, and provides a more uniform syntax. (One need not worry about the Fortran/Ratfor restrictions on subscript or DO expression forms, for example.)

COMPILER

Current Version

The current version of the EFL compiler is a two-pass translator written in portable C. It implements all of the features of the language described above except for **long complex** numbers.

Diagnostics

The EFL compiler diagnoses all syntax errors. It gives the line and file name (if known) on which the error was detected. Warnings are given for variables that are used but not explicitly declared.

Quality of Fortran Produced

The Fortran produced by EFL is quite clean and readable. To the extent possible, the variable names that appear in the EFL program are used in the Fortran code. The bodies of loops and test constructs are indented. Statement numbers are consecutive. Few unneeded GOTO and CONTINUE statements are used. It is considered a compiler bug if incorrect Fortran is produced (except for escaped lines). The following is the Fortran procedure produced by the EFL compiler for the matrix multiplication example (See " EXAMPLES" .)

```
      subroutine matmul(a, b, c, m, n, p)
      integer m, n, p
      double precision a(m, n), b(n, p), c(m, p)
      integer i, j, k
      do  3 i = 1, m
          do  2 j = 1, p
              c(i, j) = 0
              do  1 k = 1, n
                  c(i, j) = c(i, j)+a(i, k)*b(k, j)
1             continue
2             continue
3         continue
      end
```

The following is the procedure for the tree walk:

```
      subroutine walk(first)
      integer first
      common /nodes/ tree
      integer tree(4, 100)
      real tree1(4, 100)
      integer staame(2, 100), stapth, curode
      integer const1(1)
      equivalence (tree(1,1), tree1(1,1))
      data const1(1)/4h     /
c print out an expression tree
c index of root node
c array of structures
c   nextstate values
c   initialize stack with root node
      stapth = 1
      staame(1, stapth) = 1
      staame(2, stapth) = first
   1  if (stapth .le. 0) goto  9
         curode = staame(2, stapth)
         goto  7
   2        if (tree(1, curode) .ne. const1(1)) goto 3
               call outval(tree1(4, curode))
c a leaf
               stapth = stapth-1
               goto  4
   3           call outch(1h()
c a binary operator node
               staame(1, stapth) = 2
               stapth = stapth+1
               staame(1, stapth) = 1
               staame(2, stapth) = tree(2, curode)
   4        goto  8
   5        call outch(tree(1, curode))
            staame(1, stapth) = 3
            stapth = stapth+1
            staame(1, stapth) = 1
            staame(2, stapth) = tree(3, curode)
            goto  8
   6        call outch(1h))
            stapth = stapth-1
            goto  8
```

```
7           if (staame(1, stapth) .eq. 3) goto  6
            if (staame(1, stapth) .eq. 2) goto  5
            if (staame(1, stapth) .eq. 1) goto  2
8       continue
        goto  1
9   continue
    end
```

CONSTRAINTS ON EFL

Although Fortran can be used to simulate any finite computation, there are realistic limits on the generality of a language that can be translated into Fortran. The design of EFL was constrained by the implementation strategy. Certain of the restrictions are petty (six character external names), but others are sweeping (lack of pointer variables). The following paragraphs describe the major limitations imposed by Fortran.

External Names

External names (procedure and COMMON block names) must be no longer than six characters in Fortran. Further, an external name is global to the entire program. Therefore, EFL can support block structure within a procedure, but it can have only one level of external name if the EFL procedures are to be compilable separately, as are Fortran procedures.

Procedure Interface

The Fortran standards, in effect, permit arguments to be passed between Fortran procedures either by reference or by copy-in/copy-out. This indeterminacy of specification shows through into EFL. A program that depends on the method of argument transmission is illegal in either language.

There are no procedure-valued variables in Fortran: a procedure name may only be passed as an argument or be invoked; it cannot be stored. Fortran (and EFL) would be noticeably simpler if a procedure variable mechanism were available.

Pointers

The most grievous problem with Fortran is its lack of a pointer-like data type. The implementation of the compiler would have been far easier if certain hard cases could have been handled by pointers. Further, the language could have been simplified considerably if pointers were accessible in Fortran. (There are several ways of simulating pointers by using subscripts, but they founder on the problems of external variables and initialization.)

Recursion

Fortran procedures are not recursive, so it was not practical to permit EFL procedures to be recursive. (Recursive procedures with arguments can be simulated only with great pain.)

Storage Allocation

The definition of Fortran does not specify the lifetime of variables. It would be possible but cumbersome to implement stack or heap storage disciplines by using COMMON blocks.

Chapter 15

THE CURSES AND TERMINFO PACKAGE

INTRODUCTION

This chapter is an introduction to **curses**(3X) and **terminfo**(4). It is intended for the programmer who must write a screen-oriented program using the **curses** package. Several example programs are discussed. The example programs can be found in Chapter 13. This chapter also documents each **curses** function. It is intended as a reference.

For **curses** to be able to produce terminal dependent output, it has to know what kind of terminal you have. The UNIX system convention for this is to put the name of the terminal in the variable TERM in the environment. Thus, a user on a DEC VT100 would set `TERM=vt100` when logging in. Curses uses this convention.

Output

A program using **curses** always starts by calling `initscr()`. (See Figure 12-1.) Other modes can then be set as needed by the program. Possible modes include `cbreak()`, and `idlok(stdscr, TRUE)`. These modes will be explained later. During the execution of the program, output to the screen is done with routines such as `addch(ch)` and `printw(fmt,args)`. (These routines behave just like `putchar` and `printf` except that they go through **curses**.) The cursor can be moved with the call `move(row,col)`. These routines only output to a data structure called a *window*, not to the actual screen. A window is a representation of a CRT screen, containing such things as an array of characters to be displayed on the screen, a cursor, a current set of video attributes, and various modes and options. You don't need to worry about windows unless you use more than one of them, except to realize that a window is buffering your requests to output to the screen.

C-15

To send all accumulated output, it is necessary to call `refresh()`.

(This can be thought of as a `flush`.) Finally, before the program exits, it should call `endwin()`, which restores all terminal settings and positions the cursor at the bottom of the screen.

```
#include <curses.h>
...
  initscr(); /* Initialization */

  cbreak();/* Various optional mode settings */
  nonl();
  noecho();
...
  while (!done) {/* Main body of program */
    ...
    /* Sample calls to draw on screen */
    move(row, col);
    addch(ch);
    printw("Formatted print with value %d\n", value);
    ...
    /* Flush output */
    refresh();
    ...
  }

  endwin();/* Clean up */
  exit(0);
```

Figure 12-1 - Framework of a Curses Program

See the program **scatter** in Chapter 13 for an example program. This program reads a file, and displays the file in a random order on the screen. Some programs assume all screens are 24 lines by 80 columns. It is important to understand that many are not. The variables `LINES` and `COLS` are defined by `initscr` with the current screen size. Programs should use them instead of assuming a 24x80 screen.

No output to the terminal actually happens until `refresh` is called. Instead, routines such as `move` and `addch` draw on a window data structure called `stdscr` (standard screen). **Curses** always keeps track of what is on the physical screen, as well as what is in `stdscr`.

When `refresh` is called, **curses** compares the two screen images and sends a stream of characters to the terminal that will turn the current screen into what is desired. Curses considers many different ways to do this, taking into account the various capabilities of the terminal, and similarities between what is on the screen and what is desired. It usually outputs as few characters as is possible. This function is called *cursor optimization* and is the source of the name of the **curses** package.

NOTE: Due to the hardware scrolling of terminals, writing to the lower righthand character position is impossible.

Input

Curses can do more than just draw on the screen. Functions are also provided for input from the keyboard. The primary function is `getch()` which waits for the user to type a character on the keyboard, and then returns that character. This function is like `getchar` except that it goes through **curses**. Its use is recommended for programs using the `cbreak()` or `noecho()` options, since several terminal or system dependent options become available that are not possible with `getchar`.

Options available with `getch` include `keypad` which allows extra keys such as arrow keys, function keys, and other special keys that transmit escape sequences, to be treated as just another key. (The values returned for these keys are listed below.) `KEY_LEFT` in `curses.h`. The values for these keys are over octal 400, so they should be stored in a variable larger than a **char**.) *nodelay* mode causes the value -1 to be returned if there is no input waiting. Normally, `getch` will wait until a character is typed. Finally, the routine `getstr(str)` can be called, allowing input of an entire line, up to a newline. This routine handles echoing and the erase and kill characters of the user. Examples of the use of these options are in later example programs.

The following function keys might be returned by `getch` if `keypad` has been enabled. Note that not all of these are currently supported, due to lack of definitions in **terminfo** or the terminal not transmitting a unique code when the key is pressed.

Name	*Value*	*Key name*
KEY_BREAK	0401	Break key (unreliable)
KEY_DOWN	0402	The four arrow keys ...
KEY_UP	0403	
KEY_LEFT	0404	
KEY_RIGHT	0405	...
KEY_HOME	0406	Home key (upward+left arrow)
KEY_BACKSPACE	0407	Backspace (unreliable)
KEY_F0	0410	Function keys. Space for 64 keys is reserved.
KEY_F(n)	(KEY_F0+(n))	Formula for fn.
KEY_DL	0510	Delete line
KEY_IL	0511	Insert line
KEY_DC	0512	Delete character
KEY_IC	0513	Insert char or enter insert mode
KEY_EIC	0514	Exit insert char mode
KEY_CLEAR	0515	Clear screen
KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll 1 line forward
KEY_SR	0521	Scroll 1 line backwards (reverse)
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab
KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send (unreliable)
KEY_SRESET	0530	Soft (partial) reset (unreliable)
KEY_RESET	0531	Reset or hard reset (unreliable)
KEY_PRINT	0532	Print or copy
KEY_LL	0533	Home down or bottom (lower left)

See the program **show** in Chapter 13 for an example use of `getch`. **Show** pages through a file, showing one screen full each time the user presses the space bar. By creating an input file for **show** made

up of 24 line pages, each segment varying slightly from the previous page, nearly any exercise for **curses** can be created. Such input files are called *show scripts*.

In the **show** program, `cbreak` is called so that the user can press the space bar without having to hit return. The `noecho` function is called to prevent the space from echoing in the middle of a `refresh`, messing up the screen. The `nonl` function is called to enable more screen optimization. The `idlok` function is called to allow insert and delete line, since many show scripts are constructed to duplicate bugs caused by that feature. The `clrtoeol` and `clrtobot` functions clear from the cursor to the end of the line and screen, respectively.

Highlighting

The function `addch` always draws two things on a window. In addition to the character itself, a set of *attributes* is associated with the character. These attributes cover various forms of highlighting of the character. For example, the character can be put in reverse video, bold, or be underlined. You can think of the attributes as the color of the ink used to draw the character.

A window always has a set of *current attributes* associated with it. The current attributes are associated with each character as it is written to the window. The current attributes can be changed with a call to `attrset(attrs)`. (Think of this as dipping the window's pen in a particular color ink.) The names of the attributes are `A_STANDOUT`, `A_REVERSE`, `A_BOLD`, `A_DIM`, `A_INVIS`, and `A_UNDERLINE`. For example, to put a word in bold, the code in Figure 12-2 might be used. The word "boldface" will be shown in bold.

```
printw("A word in ");
attrset(A_BOLD);
printw("boldface");
attrset(0);
printw(" really stands out.\n");
...
refresh();
```

Figure 12-2 - Use of attributes.

Not all terminals are capable of displaying all attributes. If a particular terminal cannot display a requested attribute, **curses** will attempt to find a substitute attribute. If none is possible, the attribute is ignored.

One particular attribute is called *standout*. This attribute is used to make text attract the attention of the user. The particular hardware attribute used for standout varies from terminal to terminal, and is chosen to be the most visually pleasing attribute the terminal has. Standout is typically implemented as reverse video or bold. Many programs don't really need a specific attribute, such as bold or inverse video, but instead just need to highlight some text. For such applications, the `A_STANDOUT` attribute is recommended. Two convenient functions, `standout()` and `standend()` turn on and off this attribute.

Attributes can be turned on in combination. Thus, to turn on blinking bold text, use `attrset(A_BLINK|A_BOLD)`. Individual attributes can be turned on and off with `attron` and `attroff` without affecting other attributes.

For an example program using attributes, see **highlight**. The program takes a text file as input and allows embedded escape sequences to control attributes. In this example program, \U turns on underlining, \B turns on bold, and \N restores normal text. Note the initial call to `scrollok`. This allows the terminal to scroll if the file is longer than one screen. When an attempt is made to draw

past the bottom of the screen, **curses** will automatically scroll the terminal up a line and call `refresh`.

Highlight comes about as close to being a filter as is possible with **curses**. It is not a true filter, because **curses** must "take over" the CRT screen. In order to determine how to update the screen, it must know what is on the screen at all times. This requires **curses** to clear the screen in the first call to `refresh`, and to know the cursor position and screen contents at all times.

Multiple Windows

A window is a data structure representing all or part of the CRT screen. It has room for a two dimensional array of characters, attributes for each character (a total of 16 bits per character: 7 for text and 9 for attributes) a cursor, a set of current attributes, and a number of flags. Curses provides a full screen window, called `stdscr`, and a set of functions that use `stdscr`. Another window is provided called `curscr`, representing the physical screen.

It is important to understand that a window is only a data structure. Use of more than one window does not imply use of more than one terminal, nor does it involve more than one process. A window is merely an object which can be copied to all or part of the terminal screen. The current implementation of **curses** does not allow windows which are bigger than the screen.

The programmer can create additional windows with the function `newwin(lines, cols, begin_row, begin_col)` will return a pointer to a newly created window. The window will be `lines` by `cols`, and the upper left corner of the window will be at screen position `(begin_row, begin_col)`. All operations that affect `stdscr` have corresponding functions that affect an arbitrary named window. Generally, these functions have names formed by putting a "w" on the front of the `stdscr` function, and the window name is added as the first parameter. Thus, `waddch(mywin, c)` would write the character `c` to window `mywin`. The `wrefresh(win)` function is used to flush the contents of a window to the screen.

Windows are useful for maintaining several different screen images, and alternating the user among them. Also, it is possible to

C-15

subdivide the screen into several windows, refreshing each of them as desired. When windows overlap, the contents of the screen will be the more recently refreshed window.

In all cases, the non-w version of the function calls the w version of the function, using `stdscr` as the additional argument. Thus, a call to `addch(c)` results in a call to `waddch(stdscr, c)`.

The program **window** is an example of the use of multiple windows. The main display is kept in `stdscr`. When the user temporarily wants to put something else on the screen, a new window is created covering part of the screen. A call to `wrefresh` on that window causes the window to be written over `stdscr` on the screen. Calling `refresh` on `stdscr` results in the original window being redrawn on the screen. Note the calls to `touchwin` before writing out an overlapping window. These are necessary to defeat an optimization in **curses**. If you have trouble refreshing a new window which overlaps an old window, it may be necessary to call `touchwin` on the new window to get it completely written out.

For convenience, a set of "move" functions are also provided for most of the common functions. These result in a call to `move` before the other function. For example, `mvaddch(row, col, c)` is the same as `move(row, col); addch(c)`. Combinations, e.g. `mvwaddch(row, col, win, c)` also exist.

Multiple Terminals

Curses can produce output on more than one terminal at once. This is useful for single process programs that access a common database, such as multi-player games. Output to multiple terminals is a difficult business, and **curses** does not solve all the problems for the programmer. It is the responsibility of the program to determine the file name of each terminal line, and what kind of terminal is on each of those lines. The standard method, checking `$TERM` in the environment, does not work, since each process can only examine its own environment. Another problem that must be solved is that of multiple programs reading from one line. This situation produces a race condition and should be avoided. Nonetheless, a program wishing to take over another terminal cannot just shut off whatever program is currently running on that line. (Usually, security reasons

would also make this inappropriate. However, for some applications, such as an inter-terminal communication program, or a program that takes over unused tty lines, it would be appropriate.) A typical solution requires the user logged in on each line to run a program that notifies the master program that the user is interested in joining the master program, telling it the notification program's process id, the name of the tty line and the type of terminal being used. Then the program goes to sleep until the master program finishes. When done, the master program wakes up the notification program, and all programs exit.

Curses handles multiple terminals by always having a *current terminal*. All function calls always affect the current terminal. The master program should set up each terminal, saving a reference to the terminals in its own variables. When it wishes to affect a terminal, it should set the current terminal as desired, and then call ordinary **curses** routines.

References to terminals have type `struct screen *`. A new terminal is initialized by calling `newterm(type, fd)`. `newterm` returns a screen reference to the terminal being set up. `type` is a character string, naming the kind of terminal being used. `fd` is a stdio file descriptor to be used for input and output to the terminal. (If only output is needed, the file can be open for output only.) This call replaces the normal call to `initscr`, which calls `newterm(getenv(''TERM''), stdout)`.

To change the current terminal, call "`set_term(` *sp*`)`" where `sp` is the screen reference to be made current. `set_term` returns a reference to the previous terminal.

It is important to realize that each terminal has its own set of windows and options. Each terminal must be initialized separately with `newterm`. Options such as `cbreak` and `noecho` must be set separately for each terminal. The functions `endwin` and `refresh` must be called separately for each terminal. See Figure 12-3 for a typical scenario to output a message to each terminal.

```
for (i=0; i<nterm; i++) {
    set_term(terms[i]);
    mvaddstr(0, 0, "Important message");
    refresh();
}
```

Figure 12-3 - Sending a message to several terminals

See the sample program **two** for a full example. This program pages through a file, showing one page to the first terminal and the next page to the second terminal. It then waits for a space to be typed on either terminal, and shows the next page to the terminal typing the space. Each terminal has to be separately put into nodelay mode. Since no standard multiplexor is available in current versions of the UNIX system, it is necessary to either busy wait, or call `sleep(1);`, between each check for keyboard input. This program sleeps for a second between checks.

The **two** program is just a simple example of two terminal **curses**. It does not handle notification, as described above, instead it requires the name and type of the second terminal on the command line. As written, the command `sleep 100000` must be typed on the second terminal to put it to sleep while the program runs, and the first user must have both read and write permission on the second terminal.

Low Level Terminfo Usage

Some programs need to use lower level primitives than those offered by **curses**. For such programs, the *terminfo level* interface is offered. This interface does not manage your CRT screen, but rather gives you access to strings and capabilities which you can use yourself to manipulate the terminal.

Programmers are discouraged from using this level. Whenever possible, the higher level **curses** routines should be used. This will make your program more portable to other UNIX systems and to a wider class of terminals. Curses takes care of all the glitches and

misfeatures present in physical terminals, but at the terminfo level you must deal with them yourself. Also, it cannot be guaranteed that this part of the interface will not change or be upward compatible with previous releases.

There are two circumstances when it is proper to use terminfo. The first is when you are writing a special purpose tool that sends a special purpose string to the terminal, such as programming a function key, setting tab stops, sending output to a printer port, or dealing with the status line. The second situation is when writing a filter. A typical filter does one transformation on the input stream without clearing the screen or addressing the cursor. If this transformation is terminal dependent and clearing the screen is inappropriate, use of terminfo is indicated.

A program writing at the terminfo level uses the framework shown in Figure 12-4.

```
#include <curses.h>
#include <term.h>
...
    setupterm(0, 1, 0);
    ...
    putp(clear_screen);
    ...
    reset_shell_mode();
    exit(0);
```

Figure 12-4 - Terminfo level framework

Initialization is done by calling `setupterm`. Passing the values 0, 1, and 0 invoke reasonable defaults. If `setupterm` can't figure out what kind of terminal you are on, it will print an error message and exit. The program should call `reset_shell_mode` before it exits.

Global variables with names like `clear_screen` and `cursor_address` are defined by the call to `setupterm`. They can

be output using `putp`, or also using `tputs`, which allows the programmer more control. These strings *should not* be directly output to the terminal using `printf` since they contain padding information. A program that directly outputs strings will fail on terminals that require padding, or that use the xon/xoff flow control protocol.

In the terminfo level, the higher level routines described previously are not available. It is up to the programmer to output whatever is needed. For a list of capabilities and a description of what they do, see **terminfo**(4).

The example program **termhl** shows simple use of terminfo. It is a version of **highlight** that uses terminfo instead of **curses**. This version can be used as a filter. The strings to enter bold and underline mode, and to turn off all attributes, are used.

This program is more complex than it need be in order to illustrate some properties of terminfo. The routine `vidattr` could have been used instead of directly outputting `enter_bold_mode`, `enter_underline_mode`, and `exit_attribute_mode`. In fact, the program would be more robust if it did since there are several ways to change video attribute modes. This program was written to illustrate typical use of terminfo.

The function `tputs(cap, affcnt, outc)` applies padding information. Some capabilities contain strings like $<20>, which means to pad for 20 milliseconds. `tputs` generates enough pad characters to delay for the appropriate time. The first parameter is the string capability to be output. The second is the number of lines affected by the capability. (Some capabilities may require padding that depends on the number of lines affected. For example, `insert_line` may have to copy all lines below the current line, and may require time proportional to the number of lines copied. By convention `affcnt` is 1 if no lines are affected. The value 1 is used, rather than 0, for safety, since `affcnt` is multiplied by the amount of time per item, and anything multiplied by 0 is 0.) The third parameter is a routine to be called with each character.

For many simple programs, `affcnt` is always 1 and `outc` always just calls `putchar`. For these programs, the routine `putp(cap)` is

a convenient abbreviation. **termhl** could be simplified by using `putp`.

Note also the special check for the `underline_char` capability. Some terminals, rather than having a code to start underlining and a code to stop underlining, have a code to underline the current character. **termhl** keeps track of the current mode, and if the current character is supposed to be underlined, will output `underline_char` if necessary. Low level details such as this are precisely why the **curses** level is recommended over the terminfo level. Curses takes care of terminals with different methods of underlining and other CRT functions. Programs at the terminfo level must handle such details themselves.

A **Larger Example**

For a final example, see the program **editor**. This program is a very simple screen editor, patterned after the **vi** editor. The program illustrates how to use **curses** to write a screen editor. This editor keeps the buffer in `stdscr` to keep the program simple - obviously a real screen editor would keep a separate data structure. Many simplifications have been made here - no provision is made for files of any length other than the size of the screen, for lines longer than the width of the screen, or for control characters in the file.

Several points about this program are worth making. The routine to write out the file illustrates the use of the `mvinch` function, which returns the character in a window at a given position. The data structure used here does not have a provision for keeping track of the number of characters in a line, or the number of lines in the file, so trailing blanks are eliminated when the file is written out.

The program uses built-in **curses** functions `insch`, `delch`, `insertln`, and `deleteln`. These functions behave much as the similar functions on intelligent terminals behave, inserting and deleting a character or line.

The command interpreter accepts not only ASCII characters, but also special keys. This is important - a good program will accept both. (Some editors are *modeless*, using nonprinting characters for commands. This is largely a matter of taste - the point being made

here is that both arrow keys and ordinary ASCII characters should be handled.) It is important to handle special keys because this makes it easier for a new user to learn to use your program if he can use the arrow keys, instead of having to memorize that "h" means left, "j" means down, "k" means up, and "l" means right. On the other hand, not all terminals have arrow keys, so your program will be usable on a larger class of terminals if there is an ASCII character which is a synonym for each special key. Also, experienced users dislike having to move their hands from the "home row" position to use special keys, since they can work faster with alphabetic keys.

Note the call to `mvaddstr` in the input routine. `addstr` is roughly like the C `fputs` function, which writes out a string of characters. Like `fputs`, `addstr` does not add a trailing newline. It is the same as a series of calls to `addch` using the characters in the string. `mvaddstr` is the mv version of `addstr`, which moves to the given location in the window before writing.

The control-L command illustrates a feature most programs using **curses** should add. Often some program beyond the control of **curses** has written something to the screen, or some line noise has messed up the screen beyond what **curses** can keep track of. In this case, the user usually types control-L, causing the screen to be cleared and redrawn. This is done with the call to `clearok(curscr)`, which sets a flag causing the next `refresh` to first clear the screen. Then `refresh` is called to force the redraw.

Note also the call to `flash()`, which flashes the screen if possible, and otherwise rings the bell. Flashing the screen is intended as a bell replacement, and is particularly useful if the bell bothers someone within earshot of the user. The routine `beep()` can be called when a real beep is desired. (If for some reason the terminal is unable to beep, but able to flash, a call to `beep` will flash the screen.)

Another important point is that the input command is terminated by control-D, not escape. It is very tempting to use escape as a command, since escape is one of the few special keys which is available on every keyboard. (Return and break are the only others.) However, using escape as a separate key introduces an ambiguity.

Most terminals use sequences of characters beginning with escape ("escape sequences") to control the terminal, and have special keys that send escape sequences to the computer. If the computer sees an escape coming from the terminal, it cannot tell for sure whether the user pushed the escape key, or whether a special key was pressed. Curses handles the ambiguity by waiting for up to one second. If another character is received during this second, and if that character might be the beginning of a special key, more input is read (waiting for up to one second for each character) until either a full special key is read, one second passes, or a character is received that could not have been generated by a special key. While this strategy works most of the time, it is not foolproof. It is possible for the user to press escape, then to type another key quickly, which causes **curses** to think a special key has been pressed. Also, there is a one second pause until the escape can be passed to the user program, resulting in slower response to the escape key. Many existing programs use escape as a fundamental command, which cannot be changed without infuriating a large class of users. Such programs cannot make use of special keys without dealing with this ambiguity, and at best must resort to a timeout solution. The moral is clear: when designing your program, avoid the escape key.

LIST OF ROUTINES

This section describes all the routines available to the programmer in the **curses** package. The routines are organized by function. For an alphabetical list, see **curses**(3X).

Structure

All programs using **curses** should include the file `<curses.h>`. This file defines several **curses** functions as macros, and defines several global variables and the datatype `WINDOW`. References to windows are always of type `WINDOW *`. Curses also defines `WINDOW *` constants `stdscr` (the standard screen, used as a default to routines expecting a window), and `curscr` (the current screen, used only for certain low level operations like clearing and redrawing a garbaged screen). Integer constants `LINES` and `COLS` are defined, containing the size of the screen. Constants `TRUE` and `FALSE` are defined, with values 1 and 0, respectively. Additional constants which are values returned from most **curses** functions are `ERR` and

OK. OK is returned if the function could be properly completed, and ERR is returned if there was some error, such as moving the cursor outside of a window.

The include file <curses.h> automatically includes <stdio.h> and an appropriate tty driver interface file, currently either <sgtty.h*> or <termio.h>. Including <stdio.h> again is harmless but wasteful, including <sgtty.h> again will usually result in a fatal error.

A program using **curses** should include the loader option -lcurses in the makefile. This is true for both the **terminfo** level and the **curses** level. The compilation flag -DMINICURSES can be included if you restrict your program to a small subset of **curses** concerned primarily with screen output and optimization. The routines possible with mini-curses are listed in "Mini-Curses" under "OPERATION DETAILS."

Initialization

These functions are called when initializing a program.

`initscr()`
The first function called should always be `initscr`. This will determine the terminal type and initialize **curses** data structures. `initscr` also arranges that the first call to `refresh` will clear the screen.

`endwin()`
A program should always call `endwin` before exiting. This function will restore tty modes, move the cursor to the lower left corner of the screen, reset the terminal into the proper non-visual mode, and tear down all appropriate data structures.

---

* The driver interface <sgtty.h> is a tty driver interface used in other versions of the UNIX system.

`newterm(type, fd)`
A program which outputs to more than one terminal should use `newterm` instead of `initscr`. `newterm` should be called once for each terminal. It returns a variable of type `SCREEN *` which should be saved as a reference to that terminal. The arguments are the type of the terminal (a string) and a stdio file descriptor (`FILE*`) for output to the terminal. The file descriptor should be open for both reading and writing if input from the terminal is desired. The program should also call `endwin` for each terminal being used (see `set_term` below). If an error occurs, the value `NULL` is returned.

`set_term(new)`
This function is used to switch to a different terminal. The screen reference `new` becomes the new current terminal. The previous terminal is returned by the function. All other calls affect only the current terminal.

`longname()`
This function returns a pointer to a static area containing a verbose description of the current terminal. It is defined only after a call to `initscr`, `newterm`, or `setupterm`.

Option Setting

These functions set options within **curses**. In each case, `win` is the window affected, and `bf` is a boolean flag with value `TRUE` or `FALSE` indicating whether to enable or disable the option. All options are initially `FALSE`. It is not necessary to turn these options off before calling `endwin`.

`clearok(win,bf)`
If set, the next call to `wrefresh` with this window will clear the screen and redraw the entire screen. If `win` is `curscr`, the next call to `wrefresh` with any window will cause the screen to be cleared. This is useful when the contents of the screen are uncertain, or in some cases for a more pleasing visual effect.

`idlok(win,bf)`
If enabled, **curses** will consider using the hardware insert/delete line feature of terminals so equipped. If disabled, **curses** will never use this feature. The insert/delete character feature is always

considered. Enable this option only if your application needs insert/delete line, for example, for a screen editor. It is disabled by default because insert/delete line tends to be visually annoying when used in applications where it isn't really needed. If insert/delete line cannot be used, **curses** will redraw the changed portions of all lines that do not match the desired line.

`keypad(win,bf)`
This option enables the keypad of the users terminal. If enabled, the user can press a function key (such as an arrow key) and `getch` will return a single value representing the function key. If disabled, **curses** will not treat function keys specially. If the keypad in the terminal can be turned on (made to transmit) and off (made to work locally), turning on this option will turn on the terminal keypad.

`leaveok(win,bf)`
Normally, the hardware cursor is left at the location of the window cursor being refreshed. This option allows the cursor to be left wherever the update happens to leave it. It is useful for applications where the cursor is not used, since it reduces the need for cursor motions. If possible, the cursor is made invisible when this option is enabled.

`meta(win,bf)`
If enabled, characters returned by `getch` are transmitted with all 8 bits, instead of stripping the highest bit. The value `OK` is returned if the request succeeded, the value `ERR` is returned if the terminal or system is not capable of 8-bit input.

Meta mode is useful for extending the non-text command set in applications where the terminal has a meta shift key. Curses takes whatever measures are necessary to arrange for 8-bit input. On other versions of UNIX systems, raw mode will be used. On our systems, the character size will be set to 8, parity checking disabled, and stripping of the 8th bit turned off.

Note that 8-bit input is a fragile mode. Many programs and networks only pass 7 bits. If any link in the chain from the terminal to the application program strips the 8th bit, 8-bit input is impossible.

`nodelay(win,bf)`
This option causes `getch` to be a non-blocking call. If no input is ready, `getch` will return -1. If disabled, `getch` will hang until a key is pressed.

`intrflush(win,bf)`
If this option is enabled when an interrupt key is pressed on the keyboard (interrupt, quit, suspend), all output in the tty driver queue will be flushed, giving the effect of faster response to the interrupt but causing **curses** to have the wrong idea of what is on the screen. Disabling the option prevents the flush. The default is for the option to be enabled. This option depends on support in the underlying teletype driver.

`typeahead(fd)`
Sets the file descriptor for typeahead check. **fd** should be an integer returned from `open` or `fileno`. Setting typeahead to -1 will disable typeahead check. By default, file descriptor 0 (stdin) is used. Typeahead is checked independently for each screen, and for multiple interactive terminals it should probably be set to the appropriate input for each screen. A call to `typeahead` always affects only the current screen.

`scrollok(win,bf)`
This option controls what happens when the cursor of a window is moved off the edge of the window, either from a newline on the bottom line, or typing the last character of the last line. If disabled, the cursor is left on the bottom line. If enabled, `wrefresh` is called on the window, and then the physical terminal and window are scrolled up one line. Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call `idlok`.

`setscrreg(t,b)`
`wsetscrreg(win,t,b)`
These functions allow the user to set a software scrolling region in a window `win` or `stdscr`. `t` and `b` are the line numbers of the top and bottom margin of the scrolling region. (Line 0 is the top line of the window.) If this option and **scrollok** are enabled, an attempt to move off the bottom margin line will cause all lines in the scrolling region to scroll up one line. Note that this has nothing to do with use of a physical scrolling region capability in the terminal, like that in

C-15

the VT100. Only the text of the window is scrolled. If **idlok** is enabled and the terminal has either a scrolling region or insert/delete line capability, they will probably be used by the output routines.

Terminal Mode Setting

These functions are used to set modes in the tty driver. The initial mode usually depends on the setting when the program was called: the initial modes documented here represent the normal situation.

`cbreak()`
`nocbreak()`
These two functions put the terminal into and out of **CBREAK** mode. In this mode, characters typed by the user are immediately available to the program. When out of this mode, the teletype driver will buffer characters typed until newline is typed. Interrupt and flow control characters are unaffected by this mode. Initially the terminal is not in CBREAK mode. Most interactive programs using **curses** will set this mode.

`echo()`
`noecho()`
These functions control whether characters typed by the user are echoed as typed. Initially, characters typed are echoed by the teletype driver. Authors of many interactive programs prefer to do their own echoing in a controlled area of the screen, or not to echo at all, so they disable echoing.

`nl()`
`nonl()`
These functions control whether newline is translated into carriage return and linefeed on output, and whether return is translated into newline on input. Initially, the translations do occur. By disabling these translations, **curses** is able to make better use of the linefeed capability, resulting in faster cursor motion.

`raw()`
`noraw()`
The terminal is placed into or out of raw mode. Raw mode is similar to cbreak mode in that characters typed are immediately passed

through to the user program. The differences are that in RAW mode, the interrupt, quit, and suspend characters are passed through uninterpreted instead of generating a signal. RAW mode also causes 8 bit input and output. The behavior of the BREAK key may be different on different systems.

`resetty()`
`savetty()`
These functions save and restore the state of the tty modes. `savetty` saves the current state in a buffer, `resetty` restores the state to what it was at the last call to `savetty`.

Window Manipulation

`newwin(num_lines, num_cols, beg_row, beg_col)`
Create a new window with the given number of lines and columns. The upper left corner of the window is at line `beg_row` column `beg_col`. If either `num_lines` or `num_cols` is zero, they will be defaulted to `LINES-beg_row` and `COLS-beg_col`. A new full-screen window is created by calling `newwin(0,0,0,0)`.

`newpad(num_lines, num_cols)`
Creates a new *pad* data structure. A pad is like a window, except that it is not restricted by the screen size, and is not associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (e.g. from scrolling or echoing of input) do not occur. It is not legal to call `refresh` with a pad as an argument, the routines `prefresh` or `pnoutrefresh` should be called instead. Note that these routines require additional parameters to specify the part of the pad to be displayed and the location on the screen to be used for display.

`subwin(orig, num_lines, num_cols, begy, begx)`
Create a new window with the given number of lines and columns. The window is at position (*begy, begx*) on the screen. (It is relative to the screen, not `orig`.) The window is made in the middle of the window `orig`, so that changes made to one window will affect both windows. When using this function, often it will be necessary to call `touchwin` before calling `wrefresh`.

`delwin(win)`
Deletes the named window, freeing up all memory associated with it. In the case of overlapping windows, subwindows should be deleted before the main window.

`mvwin(win, br, bc)`
Move the window so that the upper left corner will be at position `(br, bc)`. If the move would cause the window to be off the screen, it is an error and the window is not moved.

`touchwin(win)`
Throw away all optimization information about which parts of the window have been touched, by pretending the entire window has been drawn on. This is sometimes necessary when using overlapping windows, since a change to one window will affect the other window, but the records of which lines have been changed in the other window will not reflect the change.

`overlay(win1, win2)`
`overwrite(win1, win2)`
These functions overlay `win1` on top of `win2`; that is, all text in `win1` is copied into `win2`. The difference is that `overlay` is nondestructive (blanks are not copied) while `overwrite` is destructive.

Causing Output to the Terminal

`refresh()`
`wrefresh(win)`
These functions must be called to get any output on the terminal, as other routines merely manipulate data structures. `wrefresh` copies the named window to the physical terminal screen, taking into account what is already there in order to do optimizations. `refresh` is the same, using `stdscr` as a default screen. Unless **leaveok** has been enabled, the physical cursor of the terminal is left at the location of the window's cursor.

`doupdate()`
`wnoutrefresh(win)`
These two functions allow multiple updates with more efficiency than **wrefresh.** To use them, it is important to understand how **curses**

works. In addition to all the window structures, **curses** keeps two data structures representing the terminal screen: a *physical* screen, describing what is actually on the screen, and a *virtual* screen, describing what the programmer *wants* to have on the screen. `wrefresh` works by first copying the named window to the virtual screen (`wnoutrefresh`), and then calling the routine to update the screen (`doupdate`). If the programmer wishes to output several windows at once, a series of calls to `wrefresh` will result in alternating calls to `wnoutrefresh` and `doupdate`, causing several bursts of output to the screen. By calling `wnoutrefresh` for each window, it is then possible to call `doupdate` once, resulting in only one burst of output, with probably fewer total characters transmitted.

```
prefresh(pad,pminrow,pmincol,sminrow,
        smincol,smaxrow,smaxcol)
pnoutrefresh(pad,pminrow,pmincol,sminrow,
        smincol,smaxrow,smaxcol)
```

These routines are analogous to `wrefresh` and `wnoutrefresh` except that pads, instead of windows, are involved. The additional parameters are needed to indicate what part of the pad and screen are involved. `pminrow` and `pmincol` specify the upper left corner, in the pad, of the rectangle to be displayed. `sminrow`, `smincol`, `smaxrow`, and `smaxcol` specify the edges, on the screen, of the rectangle to be displayed in. The lower right corner in the pad of the rectangle to be displayed is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures.

Writing on Window Structures

These routines are used to "draw" text on windows. In all cases, a missing `win` is taken to be `stdscr`. `y` and `x` are the row and column, respectively. The upper left corner is always (0,0), not (1,1). The `mv` functions imply a call to `move` before the call to the other function.

Moving the Cursor

```
move(y, x)
wmove(win, y, x)
```

The cursor associated with the window is moved to the given location. This does not move the physical cursor of the terminal until

C-15

`refresh` is called. The position specified is relative to the upper left corner of the window.

Writing One Character

```
addch(ch)
waddch(win, ch)
mvaddch(y, x, ch)
mvwaddch(win, y, x, ch)
```

The character `ch` is put in the window at the current cursor position of the window. If `ch` is a tab, newline, or backspace, the cursor will be moved appropriately in the window. If `ch` is a different control character, it will be drawn in the ^X notation. The position of the window cursor is advanced. At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if **scrollok** is enabled, the scrolling region will be scrolled up one line.

The `ch` parameter is actually an integer, not a character. Video attributes can be combined with a character by or-ing them into the parameter. This will result in these attributes also being set. (The intent here is that text, including attributes, can be copied from one place to another with `inch` and `addch`.)

Writing a String

```
addstr(str)
waddstr(win,str)
mvaddstr(y,x,str)
mvwaddstr(win,y,x,str)
```

These functions write all the characters of the null terminated character string `str` on the given window. They are identical to a series of calls to `addch`.

Clearing Areas of the Screen

```
erase()
werase(win)
```

These functions copy blanks to every position in the window.

```
clear()
wclear(win)
```
These functions are like `erase` and `werase` but they also call `clearok`, arranging that the screen will be cleared on the next call to `refresh` for that window.

```
clrtobot()
wclrtobot(win)
```
All lines below the cursor in this window are erased. Also, the current line to the right of the cursor is erased.

```
clrtoeol()
wclrtoeol(win)
```
The current line to the right of the cursor is erased.

Inserting and Deleting Text

```
delch()
wdelch(win)
mvdelch(y,x)
mvwdelch(win,y,x)
```
The character under the cursor in the window is deleted. All characters to the right on the same line are moved to the left one position. This does not imply use of the hardware delete character feature.

```
deleteln()
wdeleteln(win)
```
The line under the cursor in the window is deleted. All lines below the current line are moved up one line. The bottom line of the window is cleared. This does not imply use of the hardware delete line feature.

```
insch(c)
winsch(win, c)
mvinsch(y,x,c)
mvwinsch(win,y,x,c)
```
The character `c` is inserted before the character under the cursor. All characters to the right are moved one space to the right, possibly losing the rightmost character on the line. This does not imply use of the hardware insert character feature.

```
insertln()
winsertln(win)
```
A blank line is inserted above the current line. The bottom line is lost. This does not imply use of the hardware insert line feature.

Formatted Output

```
printw(fmt, args)
wprintw(win, fmt, args)
mvprintw(y, x, fmt, args)
mvwprintw(win, y, x, fmt, args)
```
These functions correspond to `printf`. The characters which would be output by `printf` are instead output using `waddch` on the given window.

Miscellaneous

```
box(win, vert, hor)
```
A box is drawn around the edge of the window. `vert` and `hor` are the characters the box is to be drawn with.

```
scroll(win)
```
The window is scrolled up one line. This involves moving the lines in the window data structure. As an optimization, if the window is `stdscr` and the scrolling region is the entire window, the physical screen will be scrolled at the same time.

Input from a Window

```
getyx(win,y,x)
```
The cursor position of the window is placed in the two integer variables `y` and `x`. Since this is a macro, no & is necessary.

```
inch()
winch(win)
mvinch(y,x)
mvwinch(win,y,x)
```
The character at the current position in the named window is returned. If any attributes are set for that position, their values will be or-ed into the value returned. The predefined constants `A_ATTRIBUTES` and `A_CHARTEXT` can be used with the & operator to extract the character or attributes alone.

Input from the Terminal

```
getch()
wgetch(win)
mvgetch(y,x)
mvwgetch(win,y,x)
```
A character is read from the terminal associated with the window. In nodelay mode, if there is no input waiting, the value -1 is returned. In delay mode, the program will hang until the system passes text through to the program. Depending on the setting of **cbreak**, this will be after one character, or after the first newline.

If **keypad** mode is enabled, and a function key is pressed, the code for that function key will be returned instead of the raw characters. Possible function keys are defined with integers beginning with 0401, whose names begin with `KEY_`. These are listed in "Input" under "INTRODUCTION." If a character is received that could be the beginning of a function key (such as escape), **curses** will set a 1-second timer. If the remainder of the sequence does not come in within 1 second, the character will be passed through, otherwise the function key value will be returned. For this reason, on many terminals, there will be a one second delay after a user presses the escape key. (Use by a programmer of the escape key for a single character function is discouraged.)

```
getstr(str)
wgetstr(win,str)
mvgetstr(y,x,str)
mvwgetstr(win,y,x,str)
```

A series of calls to `getch` is made, until a newline is received. The resulting value is placed in the area pointed at by the character pointer `str`. The users' erase and kill characters are interpreted.

```
scanw(fmt, args)
wscanw(win, fmt, args)
mvscanw(y, x, fmt, args)
mvwscanw(win, y, x, fmt, args)
```

This function corresponds to `scanf`. `wgetstr` is called on the window, and the resulting line is used as input for the scan.

Video Attributes

```
attroff(at)
wattroff(win, attrs)
attron(at)
wattron(win, attrs)
attrset(at)
wattrset(win, attrs)
standout()
standend()
wstandout(win)
wstandend(win)
```

These functions set the *current attributes* of the named window. These attributes can be any combination of `A_STANDOUT`, `A_REVERSE`, `A_BOLD`, `A_DIM`, `A_BLINK`, and `A_UNDERLINE`. These constants are defined in `<curses.h>` and can be combined with the C | (or) operator.

The current attributes of a window are applied to all characters that are written into the window with `waddch`. Attributes are a property of the character, and move with the character through any scrolling and insert/delete line/character operations. To the extent possible on the particular terminal, they will be displayed as the graphic rendition of characters put on the screen.

`attrset`(at) sets the current attributes of the given window to `at`. `attroff`(at) turns off the named attributes without affecting any other attributes. `attron`(at) turns on the named attributes without affecting any others. `standout` is the same as `attron`(A_STANDOUT). `standend` is the same as `attrset`(0), that is, it turns off all attributes.

Bells and Flashing Lights

`beep()`
`flash()`
These functions are used to signal the programmer. `beep` will sound the audible alarm on the terminal, if possible, and if not, will flash the screen (visible bell), if that is possible. `flash` will flash the screen, and if that is not possible, will sound the audible signal. If neither signal is possible nothing will happen. Nearly all terminals have an audible signal (bell or beep) but only some can flash the screen.

Portability Functions

These functions do not directly involve terminal dependent character output but tend to be needed by programs that use **curses**. Unfortunately, their implementation varies from one version of UNIX† to another. They have been included here to enhance the portability of programs using **curses**.

`baudrate()`
`baudrate` returns the output speed of the terminal. The number returned is the integer baud rate, for example, 9600, rather than a table index such as `B9600`.

`erasechar()`
The erase character chosen by the user is returned. This is the character typed by the user to erase the character just typed.

---

* Trademark of AT&T Bell Laboratories

`killchar()`
The line kill character chosen by the user is returned. This is the character typed by the user to forget the entire line being typed.

`flushinp()`
`flushinp` throws away any typeahead that has been typed by the user and has not yet been read by the program.

Delays

These functions are highly unportable, but are often needed by programs that use **curses**, especially real time response programs. Some of these functions require a particular operating system or a modification to the operating system to work. In all cases, the routine will compile and return an error status if the requested action is not possible. It is recommended that programmers avoid use of these functions if possible.

`draino(ms)` The program is suspended until the output queue has drained enough to complete in `ms` additional milliseconds. Thus, `draino(50)` at 1200 baud would pause until there are no more than 6 characters in the output queue, because it would take 50 milliseconds to output the additional 6 characters. The purpose of this routine is to keep the program (and thus the keyboard) from getting ahead of the screen. If the operating system does not support the ioctls needed to implement `draino`, the value ERR is returned; otherwise, OK is returned.

`napms(ms)` This function suspends the program for `ms` milliseconds. It is similar to `sleep` except with higher resolution. The resolution actually provided will vary with the facilities available in the operating system, and often a change to the operating system will be necessary to produce good results. If resolution of at least .1 second is not possible, the routine will round to the next higher second, call sleep, and return ERR. Otherwise, the value OK is returned. Often the resolution provided is 1/60th second.

Lower Level Functions

These functions are provided for programs not needing the screen optimization capabilities of **curses**. Programs are discouraged from working at this level, since they must handle various glitches in certain terminals. However, a program can be smaller if it only brings in the low level routines.

Cursor Motion

`mvcur(oldrow, oldcol, newrow, newcol)`
This routine optimally moves the cursor from (oldrow, oldcol) to (newrow, newcol). The user program is expected to keep track of the current cursor position. Note that unless a full screen image is kept, **curses** will have to make pessimistic assumptions, sometimes resulting in less than optimal cursor motion. For example, moving the cursor a few spaces to the right can be done by transmitting the characters being moved over, but if **curses** does not have access to the screen image, it doesn't know what these characters are.

Terminfo Level

These routines are called by low level programs that need access to specific capabilities of **terminfo**. A program working at this level should include both `<curses.h>` and `<term.h>` in that order. After a call to `setupterm`, the capabilities will be available with macro names defined in `<term.h>`. See **terminfo**(4) for a detailed description of the capabilities.

Boolean valued capabilities will have the value 1 if the capability is present, 0 if it is not. Numeric capabilities have the value -1 if the capability is missing, and have a value at least 0 if it is present. String capabilities (both those with and without parameters) have the value `NULL` if the capability is missing, and otherwise have type `char *` and point to a character string containing the capability. The special character codes involving the \ and ^ characters (such as \r for return, or ^A for control A) are translated into the appropriate ASCII characters. Padding information (of the form $<time>) and parameter information (beginning with %) are left uninterpreted at this stage. The routine `tputs` interprets padding information, and `tparm` interprets parameter information.

C-15

If the program only needs to handle one terminal, the definition `-DSINGLE` can be passed to the C compiler, resulting in static references to capabilities instead of dynamic references. This can result in smaller code, but prevents use of more than one terminal at a time. Very few programs use more than one terminal, so almost all programs can use this flag.

`setupterm(term, filenum, errret)`
This routine is called to initialize a terminal. `term` is the character string representing the name of the terminal being used. `filenum` is the UNIX file descriptor of the terminal being used for output. `errret` is a pointer to an integer, in which a success or failure indication is returned. The values returned can be 1 (all is well), 0 (no such terminal), or -1 (some problem locating the **terminfo** database).

The value of `term` can be given as 0, which will cause the value of TERM in the environment to be used. The `errret` pointer can also be given as 0, meaning no error code is wanted. If `errret` is defaulted, and something goes wrong, `setupterm` will print an appropriate error message and exit, rather than returning. Thus, a simple program can call `setupterm(0, 1, 0)` and not worry about initialization errors.

If the variable TERMINFO is set in the environment to a path name, `setupterm` will check for a compiled **terminfo** description of the terminal under that path, before checking /etc/term. Otherwise, only /etc/term is checked.

`setupterm` will check the tty driver mode bits, using `filenum`, and change any that might prevent the correct operation of other low level routines. Currently, the mode that expands tabs into spaces is disabled, because the tab character is sometimes used for different functions by different terminals. (Some terminals use it to move right one space. Others use it to address the cursor to row or column 9.) If the system is expanding tabs, `setupterm` will remove the definition of the `tab` and `backtab` functions, making the assumption that since the user is not using hardware tabs, they may not be properly set in the terminal. Other system dependent changes, such as disabling a virtual terminal driver, may be made here.

As a side effect, `setupterm` initializes the global variable `ttytype`, which is an array of characters, to the value of the list of names for the terminal. This list comes from the beginning of the **terminfo** description.

After the call to `setupterm`, the global variable `cur_term` is set to point to the current structure of terminal capabilities. By calling `setupterm` for each terminal, and saving and restoring `cur_term`, it is possible for a program to use two or more terminals at once.

The mode that turns newlines into CRLF on output is not disabled. Programs that use `cursor_down` or `scroll_forward` should avoid these capabilities if their value is linefeed unless they disable this mode. `setupterm` calls `reset_prog_mode` after any changes it makes.

`reset_prog_mode()`
`reset_shell_mode()`
`def_prog_mode()`
`def_shell_mode()`
These routines can be used to change the tty modes between the two states: *shell* (the mode they were in before the program was started) and *program* (the mode needed by the program). `def_prog_mode` saves the current terminal mode as program mode. `setupterm` and `initscr` call def_shell_mode automatically. `reset_prog_mode` puts the terminal into program mode, and `reset_shell_mode` puts the terminal into normal mode.

A typical calling sequence is for a program to call `initscr` (or `setupterm` if a **terminfo** level program), then to set the desired program mode by calling routines such as `cbreak` and `noecho`, then to call `def_prog_mode` to save the current state. Before a shell escape or control-Z suspension, the program should call `reset_shell_mode`, to restore normal mode for the shell. Then, when the program resumes, it should call `reset_prog_mode`. Also, all programs must call `reset_shell_mode` before they exit. (The higher level routine `endwin` automatically calls `reset_shell_mode`.)

Normal mode is stored in `cur_term->Ottyb`, and program mode is in `cur_term->Nttyb`. These structures are both of type `SGTTYB`

(which varies depending on the system). Currently the possible types are `struct sgttyb` (on some other systems) and `struct termio` (on this version of the UNIX system). `def_prog_mode` should be called to save the current state in `Nttyb`.

`vidputs(newmode, putc)`
`newmode` is any combination of attributes, defined in `<curses.h>`. `putc` is a putchar-like function. The proper string to put the terminal in the given video mode is output. The previous mode is remembered by this routine. The result characters are passed through `putc`.

`vidattr(newmode)`
The proper string to put the terminal in the given video mode is output to **stdout**.

`tparm(instring, p1, p2, p3, p4, p5, p6, p7, p8, p9)`
`tparm` is used to instantiate a parameterized string. The character string returned has the given parameters applied, and is suitable for `tputs`. Up to 9 parameters can be passed, in addition to the parameterized string.

`tputs(cp, affcnt, outc)`
A string capability, possibly containing padding information, is processed. Enough padding characters to delay for the specified time replace the padding specification, and the resulting string is passed, one character at a time, to the routine `outc`, which should expect one character parameter. (This routine often just calls `putchar`.) `cp` is the capability string. `affcnt` is the number of units affected by the capability, which varies with the particular capability. (For example, the `affcnt` for `insert_line` is the number of lines below the inserted line on the screen, that is, the number of lines that will have to be moved by the terminal.) `affcnt` is used by the padding information of some terminals as a multiplication factor. If the capability does not have a factor, the value 1 should be passed.

`putp(str)`
This is a convenient function to output a capability with no `affcnt`. The string is output to `putchar` with an `affcnt` of 1. It can be used in simple applications that do not need to process the output of `tputs`.

`delay_output(ms)`
A delay is inserted into the output stream for the given number of milliseconds. The current implementation inserts sufficient pad characters for the delay. This should not be used in place of a high resolution sleep, but rather for delay effects in the output. Due to buffering in the system, it is unlikely that this call will result in the process actually sleeping. Since large numbers of pad characters can be output, it is recommended that `ms` not exceed 500.

OPERATION DETAILS

These paragraphs describe many of the details of how the **curses** and terminfo package operates.

Insert and Delete Line and Character

The algorithm used by **curses** takes into account insert and delete line and character functions, if available, in the terminal. Calling the routine

```
idlok(stdscr, TRUE);
```

will enable insert/delete line. By default, **curses** will not use insert/delete line. This was not done for performance reasons, since there is no speed penalty involved. Rather, experience has shown that some programs do not need this facility, and that if **curses** uses insert/delete line, the result on the screen can be visually annoying. Since many simple programs using **curses** do not need this, the default is to avoid insert/delete line. Insert/delete character is always considered.

Additional Terminals

Curses will work even if absolute cursor addressing is not possible, as long as the cursor can be moved from any location to any other location. It considers local motions, parameterized motions, home, and carriage return.

Curses is aimed at full duplex, alphanumeric, video terminals. No attempt is made to handle half-duplex, synchronous, hard copy, or bitmapped terminals. Bitmapped terminals can be handled by programming the bitmapped terminal to emulate an ordinary alphanumeric terminal. This does not take advantage of the bitmap capabilities, but it is the fundamental nature of **curses** to deal with alphanumeric terminals.

The **curses** handles terminals with the "magic cookie glitch" in their video attributes. The term "magic cookie" means that a change in video attributes is implemented by storing a "magic cookie" in a location on the screen. This "cookie" takes up a space, preventing an exact implementation of what the programmer wanted. Curses takes the extra space into account, and moves part of the line to the right, as necessary. In some cases, this will unavoidably result in losing text from the right hand edge of the screen. Advantage is taken of existing spaces.

Multiple Terminals

Some applications need to display text on more than one terminal, controlled by the same process. Even if the terminals are of different types, **curses** can handle this.

All information about the current terminal is kept in a global variable

```
struct screen *SP;
```

Although the screen structure is hidden from the user, the C compiler will accept declarations of variables which are pointers. The user

program should declare one screen pointer variable for each terminal it wishes to handle. The routine

```
struct screen *
newterm(type, fd)
```

will set up a new terminal of the given terminal type which does output on file descriptor fd. A call to `initscr` is essentially `newterm(getenv(''TERM''),stdout)`. A program wishing to use more than one terminal should use `newterm` for each terminal and save the value returned as a reference to that terminal.

To switch to a different terminal, call

```
set_term(term)
```

The old value of SP will be returned. The programmer should not assign directly to SP because certain other global variables must also be changed.

All **curses** routines always affect the current terminal. To handle several terminals, switch to each one in turn with `set_term`, and then access it. Each terminal must be set up with `newterm`, and closed down with `endwin`.

Video Attributes

Video attributes can be displayed in any combination on terminals with this capability. They are treated as an extension of the standout capability, which is still present.

Each character position on the screen has 16 bits of information associated with it. Seven of these bits are the character to be displayed, leaving separate bits for nine video attributes. These bits are used for standout, underline, reverse video, blink, dim, bold, blank, protect, and alternate character set. Standout is taken to be whatever highlighting works best on the terminal, and should be used by any program that does not need specific or combined attributes. Underlining, reverse video, blink, dim, and bold are the usual video

attributes. Blank means that the character is displayed as a space, for security reasons. Protected and alternate character set depend on the particular terminal. The use of these last three bits is subject to change and not recommended. Note also that not all terminals implement all attributes - in particular, no current terminal implements both dim and bold.

The routines to use these attributes include

```
attrset(attrs)          wattrset(win, attrs)
attron(attrs)           wattron(win, attrs)
attroff(attrs)          wattroff(win, attrs)
standout()              wstandout(win)
standend()              wstandend(win)
```

Attributes, if given, can be any combination of `A_STANDOUT`, `A_UNDERLINE`, `A_REVERSE`, `A_BLINK`, `A_DIM`, `A_BOLD`, `A_INVIS`, `A_PROTECT`, and `A_ALTCHARSET`. These constants, defined in **curses**.h, can be combined with the C | (or) operator to get multiple attributes. `attrset` sets the current attributes to the given `attrs`; `attron` turns on the given `attrs` in addition to any attributes that are already on; `attroff` turns off the given attributes, without affecting any others. `standout` and `standend` are equivalent to `attron(A_STANDOUT)` and `attrset(A_NORMAL)`.

If the particular terminal does not have the particular attribute or combination requested, **curses** will attempt to use some other attribute in its place. If the terminal has no highlighting at all, all attributes will be ignored.

Special Keys

Many terminals have special keys, such as arrow keys, keys to erase the screen, insert or delete text, and keys intended for user functions. The particular sequences these terminals send differs from terminal to terminal. **Curses** allows the programmer to handle these keys.

A program using special keys should turn on the keypad by calling

```
keypad(stdscr, TRUE)
```

at initialization. This will cause special characters to be passed through to the program by the function `getch`. These keys have constants which are listed in "Input" under "INTRODUCTION." They have values starting at 0401, so they should not be stored in a **char** variable, as significant bits will be lost.

A program using special keys should avoid using the **escape** key, since most sequences start with escape, creating an ambiguity. **Curses** will set a one second alarm to deal with this ambiguity, which will cause delayed response to the escape key. It is a good idea to avoid escape in any case, since there is eventually pressure for nearly *any* screen oriented program to accept arrow key input.

Scrolling Region

There is a programmer accessible scrolling region. Normally, the scrolling region is set to the entire window, but the calls

```
setscrreg(top, bot)
wsetscrreg(win, top, bot)
```

set the scrolling region for `stdscr` or the given window to any combination of top and bottom margins. When scrolling past the bottom margin of the scrolling region, the lines in the region will move up one line, destroying the top line of the region. If scrolling has been enabled with `scrollok`, scrolling will take place only within that window. Note that the scrolling region is a software feature, and only causes a window data structure to scroll. This may or may not translate to use of the hardware scrolling region feature of a terminal, or insert/delete line.

Mini-Curses

Curses copies from the current window to an internal screen image for every call to `refresh`. If the programmer is only interested in screen output optimization, and does not want the windowing or input functions, an interface to the lower level routines is available. This will make the program somewhat smaller and faster. The interface is a subset of full **curses**, so that conversion between the levels is not necessary to switch from mini-curses to full **curses**.

The following functions of **curses** and terminfo are available to the user of minicurses:

addch(ch)	addstr(str)	attroff(at)	attron(at)
attrset(at)	clear()	erase()	initscr
move(y, x)	mvaddch(y,x,ch)	mvaddstr(y,x,str)	newterm
refresh()	standend()	standout()	

The following functions of **curses** and terminfo are *not* available to the user of minicurses:

box	clrtobot	clrtoeol	delch
deleteln	delwin	getch	getstr
inch	insch	insertln	longname
makenew	mvdelch	mvgetch	mvgetstr
mvinch	mvinsch	mvprintw	mvscanw
mvwaddch	mvwaddstr	mvwdelch	mvwgetch
mvwgetstr	mvwin	mvwinch	mvwinsch
mvwprintw	mvwscanw	newwin	overlay
overwrite	printw	putp	scanw
scroll	setscrreg	subwin	touchwin
vidattr	waddch	waddstr	wclear
wclrtobot	wclrtoeol	wdelch	wdeleteln
werase	wgetch	wgetstr	winsch
winsertln	wmove	wprintw	wrefresh
wscanw	wsetscrreg		

The subset mainly requires the programmer to avoid use of more than the one window `stdscr`. Thus, all functions beginning with "w" are generally undefined. Certain high level functions that are convenient but not essential are also not available, including `printw` and `scanw`. Also, the input routine `getch` cannot be used with

mini-curses. Features implemented at a low level, such as use of hardware insert/delete line and video attributes, are available in both versions. Also, mode setting routines such as `crmode` and `noecho` are allowed.

To access mini-curses, add `-DMINICURSES` to the CFLAGS in the makefile. If routines are requested that are not in the subset, the loader will print error messages such as

```
Undefined:
m_getch
m_waddch
```

to tell you that the routines `getch` and `waddch` were used but are not available in the subset. Since the preprocessor is involved in the implementation of mini-curses, the entire program must be recompiled when changing from one version to the other.

TTY Mode Functions

In addition to the save/restore routines `savetty()` and `resetty()`, standard routines are available for going into and out of normal tty mode. These routines are `resetterm()`, which puts the terminal back in the mode it was in when **curses** was started; `fixterm()`, which undoes the effects of `resetterm`, that is, restores the "current **curses** mode"; and `saveterm()`, which saves the current state to be used by `fixterm()`. `endwin` automatically calls `resetterm`, and the routine to handle control-Z (on other systems that have process control) also uses `resetterm` and `fixterm`. Programmers should use these routines before and after shell escapes, and also if they write their own routine to handle control-Z. These routines are also available at the *terminfo* level.

Typeahead Check

If the user types something during an update, the update will stop, pending a future update. This is useful when the user hits several keys, each of which causes a good deal of output. For example, in a screen editor, if the user presses the "forward screen" key, which draws the next screen full of text, several times rapidly, rather than drawing several screens of text, the updates will be cut short, and

only the last screen full will actually be displayed. This feature is automatic and cannot be disabled. The feature only works on versions of the UNIX system with the necessary support in the operating system.

getstr

No matter what the setting of *echo* is, strings typed in here are echoed at the current cursor location. The users erase and kill characters are understood and handled. This makes it unnecessary for an interactive program to deal with erase, kill, and echoing when the user is typing a line of text.

longname

The `longname` function does not need any arguments. It returns a pointer to a static area containing the actual long name of the terminal.

Nodelay Mode

The call

```
nodelay(stdscr, TRUE)
```

will put the terminal in "nodelay mode". While in this mode, any call to `getch` will return -1 if there is nothing waiting to be read immediately. This is useful for writing programs requiring "real time" behavior where the users watch action on the screen and press a key when they want something to happen. For example, the cursor can be moving across the screen, in real time. When it reaches a certain point, the user can press an arrow key to change direction at that point.

Portability

Several useful routines are provided to improve portability. The implementation of these routines is different from system to system, and the differences can be isolated from the user program by including them in **curses**.

Functions `erasechar()` and `killchar()` return the characters which erase one character, and kill the entire input line, respectively. The function `baudrate()` will return the current baud rate, as an integer. (For example, at 9600 baud, the integer 9600 will be returned, not the value `B9600` from `<sgtty.h>`.) The routine `flushinp()` will cause all typeahead to be thrown away.

Chapter 16

CURSES EXAMPLES

The following examples are provided to demonstrate uses of **curses**. They are for illustration purposes only. A good programmer would expand the programs presented here before using them.

EXAMPLE PROGRAM 'editor'

```
/*
 * editor: A screen-oriented editor.  The user
 * interface is similar to a subset of vi.
 * The buffer is kept in stdscr itself to simplify
 * the program.
 */

#include <curses.h>

#define CTRL(c) ('c' & 037)

main(argc, argv)
char **argv;
{
     int i, n, l;
     int c;
     FILE *fd;

     if (argc != 2) {
          fprintf(stderr, " Usage: edit file0);
          exit(1);
     }

     fd = fopen(argv[1], " r" );
     if (fd == NULL) {
          perror(argv[1]);
          exit(2);
     }

     initscr();
     cbreak();
     nonl();
```

```
        noecho();
        idlok(stdscr, TRUE);
        keypad(stdscr, TRUE);

        /* Read in the file */
        while ((c = getc(fd)) != EOF)
                addch(c);
        fclose(fd);

        move(0,0);
        refresh();
        edit();

        /* Write out the file */
        fd = fopen(argv[1], "w" );
        for (l=0; l<23; l++) {
                n = len(l);
                for (i=0; i<n; i++)
                        putc(mvinch(l, i), fd);
                putc('0, fd);
        }
        fclose(fd);

        endwin();
        exit(0);
}

len(lineno)
int lineno;
{
        int linelen = COLS-1;

        while (linelen >=0 &&mvinch(lineno, linelen) == ' ')
                linelen--;
        return linelen + 1;
}

/* Global value of current cursor position */
int row, col;

edit()
{
        int c;
```

```
for (;;) {
    move(row, col);
    refresh();
    c = getch();
    switch (c) {   /* Editor commands */

    /* hjkl and arrow keys: move cursor */
    /* in direction indiated */
    case 'h':
    case KEY_LEFT:
        if (col > 0)
            col--;
        break;

    case 'j':
    case KEY_DOWN:
        if (row < LINES-1)
            row++;
        break;

    case 'k':
    case KEY_UP:
        if (row > 0)
            row--;
        break;

    case 'l':
    case KEY_RIGHT:
        if (col < COLS-1)
            col++;
        break;

    /* i: enter input mode */
    case KEY_IC:
    case 'i':
        input();
        break;

    /* x: delete current character */
    case KEY_DC:
    case 'x':
        delch();
        break;
```

```
            /* o: open up a new line and enter input mode */
            case KEY_IL:
            case 'o':
                move(++row, col=0);
                insertln();
                input();
                break;

            /* d: delete current line */
            case KEY_DL:
            case 'd':
                deleteln();
                break;

            /* ^L: redraw screen */
            case KEY_CLEAR:
            case CTRL(L):
                clearok(curscr);
                refresh();
                break;

            /* w: write and quit */
            case 'w':
                return;

            /* q: quit without writing */
            case 'q':
                endwin();
                exit(1);
            default:
                flash();
                break;
            }
        }
}

/*
 * Insert mode: accept characters and insert them.
 *  End with ^D or EIC
 */
input()
{
    int c;
```

```
standout();
mvaddstr(LINES-1, COLS-20, " INPUT MODE" );
standend();
move(row, col);
refresh();
for (;;) {
      c = getch();
      if (c == CTRL(D) || c == KEY_EIC)
            break;
      insch(c);
      move(row, ++col);
      refresh();
}
move(LINES-1, COLS-20);
clrtoeol();
move(row, col);
refresh();
```

EXAMPLE PROGRAM 'highlight'

```
/*
 * highlight: a program to turn U, B, and
 * N sequences into highlighted
 * output, allowing words to be
 * displayed underlined or in bold.
 */
#include <curses.h>

main(argc, argv)
char **argv;
{
     FILE *fd;
     int c, c2;

     if (argc != 2) {
          fprintf(stderr, " Usage: highlight file0);
          exit(1);
     }

     fd = fopen(argv[1], " r" );
     if (fd == NULL) {
          perror(argv[1]);
          exit(2);
     }

     initscr();
     scrollok(stdscr, TRUE);

     for (;;) {
          c = getc(fd);
          if (c == EOF)
               break;
          if (c == '\') {
               c2 = getc(fd);
               switch (c2) {
                    'B':
                    attrset(A_BOLD);
                    continue;
               case 'U':
                    attrset(A_UNDERLINE);
                    continue;
               case 'N':
```

```
                    attrset(0);
                    continue;
                }
                addch(c);
                addch(c2);
            }
            else
                addch(c);
        }
        fclose(fd);
        refresh();
        endwin();
        exit(0);
}
```

EXAMPLE PROGRAM 'scatter'

```
/*
 *   SCATTER. This program takes the first
 *   23 lines from the standard
 *   input and displays them on the
 *   VDU screen, in a random manner.
 */

#include <curses.h>

#define MAXLINES 120
#define MAXCOLS  160
char s[MAXLINES][MAXCOLS];/* Screen Array */

main()
{
     register int row=0,col=0;
     register char c;
     int char_count=0;
     long t;
     char buf[BUFSIZ];

     initscr();
     for(row=0;row<MAXLINES;row++)
          for(col=0;col<MAXCOLS;col++)
               s[row][col]=' ';

     row = 0;
     /* Read screen in */
     while( (c=getchar()) != EOF &&row < LINES ) {
          if(c != '0) {
               /* Place char in screen array */
               s[row][col++] = c;
               if(c != ' ')
                    char_count++;
          } else {
               col=0;
               row++;
          }
     }

     time(&t);/* Seed the random number generator */
     srand((int)(t&0177777L));
```

```
        while(char_count) {
            row=rand() % LINES;
            col=(rand()>>2) % COLS;
            if(s[row][col] != ' ')
            {
                move(row, col);
                addch(s[row][col]);
                s[row][col]=EOF;
                char_count--;
                refresh();
            }
        }
        endwin();
        exit(0);
}
```

EXAMPLE PROGRAM 'show'

```
#include <curses.h>
#include <signal.h>

main(argc, argv)
int argc;
char *argv[];
{
    FILE *fd;
    char linebuf[BUFSIZ];
    int line;
    void done(), perror(), exit();

    if(argc != 2)
    {
        fprintf(stderr," usage: %s file0, argv[0]);
        exit(1);
    }
    if((fd=fopen(argv[1]," r" )) == NULL)
    {
        perror(argv[1]);
        exit(2);
    }
    signal(SIGINT, done);

    initscr();
    noecho();
    cbreak();
    nonl();
    idlok(stdscr, TRUE);

    while(1)
    {
        move(0,0);
        for(line=0; line<LINES; line++)
        {
            if(fgets(linebuf, sizeof linebuf, fd) == NULL)
            {
                clrtobot();
                done();
            }
            move(line, 0);
            printw(" %s" , linebuf);
```

```
            }
            refresh();
            if(getch() == 'q')
                done();
        }
    }

void
done()
{
    move(LINES-1, 0);
    clrtoeol();
    refresh();
    endwin();
    exit(0);
}
```

EXAMPLE PROGRAM 'termhl'

```
/*
 * A terminfo level version of highlight.
 */
#include <curses.h>
#include <term.h>

int ulmode = 0;          /* Currently underlining */

main(argc, argv)
char **argv;
{
     FILE *fd;
     int c, c2;
     int outch();

     if (argc > 2) {
          fprintf(stderr, " Usage: termhl [file]0);
          exit(1);
     }

     if (argc == 2) {
          fd = fopen(argv[1], " r" );
          if (fd == NULL) {
               perror(argv[1]);
               exit(2);
          }
     } else {
          fd = stdin;
     }

     setupterm(0, 1, 0);

     for (;;) {
          c = getc(fd);
          if (c == EOF)
               break;
          if (c == '\') {
               c2 = getc(fd);
               switch (c2) {
               case 'B':
                    tputs(enter_bold_mode, 1, outch);
                    continue;
```

```
                case 'U':
                    tputs(enter_underline_mode, 1, outch);
                    ulmode = 1;
                    continue;
                case 'N':
                    tputs(exit_attribute_mode, 1, outch);
                    ulmode = 0;
                    continue;
                }
                putch(c);
                putch(c2);
            }
            else
                putch(c);
        }
        fclose(fd);
        fflush(stdout);
        resetterm();
        exit(0);
}

/*
 * This function is like putchar, but it checks for underlining.
 */
putch(c)
int c;
{
    outch(c);
    if (ulmode &&underline_char) {
        outch('
)           tputs(underline_char, 1, outch);
;       }
}

/*
 * Outchar is a function version of putchar that can be passed to
 * tputs as a routine to call.
 */
outch(c)
int c;
{
    putchar(c);
}
```

EXAMPLE PROGRAM 'two'

```
#include <curses.h>
#include <signal.h>

struct screen *me, *you;
struct screen *set_term();

FILE *fd, *fdyou;
char linebuf[512];

main(argc, argv)
char **argv;
{
    int done();
    int c;

    if (argc != 4) {
        fprintf(stderr, " Usage: two othertty otherttytype inputfile0);
        exit(1);
    }

    fd = fopen(argv[3], " r" );
    fdyou = fopen(argv[1], " w+" );
    signal(SIGINT, done);  /* die gracefully */

    me = newterm(getenv(" TERM" ), stdout);/* initialize my tty */
    you = newterm(argv[2], fdyou);/* Initialize his terminal */

    set_term(me);           /* Set modes for my terminal */
    noecho();           /* turn off tty echo */
    cbreak();           /* enter cbreak mode */
    nonl();             /* Allow linefeed */
    nodelay(stdscr,TRUE); /* No hang on input */

    set_term(you);          /* Set modes for other terminal */
    noecho();
    cbreak();
    nonl();
    nodelay(stdscr,TRUE);

    /* Dump first screen full on my terminal */
    dump_page(me);
```

```
        /* Dump second screen full on his terminal */
        dump_page(you);

        for (;;) {              /* for each screen full */
                set_term(me);
                c = getch();
                if (c == 'q')           /* wait for user to read it */
                        done();
                if (c == ' ')
                        dump_page(me);

                set_term(you);
                c = getch();
                if (c == 'q')           /* wait for user to read it */
                        done();
                if (c == ' ')
                        dump_page(you);
                sleep(1);
        }
}

dump_page(term)
struct screen *term;
{
        int line;

        set_term(term);
        move(0, 0);
        for (line=0; line<LINES-1; line++) {
                if (fgets(linebuf, sizeof linebuf, fd) == NULL) {
                        clrtobot();
                        done();
                }
                mvprintw(line, 0, " %s" , linebuf);
        }
        standout();
        mvprintw(LINES-1, 0, " --More--" );
        standend();
        refresh();      /* sync screen */
}

/*
 * Clean up and exit.
```

```
 */
done()
{
    /* Clean up first terminal */
    set_term(you);
    move(LINES-1,0); /* to lower left corner */
    clrtoeol();          /* clear bottom line */
    refresh();      /* flush out everything */
    endwin();       /* curses cleanup */

    /* Clean up second terminal */
    set_term(me);
    move(LINES-1,0); /* to lower left corner */
    clrtoeol();          /* clear bottom line */
    refresh();      /* flush out everything */
    endwin();       /* curses cleanup */

    exit(0);
}
```

EXAMPLE PROGRAM 'window'

```
#include <curses.h>

WINDOW *cmdwin;

main()
{
     int i, c;
     char buf[120];

     initscr();
     nonl();
     noecho();
     cbreak();

     cmdwin = newwin(3, COLS, 0, 0);/* top 3 lines */
     for (i=0; i<LINES; i++)
          mvprintw(i, 0, " This is line %d of stdscr" , i);

     for (;;) {
          refresh();
          c = getch();
          switch (c) {
          case 'c':  /* Enter command from keyboard */
               werase(cmdwin);
               wprintw(cmdwin, " Enter command:" );
               wmove(cmdwin, 2, 0);
               for (i=0; i<COLS; i++)
                    waddch(cmdwin, '-');
               wmove(cmdwin, 1, 0);
               touchwin(cmdwin);
               wrefresh(cmdwin);
               wgetstr(cmdwin, buf);
               touchwin(stdscr);
               /*
                * The command is now in buf.
                * It should be processed here.
```

```
                */
                break;
            case 'q':
                endwin();
                exit(0);
            }
        }
    }
```

NOTES

NOTES

NOTES

NOTES