UniPlus+®
System V, Release 2

Volume 7

Programming Guide

This guide was edited and enhanced by Mary Ann Finnerty of UniSoft Systems.

CONTENTS

PREFACE

This guide describes the 68000 C programming language supported by the UniPlus+® Operating System, the assembler, linker, common object file format (COFF), debugging programs and the FORTRAN, RATFOR and EFL programming languages.

It is assumed that those using this guide have at least two years of specialized training in computer-related fields and should prove especially useful to those using the UniPlus+ Operating System for system development.

This guide contains 13 chapters:

Chapter 1. C INTERFACE NOTES
This chapter describes the way in which the UniSoft 68000 C programming language represents data in storage and how that data is passed between functions.

Chapter 2. C COMPILER
This chapter describes the use and options of the UniSoft Systems 68000 C compiler, **cc**.

Chapter 3. C LANGUAGE
This chapter provides a summary of the grammar and rules of the C programming language which was used to write most of the UNIX™ operating system.

Chapter 4. C LIBRARIES
This chapter describes the functions and declarations that support the C Language and how to use these functions.

Chapter 5. OBJECT AND MATH LIBRARIES
This chapter describes the Object file and Math libraries that are supported by the UniPlus+

Operating System.

Chapter 6. MOTOROLA 68000 ASSEMBLER
This chapter describes the machine language of the System V, Release 2 UniPlus$^+$ Operating System.

Chapter 7. COMMON LINK EDITOR
This chapter (**LD**) describes the options and usage of the UniPlus$^+$ Operating System link editor.

Chapter 8. COMMON OBJECT FILE FORMAT
This chapter (**COFF**) describes the object file format produced by both the C and FORTRAN compilers in UniPlus$^+$ System V, Release 2.

Chapter 9. FORTRAN 77
This chapter describes the compiler and run-time system for Fortran 77 as implemented on the UniPlus$^+$ Operating System.

Chapter 10. RATFOR
This chapter describes the **ratfor(1)** preprocessor which allows the user to write FORTRAN programs in a fashion similar to the C programming language.

Chapter 11. THE EFL PROGRAMMING LANGUAGE
This chapter describes a clean, general purpose computer language intended to encourage portable programming. Although the name EFL originally stood for "Extended Fortran Language" and EFL programs can be translated into efficient Fortran code, the EFL programmer can take advantage of the ubiquity and portability of Fortran (and the software and libraries written in that language), without suffering from Fortran's failings.

Chapter 12. LINT
This chapter describes a program that attempts to detect compile-time bugs and non-portable

features in C programs.

Chapter 13. SYMBOLIC DEBUGGER
This chapter **(SDB)** describes the symbolic debugger **sdb(1)** for UniPlus+ Operating System object files.

Throughout this document, any reference of the form **name(1M)**, **name(7)**, or **name(8)** refers to entries in the *UniPlus+ Administrator Manual.*

Any reference of the form **name(N)** where **N** is a number 1 through 6, possibly followed by a letter, refers to entry **name** in section N of the *UniPlus+ User Manual.*

Chapter 1: C INTERFACE NOTES

CONTENTS

Chapter 1
C INTERFACE NOTES

1. Introduction

This chapter describes the way in which the UniSoft® 68000 C programming language represents data in storage and passes data between functions. Also described are the environment of and calling mechanism for a function.

The information in this chapter is intended for programmers who must have detailed knowledge of the interface mechanisms in order to match C code with the assembler. It is also intended for those who wish to write new system or mathematical functions.

When a C program is compiled and assembled, the program is split into three parts:

1. **.text**	The executable code of the program. The compiler/assembler combination produces this.
2. **.data**	The initialized data area. This contains literal constants, character strings, and so on. The compiler/assembler combination produces this.
3. **.bss**	The uninitialized data areas. The loader generates and clears this area to zero at load time. This is a feature of the system and can be relied upon.

During execution of a program, the stack area contains indeterminate data. In other words, its previous contents (if any) cannot be relied upon.

2. Data Representations

In general, all data elements of whatever size are stored such that their least significant bit is in the highest addressed byte and their most significant bit is in the lowest addressed byte. The list below describes the representation of data.

char	Values of type *char* occupy 8 bits. Such values can be aligned on any byte boundary.
short	Values of type *short* occupy 16 bits. Values of type *short* are aligned on word (16-bit) address boundaries.
long	Values of type *long* occupy 32 bits. A *long* value is the same as an *int* value in 68000 C. Values of this type are aligned on word (16-bit) boundaries.
float	Values of type *float* occupy 32 bits. All *float* values are automatically converted to type *double* for computation purposes — except when testing for zero or non-zero. Values of this type are aligned on word (16-bit) boundaries. A *float* value consists of a sign bit, followed by an 8-bit biased exponent, followed by a 23-bit mantissa.
double	Values of type *double* occupy 64 bits. Values of this type are aligned on word (16-bit) boundaries. A *double* value consists of a sign bit, followed by an 11-bit biased exponent, followed by a 52-bit mantissa.
pointers	All *pointers* are represented as long (32-bit) values. Pointers are aligned on word (16-bit) boundaries.

arrays

The base address of an *array* value is always aligned on a word (16-bit) address boundary.

Elements of an array are stored contiguously, one after the other. Elements of multi-dimensional arrays are stored in row-major order. That is, the last dimension of an array varies the fastest.

When a multi-dimensional array is declared, it is possible to omit the size specification for the last dimension. In such a case, what is allocated is actually an array of pointers to the elements of the last dimension.

structures and unions

Within structures and unions, it is possible to obtain unfilled holes of size *char*. This is due to the compiler rounding addresses up to 16-bit boundaries to accommodate word-aligned elements.

This situation can best be demonstrated by an example. Consider the following structure:

```
struct {
    int   x;        /* This is a 32-bit element        */
    char  y;        /* Takes up a single byte          */
    short z;        /* Aligned to a 16-bit boundary */
};
```

The total number of bytes declared above is seven: four for the *int*, one for the *char*, and two for the *short*.

In reality, the "z" field which is a *short* will be aligned on a 16-bit boundary by the C compiler. In this case, the compiler inserts a hole after the *char* element "y", to align the *short* element "z". The net effect of these machinations is a structure that behaves like this:

```
struct {
    int   x;        /* This is a 32-bit element       */
    char  y;        /* Takes up a single byte         */
    char  dummy;    /* Fills the structure            */
    short z;        /* Aligned to a 16-bit boundary   */
};
```

The C compiler never reorders any parts of a structure.

Similar considerations apply to arrays of structures or unions. Each element of an array (other than an array of *char*) begins on a 16-bit boundary.

For a detailed treatment of data storage, consult *The C Programming Language* by Kernighan and Ritchie.

3. Parameter Passing in C

The C programming language is unique in that it really has only functions. The effect of a subroutine is achieved simply by having a function which does not return a value. The function type should be "void."

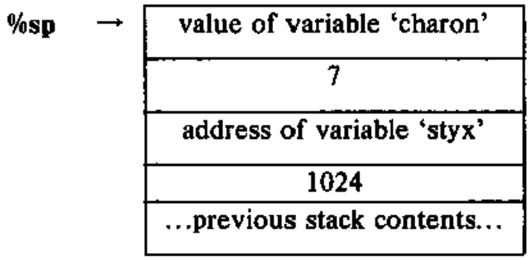
Another unique feature of C is that parameters to functions are always passed by value. The C programming language has no concept of declaring parameters to be passed by reference, as in languages such as Pascal. In order to pass a parameter by reference in a C program, the programmer must explicitly pass the address of the parameter. The called function must be aware that it is receiving an address instead of a value, and the appropriate code must be present to handle that case.

When a function is called, its parameters (if any) are evaluated and are then pushed onto the stack in **reverse order**. All parameters are pushed onto the stack as 32-bit longs. If a parameter is shorter than 32 bits, it is expanded to a 32-bit value with sign-extension, if necessary. The calling procedure is responsible for popping the parameters off the stack.

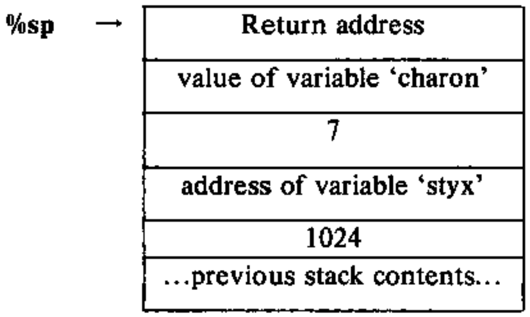
Consider a C function call like this:

```
ferry (charon, 7, &styx, 1<<10);
```

After evaluation, but just before the call, the stack looks like this:


%sp →
value of variable 'charon'
7
address of variable 'styx'
1024
...previous stack contents...

Functions are called by issuing either a "bsr" instruction or a "jsr" instruction, depending upon whether the callee is within a 16-bit addressing range or not, and whether the C optimizer was used. The "bsr" or "jsr" instruction pushes the return address upon the stack, and then branches to the indicated function. After the call, on entry to the function, the stack looks like this:


%sp
Return address
value of variable 'charon'
7
address of variable 'styx'
1024
...previous stack contents...

In each function, register %a6 is used as a stack frame base. The stack location referenced by %a6 contains the return address.

4. Setting Up the Stack

Upon entry into the function, the prolog code is executed. The prolog code allocates enough space on the stack for the local variables, plus enough space to save any registers that this function uses. The prolog code then ensures that there is enough stack space available for executing the function. If there is not enough space, the system grows the stack to allot more space. The prolog code for the 68000 looks like this:

```
        lea.l    F%1-256(%sp),%a0
        cmp.l    %a0,splimit%
        bhi.b    L%12
        jsr      spgrow%
L7.12:

        link     %fp,&F%1
        moveml   &M%1,S%1(%fp)
```

The first section of the above code is a stack test, and the second section is the "normal prolog code." The prolog code for the 68010 does NOT contain a stack test section, but consists only of a normal prolog code identical to that of the 68000.

The "F%1" constant is the size of the stack frame for the local variables, plus four bytes for each register variable.

For the 68000, the current lower bound for the stack is compared against the current function's requirements (plus a safety factor of 256 bytes) and if the available stack is not sufficient, calls **spgrow%** to grow the stack.

Finally, the "M%1" constant is a mask to determine which registers need to be saved on the stack for this particular function. This is, of course, dependent on the register variables that the programmer declared for that particular routine.

5. Allocation of Local Variables and Registers

A total of ten registers are available for register variables. Six of these are the 68000 data (%d) registers, and four are the 68000 address (%a) registers. The available %a registers are %a2 through %a5. The available %d registers are %d2 through %d7.

Functions that return integers return their results in one of the data registers, whereas functions that return pointers return their result in an address register. In C, if a function is not declared in advance, it is assumed to return an integer. Unless the compiler is told otherwise, it will expect a funcition to return a value in a data register. If a function (such as **malloc**) returns a pointer, it MUST be declared, or the generated code will be wrong. Use the **lint** program to find places where functions have not been declared.

NOTE: The following instructions are NOT available in the current release of the assembler:

```
clr.w   %a0
clr.l   %a0
```

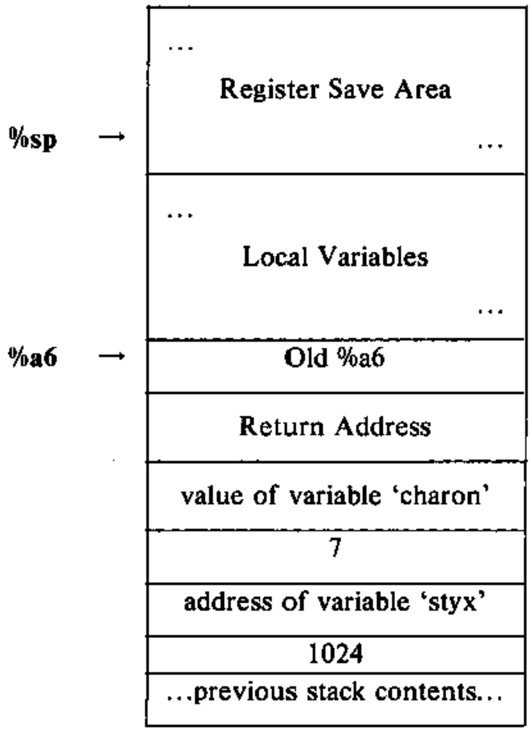
Use the following instructions instead:

```
                    mov.w   &0,%a0
                    mov.l   &0,%a0
```

Any variable declared as a pointer variable is always allocated to an address register. Non-pointer variables are assigned to data registers. Register variables are allocated to registers in the order in which they are declared in the C source program, starting at the low end (%a2 or %d2) of the appropriate type of register.

If there are more register variables of either kind than there are registers to accommodate them, the remaining variables are allocated on the stack as local variables, just as if the register attribute had never been given in the declaration.

Upon completion of the prolog code, the stack then looks like this:


...
Register Save Area
%sp →
...
...
Local Variables
...
%a6 →
Old %a6
Return Address
value of variable 'charon'
7
address of variable 'styx'
1024
...previous stack contents...

6. Returning from a Function or Subroutine

Upon reaching a "return" statement, either explicit or implicit, the function executes the epilog code. If the function has a return value, generated from a

```
return(expression);
```

statement, the value of the expression (which is synonymous with the value of the function) is placed in register %d0 or %a0 for pointer functions. The epilog code is then executed to effect a return from the function. The epilog code for both the 68000 and the 68010 looks like this:

```
moveml          S%1(%fp),&M%1
unlk            %fp
rts
```

The "moveml" instruction restores any registers which were saved during the prolog. The stack frame base pointer in %fp is then put back to the point where %fp once again points to the return address. The function is then exited via the "rts" instruction, which pops the stack to the state it was in prior to the original call, and then returns to the function that called it.

7. System Calls

The C compiler generates code for system calls in the following way:

- The system call number is placed in register %d0.
- A "TRAP #0" instruction is executed.

Parameters are passed on the user stack in the C calling convention. On return from the system call, errors are signaled by the carry flag being set. The C interface to the system calls typically returns a −1 on error as the carry flag cannot be tested from C.

8. Optimizations

This section describes some of the ways in which the programmer can optimize the use of the C language.

The C compiler can be run to optimize the code it generates, making that code both compact and fast. Using a C command line as follows:

```
cc −O file
```

generates optimized code. The option for optimized code generation is an upper-case "O".

If a C program contains a "do" loop of the form:

```
register short x;
x = 10;
do {
     statement
} while (--x != -1);
```

Such a loop is optimized to use the "dbra" instruction, resulting in faster execution.

The optimizer may work incorrectly near C code that includes a structure assignment. As a result, semantically correct C code will function incorrectly. The example below illustrates C code that the optimizer "breaks."

```
struct st { long i, j };
main()
{
     static struct st temp;
     struct st *p;
     temp = *p;
     p = &temp;
{
```

9. Use of Register Variables

The decision as to whether to declare a variable in a register depends on the number of times that variable is referenced in the function. If a variable is used more than twice in a function, it can be declared as a register variable. If a variable is used less than twice in a function, it is not useful to declare it as a register variable because the amount of time spent saving and restoring that register is more than the time saved in using a register instead of a location on the stack.

10. Miscellaneous Notes

The object files that the assembler and linker create use the Common Object File Format (COFF). See the chapter in the *UniPlus+ Programming Guide* entitled "COFF – THE COMMON OBJECT FILE FORMAT."

The C compiler will accept multiply-defined external variables as long as no more than one of the definitions includes an initialization.

The floating point emulation package is automatically invoked when the program is linked with the C library (**libc.a**).

The C compiler supports floating and double variables by making calls to a library of emulation routines. Although floating point data values are represented in IEEE standard floating point format, the emulation routines DO NOT implement the IEEE exception handling routines.

Chapter 2: THE UNIPLUS+ C COMPILER

CONTENTS

Chapter 2
THE UNIPLUS$^+$ C COMPILER

1. Introduction

This chapter describe the UniPlus$^+$ Operating System's C compiler, **cc**, and the C programming language that the compiler translates.

The C language is implemented for high-level programming and contains many control and structuring facilities that greatly simplify the task of algorithm construction. The C compiler prepares C programs which will ultimately be 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.

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

```
man cc
```

2. Use Of The Compiler

To use the compiler, first create a file (typically by using the UniPlus$^{+®}$ Operating System text editor) containing C source code. The last two characters (or *extention*), of the file name MUST be **.c**, for example, "*file1.c*."

```
cc options file.c
```

to invoke the compiler on the C source file *file.c* with the *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. If no controlling options are used, and only one file is named, **cc** automatically calls the assembler, **as**, and the link editor, **ld**, to produce the executable file, *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 using the **ld** command.

The **cc** compiler also accepts input file names with the last two characters **.s**. The **.s** extension 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 in a file of the same "base" name but with a **.o** extension (i.e., "file.s" → "file.o").

The program **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.

2.1 Options

All options recognized by the **cc** command are listed below:

RECOGNIZED AND EXECUTED BY cc

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.
−p	*none*	Reserved for invoking a profiler.
−t	*[p012al]*	Find only the designated preprocessor (*p*), compiler (*0* and *1*), optimizer (*2*), assembler (*a*) and link editor (*l*) passes whose names are constructed with the *string* argument to the **−B** option. In the absence of a **−B** option and its argument, *string* is taken to be **/lib/n**. The value of **−t** "" is equivalent to **−tp012**.
−B	*string*	Construct path names for substitute preprocessor, compiler, and link editor passes by concatenating *string* with the suffixes **cpp**, **c0** (or **ccom** or **comp**), **c1**, **c2** (or **optim**), **as** and **ld**. If *string* is empty it is taken to be **/lib/o**.
−E	*none*	Same as the **−P** option except output is directed to the standard output.
−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**.
−R	*none*	Have assembler remove its input file when done.
−T	*none*	Truncate identifier names to 8 significant characters.
−V	*none*	Print the version of the assembler that is invoked.
−W	*c,arg1[,arg2...]*	Pass the argument(s) *arg1* to *c*, where *c* is one of **[p012al]**, indicating preprocessor (**p**), compiler first pass (**0**), compiler second pass (**1**), optimizer (**2**), assembler (**a**) or link editor (**l**), respectively.
−X	*none*	Ignored by UniPlus+ for 68000.
−#	*none*	Special debug option which echoes the names and arguments of subprocesses which would have started without actually executing the program.

RECOGNIZED BY cc AND PASSED TO ld

OPTION	ARGUMENT	DESCRIPTION
−l	*x*	Same as **−l** in **ld(1)**. Search a library lib*x*.a, where *x* is up to seven characters. A library is

		searched when its name is encountered, so the placement of a −l is significant. By default, libraries are located in LIBDIR. If you plan to use the **−L** option, that option MUST PRECEDE −l on the command line.
−o	*outfile*	Same as **−o** in **ld(1)**. Produce an output object file called *outfile.* The name of the default object file is *a.out.*
−s	*none*	Same as **−s** in **ld(1)**. Strip line number entries and symbol table information from the output of object file.
−L	*dir*	Same as −L in **ld(1)**. Change the algorithm of searching for lib*x*.a to look in *dir* before looking in LIBDIR. This option is effective only if it precedes the **−l** option on the command line.

RECOGNIZED BY cc AND PASSED TO cpp

OPTION	ARGUMENT	DESCRIPTION
−C	*none*	Same as **−C** in **cpp(1)**. All comments, except those found on **cpp** directive lines are passed along. The default is that ALL comments are stripped out.

−D	*ident=def*	Same as **−D** in **cpp(1)**. Define the external symbol *ident* and give it the value *def* (if specified). If no *def* is given, *ident* is defined as 1.
−I	*dir*	Change the algorithm that searches for **#include** files whose names do not begin with / to look in the named *dir* before looking in the directories on the standard list. Thus, **#include** files whose names are enclosed in "" (i.e., #include "include file") 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 <> (i.e., #include <include file>), the directory of the *file* argument is not searched.
−U	*name*	Remove any initial definition of *name*, where *name* is a reserved symbol that is predefined by the particular preprocessor.

By using appropriate options, compilation can be terminated early to produce one of several intermediate translations such as:

(**−c** option) This option produces relocatable object files.

It is often desirable to use this option to save relocatable files so that changes to one file do not then require that the others be recompiled. A separate call to **cc** with the relocatable files (indicated by a **.c** extention, as in *file.c*), but without the **−c** option, creates the linked executable **a.out** file. A relocatable object file created under the **−c** option has the same root as the relocatable object file, but the extention is **.o** instead of **.c**.

(**−S** option)	This option produces assembly source expansions for C code.
(**−P** option)	This option produces the output of the preprocessor. When used, the compilation process stops after preprocessing. Output from the preprocessor is left in an outfile with an extension **.i**, for example, *file.i*. These output files can be subsequently processed by **cc** but only if their file name is changed to a name ending in "**.c**".

Except for those produced by the preprocessor, intermediate files may be saved and resubmitted to the **cc** command, with other files or libraries included as necessary.

The **−W** option provides the mechanism to specify options for each step that is normally invoked from the **cc** command line. These steps are:

1. preprocessing,
2. the first pass of the compiler,
3. the second pass of the compiler,
4. optimization,
5. assembly, and
6. link editing.

At this time, only assembler and link editor options can be used with the **−W** option. The most common example of the **−W** option is

−W *l*, *−VS,n*

which passes the **−VS n** option to the link editor (**ld(1)**). In the following example:

−W *a*, *−option*

the compiler will pass the *−option* to the assembler.

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. (For more information see the chapter entitled "SDB − A SYMBOLIC DEBUGGER" in this manual.)

For more information on any of the options which are passed by **cc(1)** to either the preprocessor **cpp(1)** or the link editor **ld(1)**, see the appropriate manual page in the *UniPlus*+ *User Manual*.

Chapter 3: THE C LANGUAGE

CONTENTS

LIST OF FIGURES

Chapter 3
THE C LANGUAGE

1. Lexical Conventions

There are six classes of tokens:

1. **identifiers**
2. **keywords**
3. **constants**
4. **strings**
5. **operators**
6. **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.

1.1 Comments

The characters /* introduce a comment which terminates with the characters */.

```
/* COMMENTS /* DO NOT */ NEST */
```

1.2 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. For the 68000:

68000	7 characters, 2 cases

1.3 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**	**while**
continue	**external**	**long**	**struct**	
default	**float**	**register**	**switch**	

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

1.4 Constants

There are several kinds of constants, each of which has a type. The introduction to types is given in the section entitled "Names." Hardware characteristics that affect sizes are summarized in the subsection "Hardware Characteristics" under the general heading "Lexical Conventions."

1.4.1 Integer Constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with a zero. An octal constant consists of the digits **0** through 7 only. A sequence of digits preceded by **0x** or **0X** is taken to be a hexadecimal integer. The hexadecimal digits include a through f (or A through 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**.

1.4.2 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.

1.4.3 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	\0[*0-7*][*0-7*]	\0[*0-7*][*0-7*]

The escape \0[*0-7*][*0-7*] consists of the backslash followed by 1, 2, or 3 octal digits (0 through 7) 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 NULL. 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**.

1.4.4 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 may be missing — but NOT BOTH. Either the decimal point OR the **e** and exponent may be missing — but NOT BOTH. Every floating constant has type **double**.

1.4.5 Enumeration Constants

Names declared as enumerators have type **int**. For more information see the section entitled "Structure, Union and Enumeration Declarations."

1.5 Strings

A string is a sequence of characters surrounded by double quotes, as in "string". A string has type "array of char" and storage class static 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.

1.6 Hardware Characteristics

The following figures summarize certain hardware properties for the 68000.

68000 (ASCII)	
char	8 bits
int	32
short	16
long	32
float	32
double	64
float range	$\pm 10^{\pm 38}$
double range	$\pm 10^{\pm 307}$

Figure 3.1. 68000 Hardware Characteristics

2. Syntax Notation

Syntactic categories are indicated by *ITALIC* type, commands in **BOLD** type and other literal words and characters in ROMAN 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.

3. Names

The C language bases the interpretation of an identifier upon two attributes of the identifier:

1. **storage class**	The storage class determines the location and lifetime of the storage associated with an identifier.
2. **type**	the type determines the meaning of the values found in the identifier's storage.

3.1 Storage Class

There are four declarable storage classes:

1. **Automatic**	Automatic variables are local to each invocation of a block and are discarded upon exit from the block.
2. **Static**	Static variables are local to a block but retain their values upon reentry to a block even after control has left the block.
2. **External**	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.
2. **Register**	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.

3.2 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 3short **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 are identical to those of some integer types. The implementation may use the range of values to determine how to allot storage.

Unsigned integers, declared **unsigned**, obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation.

Single-precision floating point (**float**) and double precision floating point (**double**) may be synonymous in some implementations.

Because objects of the foregoing types can usefully be interpreted as numbers, they will be referred to as **arithmetic** types. **Char**, **int** of all sizes whether **unsigned** or not, and **enum** will collectively be called *integral* types. The **float** and **double** types will collectively be called *floating* types.

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.

4. 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.

5. 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 later in this section under the subheading "Arithmetic Conversions." The summary will be supplemented as required by the discussion of each operator.

5.1 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.

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.

5.2 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.

5.3 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.

5.4 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.

5.5 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.

5.6 Arithmetic Conversions

A great many operators cause conversions and yield result types in a similar way. From now on in this document 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.

6. 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 + are those expressions defined under the sections "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.

6.1 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."

6.2 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**.

6.3 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).

6.4 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.

6.5 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.

6.6 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, and 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.

6.7 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.

6.8 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.

6.9 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.

6.10 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.

6.11 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**.

6.12 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**.

6.13 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.

6.14 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.

6.15 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.

7. 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.

7.1 Storage Class Specifiers

The sc-specifiers are:

> **auto**
> **static**
> **extern**
> **register**
> **typedef**

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

The **auto**, **static**, and **register** declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the **extern** case, there must be an external definition (see "External Definitions") for the given identifiers somewhere outside the function in which they are declared.

A **register** declaration is best thought of as an **auto** declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few such declarations in each function are effective. Moreover, only variables of certain types will be stored in registers. One other restriction applies to 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.

7.2 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
```

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."

7.3 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.

7.3.1 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:

int i	an integer **i**
***ip**	a pointer to an integer,
f()	a function returning an integer,
***fip()**	a function returning a pointer to an integer
(*pfi)()	a pointer 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**.

7.4 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.

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.

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:

1. when a pointer to a structure or union is being declared, and
2. 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**.

7.5 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}.

7.6 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.

7.7 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*
    tract-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	is type integer
int *	is type pointer to integer
int *[3]	is type array of three pointers to integers
int (*)[3]	is type pointer to an array of three integers
int *()	is type function returning pointer to integer
int (*)()	is type pointer to function returning an integer
int (*[3])()	is type array of three pointers to functions returning an integer

7.8 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**; that of **z** is the specified structure **complex** and that of **zp** is **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.

8. Statements

Except as indicated, statements are executed in sequence.

8.1 Expression Statement

Most statements are expression statements, which have the form

expression;

Usually expression statements are assignments or function calls.

8.2 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.

8.3 Conditional Statement

The two forms of the conditional statement are

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

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

8.4 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.

8.5 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.

8.6 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.

8.7 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 are 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.

8.8 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.

8.9 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 (...)
{                {                 {
  ...              ...               ...
  continue ;       continue ;        continue ;
}                } while (...);    }
```

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

8.10 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.

8.11 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.

8.12 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."

8.13 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**.

9. 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.

9.1 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 ...."

9.2 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**.

10. 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:

1. *lexical scope* — is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics.
2. *scope of externals* — 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.

10.1 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.**

10.2 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**.

11. 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.

11.1 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 **#defined** identifier is the subject of a subsequent **#define** with no intervening **#undef**, then the two token-strings are compared textually. If the two token-strings are not identical (all white space is considered as equivalent), then the identifier is considered to be redefined.

11.2 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.

11.3 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.

11.4 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.

12. 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**."

13. Types Revisited

This part summarizes the operations which can be performed on objects of certain types.

13.1 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) ...
```

13.2 Functions

There are only two things that can be done with a function - call it or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```
int f();
...
g(f);
```

Then the definition of **g** might read

```
g(funcp)
        int (*funcp)();
{
        ...
        (*funcp)();
        ...
}
```

Notice that **f** must be declared explicitly in the calling routine since its appearance in **g(f)** was not followed by **(**.

13.3 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.

13.4 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.

On the 68000, pointers are 32-bits long and measure bytes. The **char**'s have no alignment requirements; everything else must have an even address.

14. 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.

15. 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.

16. Syntax Summary

This summary of C syntax is intended more for aiding comprehension than as an exact statement of the language.

16.1 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:
```
=   +=   mi=   *=   /=   %=   >>=   <<=   &=   ^=   |
```

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

16.2 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

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*

16.3 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
;

16.4 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* ;

16.5 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 4: C LIBRARIES

CONTENTS

Chapter 4
C LIBRARIES

1. Introduction

This chapter describes the UNIX Operating System C library. A *library* is a collection of related functions and/or declarations that simplify programming effort by linking what is needed, allowing use of locally produced functions, etc. All of the functions described in this chapter are also described in Section 3 of the *UniPlus+ User Manual.* Most of the declarations described in this chpater are also described in Section 5 of the *UniPlus+ User Manual.* The three main libraries on the UNIX system are:

C library	This is the basic library for C language programs. The C library is composed of functions and declarations used for file access, string testing and manipulation, character testing and manipulation, memory allocation, and other functions. This library is described later in this chapter.
Object file	This library provides functions for the access and manipulation of object files. This library is described in the next chapter.
Math library	This library provides exponential, bessel functions, logarithmic, hyperbolic, and trigonometric functions. This library is described in the next chapter.

Some libraries consist of two portions — functions and declarations. In some cases, the user must request that the functions (and/or declarations) of a specific library be included in a program being compiled. In other cases, the functions (and/or declarations) are included automatically.

2. Including Functions

When a program is being compiled, the compiler will automatically search the C language library to locate and include functions that are used in the program. This is the case only for the C library and no other library. In order for the compiler to locate and include functions from other libraries, the user must specify these libraries on the command line for the compiler. For example, when using functions of the math library, the user must request that the math library be searched by including the argument **−lm** on the command line:

```
cc file.c  −lm
```

The argument **−lm** must come after all files that reference functions in the math library in order for the link editor to know which functions to include in the a.out file.

This method should be used for all functions that are not part of the C language library.

3. Including Declarations

Some functions require a set of declarations in order to operate properly. A set of declarations is stored in a file under the */usr/include* directory. These files are referred to as *header files.* In order to include a certain header file, the user must specify this near the top of the file containing the program:

```
#include <file.h>
```

where *file.h* is the name of the header file. Since the header files define the type of the functions and various preprocessor constants, they must be included BEFORE invoking the functions they declare.

The remainder of this chapter descri›es the functions and header files of the C Library. The description of the library begins with the actions required by the user to include the functions and/or header files in a program being compiled (if any). Following the description of the actions required is information in three-column format:

function	**reference**(N)	Brief description.

The functions are grouped by type and the reference refers to section "N" in the *UniPlus+ User Manual.* Following this, if applicable, are descriptions of the header files associated with these functions.

4. The C Library

The C library consists of several types of functions. All the functions of the C library are loaded automatically by the compiler. Various declarations must sometimes be included by the user. The functions of the C library are divided into the following types:

- Input/output control
- String manipulation
- Character manipulation
- Time functions
- Miscellaneous functions.

4.1 Input/Output Control

These functions of the C library are automatically included as needed during the compiling of a C language program. No command line request is needed.

The header file required by the input/output functions should be included near the beginning of each file that references an input or output function:

#include < *stdio.h*>

The input/output functions are grouped into the following categories:

- File access
- File status
- Input
- Output
- Miscellaneous.

4.1.1 File Access Functions

FUNCTION	REFERENCE	BRIEF DESCRIPTION
fclose	**fclose**(3S)	Close an open stream.
fdopen	**fopen**(3S)	Associate stream with an **open**(2) ed file.
fileno	**ferror**(3S)	File descriptor associated with an open stream.
fopen	**fopen**(3S)	Open a file with specified permissions and return a pointer to a stream which is used in subsequent references to the file.
freopen	**fopen**(3S)	Substitute named file in place of open stream.
fseek	**fseek**(3S)	Reposition the file pointer.
pclose	**popen**(3S)	Close a stream opened by **popen**.
popen	**popen**(3S)	Create pipe as a stream between calling process and command.
rewind	**fseek**(3S)	Reposition file pointer at beginning of file.

setbuf	**setbuf(3S)**	Assign buffering to stream.
vsetbuf	**setbuf(3S)**	Similar to **setbuf**, but allowing finer control.

4.1.2 File Status Functions

FUNCTION	REFERENCE	BRIEF DESCRIPTION
clearerr	**ferror(3S)**	Reset error condition on stream.
feof	**ferror(3S)**	Test for "end of file" (EOF) on stream.
ferror	**ferror(3S)**	Test for error condition on stream.
ftell	**fseek(3S)**	Return current position in the file.

4.1.3 Input Functions

FUNCTION	REFERENCE	BRIEF DESCRIPTION
fgetc	**getc(3S)**	True function for **getc** (3S).
fgets	**gets(3S)**	Read string from stream.
fread	**fread(3S)**	General buffered read from stream.
fscanf	**scanf(3S)**	Formatted read from stream.

getc	**getc(3S)**	Read character from stream.
getchar	**getc(3S)**	Read character from standard input.
gets	**gets(3S)**	Read string from standard input.
getw	**getc(3S)**	Read word from stream.
scanf	**scanf(3S)**	Read using format from standard input.
sscanf	**scanf(3S)**	Formatted from string.
ungetc	**ungetc(3S)**	Put back one character on stream.

4.1.4 Output Functions

FUNCTION	REFERENCE	BRIEF DESCRIPTION
fflush	**fclose(3S)**	Write all currently buffered characters from stream.
fprintf	**printf(3S)**	Formatted write to stream.
fputc	**putc(3S)**	True function for **putc** (3S).
fputs	**puts(3S)**	Write string to stream.
fwrite	**fread(3S)**	General buffered write to stream.

printf	**printf**(3S)	Print using format to standard output.
putc	**putc**(3S)	Write character to standard output.
putchar	**putc**(3S)	Write character to standard output.
puts	**puts**(3S)	Write string to standard output.
putw	**putc**(3S)	Write word to stream.
sprintf	**printf**(3S)	Formatted write to string.
vfprintf	**vprint**(3C)	Print using format to stream by **varargs**(5) argument list.
vprintf	**vprint**(3C)	Print using format to standard output by **varargs**(5) argument list.
vsprintf	**vprintf**(3C)	Print using format to stream string by **varargs**(5) argument list.

4.1.5 Miscellaneous Functions

FUNCTION	REFERENCE	BRIEF DESCRIPTION
ctermid	**ctermid**(3S)	Return file name for controlling terminal.
cuserid	**cuserid**(3S)	Return login name for

		owner of current process.
system	**system(3S)**	Execute shell command.
tempnam	**tmpnam(3S)**	Create temporary file name using directory and prefix.
tmpnam	**tmpnam(3S)**	Create temporary file name.
tmpfile	**tmpfile(3S)**	Create temporary file.

4.2 String Manipulation Functions

These functions are used to locate characters within a string or to copy, concatenate, or compare strings. These functions are automatically located and loaded during the compiling of a C language program. No command line request is needed since these functions are part of the C library. The string manipulation functions are declared in a header file that should be included near the beginning of each file that uses any of these functions:

#include < *string.h*>

FUNCTION	REFERENCE	BRIEF DESCRIPTION
strcat	**string(3C)**	Concatenate two strings.
strchr	**string(3C)**	Search string for character.
strcmp	**string(3C)**	Compares two strings.
strcpy	**string(3C)**	Copy string.
strcspn	**string(3C)**	Length of initial string

		not containing set of characters.
strlen	string(3C)	Length of string.
strncat	string(3C)	Concatenate two strings with a maximum length.
strncmp	string(3C)	Compares two strings with a maximum length.
strncpy	string(3C)	Copy string over string with a maximum length.
strpbrk	string(3C)	Search string for any set of characters.
strrchr	string(3C)	Search string backwards for character.
strspn	string(3C)	Length of initial string containing set of characters.
strtok	string(3C)	Search string for token separated by any of a set of characters.

4.3 Character Manipulation

The following functions and declarations are used for testing and translating ASCII characters. These functions are located and loaded automatically during the compiling of a C language program. No command line request is needed since these functions are part of the C library.

The declarations associated with these functions should be included near the beginning of the file being compiled:

#include < *ctype.h*>

4.3.1 Character Testing Functions

These functions can be used to identify characters as uppercase or lowercase letters, digits, punctuation, etc.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
isalnum	ctype(3C)	Is character alphanumeric?
isalpha	ctype(3C)	Is character alphabetic?
isascii	ctype(3C)	Is integer ASCII character?
iscntrl	ctype(3C)	Is character a control character?
isdigit	ctype(3C)	Is character a digit?
isgraph	ctype(3C)	Is character a printable character?
islower	ctype(3C)	Is character a lowercase letter?
isprint	ctype(3C)	Is character a printing character including space?
ispunct	ctype(3C)	Is character a punctuation character?
isspace	ctype(3C)	Is character a white space character?

isupper	**ctype**(3C)	Is character an uppercase letter?
isxdigit	**ctype**(3C)	Is character a hex digit?

4.3.2 Character Translation Functions

These functions provide translation of uppercase to lowercase, lowercase to uppercase, and integer to ASCII.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
toascii	**conv**(3C)	Convert integer to ASCII character.
tolower	**conv**(3C)	Convert character to lowercase.
toupper	**conv**(3C)	Convert character to uppercase.

4.4 Time Functions

These functions are used for accessing and reformatting the system's idea of the current date and time. These functions are located and loaded automatically during the compiling of a C language program. No command line request is needed since these functions are part of the C library.

The header file associated with these functions should be included near the beginning of any file using the time functions.

#include < *time.h*>

These functions (except **tzset**) convert a time such as returned by **time**(2)

FUNCTION	REFERENCE	BRIEF DESCRIPTION
asctime	**ctime(3C)**	Return string representation of date and time.
ctime	**ctime(3C)**	Return string representation of date and time, given integer form.
gmtime	**ctime(3C)**	Return Greenwich Mean Time.
localtime	**ctime(3C)**	Return local time.
tzset	**ctime(3C)**	Set time zone field from environment variable.

4.5 Miscellaneous Functions

These functions support a wide variety of operations:

- Numerical Conversion
- DES Algorithm Access
- Group File Access
- Password File Access
- Parameter Access
- Hash Table Management
- Binary Tree Management
- Table Management
- Memory Allocation

- Pseudorandom Number Generation

These functions are automatically located and included in a program being compiled. No command line request is needed since these functions are part of the C library.

Some of these functions require declarations to be included. These are described following the descriptions of the functions.

4.5.1 Numerical Conversion

The following functions perform numerical conversion.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
a64l	**a64l(3C)**	Convert string to base 64 ASCII.
atof	**atof(3C)**	Convert string to floating.
atoi	**atof(3C)**	Convert string to integer.
atol	**atof(3C)**	Convert string to long.
frexp	**frexp(3C)**	Split floating into mantissa and exponent.
l3tol	**l3tol(3C)**	Convert 3-byte integer to long.
ltol3	**l3tol(3C)**	Convert long to 3-byte integer.

ldexp	**frexp(3C)**	Combine mantissa and exponent.
l64a	**a64l(3C)**	Convert base 64 ASCII to string.
modf	**frexp(3C)**	Split mantissa into integer and fraction.

4.5.2 DES Algorithm Access

The following functions allow access to the Data Encryption Standard (DES) algorithm used on the UNIX operating system. The DES algorithm is implemented with variations to frustrate use of hardware implementations of the DES for key search.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
crypt	**crypt(3C)**	Encode string.
encrypt	**crypt(3C)**	Encode/decode string of 0s and 1s.
setkey	**crypt(3C)**	Initialize for subsequent use of **encrypt**.

4.5.3 Group File Access

The following functions are used to obtain entries from the group file. Declarations for these functions must be included in the program being compiled with the line:

#include <*grp.h*>

FUNCTION	REFERENCE	BRIEF DESCRIPTION
endgrent	**getgrent**(3C)	Close group file being processed.
getgrent	**getgrent**(3C)	Get next group file entry.
getgrgid	**getgrent**(3C)	Return next group with matching gid.
getgrnam	**getgrent**(3C)	Return next group with matching name.
setgrent	**getgrent**(3C)	Rewind group file being processed.
fgetgrent	**getgrent**(3C)	Get next group file entry from a specified file.

4.5.4 Password File Access

These functions are used to search and access information stored in the password file (*/etc/passwd*). Some functions require declarations that can be included in the program being compiled by adding the line:

#include <*pwd.h*>

FUNCTION	REFERENCE	BRIEF DESCRIPTION
endpwent	**getpwent**(3C)	Close password file being processed.

getpw	**getpw**(3C)	Search password file for uid.
getpwent	**getpwent**(3C)	Get next password file entry.
getpwnam	**getpwent**(3C)	Return next entry with matching name.
getpwuid	**getpwent**(3C)	Return next entry with matching uid.
putpwent	**putpwent**(3C)	Write entry on stream.
setpwent	**getpwent**(3C)	Rewind password file being accessed.
fgetpwent	**getpwent**(3C)	Get next password file entry from a specified file.

4.5.5 Parameter Access

The following functions provide access to several different types of parameters. None require any declarations.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
getopt	**getopt**(3C)	Get next option from option list.
getcwd	**getcwd**(3C)	Return string representation of current working directory.

getenv	**getenv(3C)**	Return string value associated with environment variable.
getpass	**getpass(3C)**	Read string from terminal without echoing.
putenv	**putenv(3C)**	Change or add value of an environment variable.

4.5.6 Hash Table Management

The following functions are used to manage hash search tables. The header file associated with these functions should be included in the program being compiled. This is accomplished by including the line:

#include < *search.h*>

near the beginning of any file using the search functions.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
hcreate	**hsearch(3C)**	Create hash table.
hdestroy	**hsearch(3C)**	Destroy hash table.
hsearch	**hsearch(3C)**	Search hash table for entry.

4.5.7 Binary Tree Management

The following functions are used to manage a binary tree. The header file associated with these functions should be included near the beginning of any file using the search functions:

#include < *search.h*>

FUNCTION	REFERENCE	BRIEF DESCRIPTION
tdelete	**tsearch(3C)**	Deletes nodes from binary tree.
tfind	**tsearch(3C)**	Find element in binary tree.
tsearch	**tsearch(3C)**	Look for and add element to binary tree.
twalk	**tsearch(3C)**	Walk binary tree.

4.5.8 Table Management

The following functions are used to manage a table. Since none of these functions allocate storage, sufficient memory must be allocated before using these functions. The header file associated with these functions should be included near the beginning of any file using the search functions:

#include < *search.h*>

FUNCTION	REFERENCE	BRIEF DESCRIPTION
bsearch	**bsearch(3C)**	Search table using binary search.
lsearch	**lsearch(3C)**	Look for and add element in binary tree.
lfind	**lsearch(3C)**	Find element in library tree.

qsort	**qsort(3C)**	Sort table using quick-sort algorithm.

4.5.9 Memory Allocation

The following functions provide a means by which memory can be dynamically allocated or freed.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
calloc	**malloc(3C)**	Allocate zeroed storage.
free	**malloc(3C)**	Free previously allocated storage.
malloc	**malloc(3C)**	Allocate storage.
realloc	**malloc(3C)**	Change size of allocated storage.

The following is another set of memory allocation functions available.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
calloc	**malloc(3X)**	Allocate zeroed storage.
free	**malloc(3X)**	Free previously allocated storage.
malloc	**malloc(3X)**	Allocate storage.
mallopt	**malloc(3X)**	Control allocation algorithm.

mallinfo	**malloc(3X)**	Space usage.
realoc	**malloc(3X)**	Change size of allocated storage.

4.5.10 Pseudorandom Number Generation

The following functions are used to generate pseudorandom numbers. The functions that end with **48** are a family of interfaces to a pseudorandom number generator based upon the linear congruent algorithm and 48-bit integer arithmetic. The **rand** and **srand** functions provide an interface to a multiplicative congruential random number generator with period of 232.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
drand48	**drand48(3C)**	Random double over the interval [0 to 1).
lcong48	**drand48(3C)**	Set parameters for **drand48**, **lrand48**, and **mrand48**.
lrand48	**drand48(3C)**	Random long over the interval [0 to 2^{31}).
mrand48	**drand48(3C)**	Random long over the interval [-2^{31} to 2^{31}).
rand	**rand(3C)**	Random integer over the interval [0 to 32767).
seed48	**drand48(3C)**	Seed the generator for **drand48**, **lrand48**, and **mrand48**.

srand	**rand**(3C)	Seed the generator for **rand**.
srand48	**drand48**(3C)	Seed the generator for **drand48**, **lrand48**, and **mrand48** using a long.

4.5.11 Signal Handling Functions

The functions **gsignal** and **ssignal** implement a software facility similar to **signal**(2) in the *UniPlus+ User Manual.* This facility enables users to indicate the disposition of error conditions and allows users to handle signals for their own purposes. The declarations associated with these functions should be included near the beginning of any file using the signal handling functions:

```
#include <signal.h>
```

These declarations define ASCII names for the 15 software signals.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
gsignal	**ssignal**(3C)	Send a software signal.
ssignal	**ssignal**(3C)	Arrange for handling of software signals.

4.5.12 Miscellaneous

The following functions do not fall into any previously described category.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
abort	**abort(3C)**	Cause an IOT signal to be sent to the process.
abs	**abs(3C)**	Return the absolute integer value.
ecvt	**ecvt(3C)**	Convert double to string.
fcvt	**ecvt(3C)**	Convert double to string using Fortran Format.
gcvt	**ecvt(3C)**	Convert double to string using Fortran F or E format.
isatty	**ttyname(3C)**	Test whether integer file descriptor is associated with a terminal.
mktemp	**mktemp(3C)**	Create file name using template.
monitor	**monitor(3C)**	Cause process to record a histogram of program counter location.
swab	**swab(3C)**	Swap and copy bytes.

ttyname	**ttyname(3C)**	Return pathname of terminal associated with integer file descriptor.

Chapter 5: OBJECT AND MATH LIBRARIES

CONTENTS

Chapter 5
OBJECT AND MATH LIBRARIES

1. Introduction

This chapter describes the UniPlus+® Object and Math Libraries. A library is a collection of related functions and/or declarations that simplify programming effort. All of the functions described are also described in Section 3 of the *UniPlus+ User Manual.* Most of the declarations described in this chapter can be found in Section 5 of the *UniPlus+ User Manual.*

The three main libraries of the UniPlus+ Operating System are:

C library	This is the basic library for C language programs. The C library is composed of functions and declarations used for file access, string testing and manipulation, character testing and manipulation, memory allocation, and other functions. This library is described in the chapter entitled "THE C LIBRARY," in the *UniPlus+ Programming Guide.*
Object file	This library provides functions for the access and manipulation of object files. This library is described later in this chapter.
Math library	This library provides exponential, bessel functions, logarithmic, hyperbolic, and trigonometric functions. This library is described in more detail later in this chapter.

2. Object File Library

The object file library provides functions for the access and manipulation of object files. Some functions locate portions of an object file such as the symbol table, the file header, sections, and line number entries associated with a function. Other functions read these types of entries into memory. For a description of the format of an object file, see the chapter entitled "COFF – COMMON OBJECT FILE FORMAT" in the *UniPlus+ Programming Guide.*

The object file library functions reside in */usr/lib/libld.a* and may be located and loaded at compile time if the following command-line request is given:

```
cc file −lld
```

This command causes the link editor to search the object file library. The argument −lld must appear AFTER all files that reference functions in *libld.a.*

In addition, various header files must be included:

```
#include <stdio.h>
#include <a.out.h>
#include <ldfcn.h>
```

FUNCTION	REFERENCE	BRIEF DESCRIPTION
ldaclose	**ldclose**(3X)	Close object file being processed.
ldahread	**ldahread**(3X)	Read archive header.
ldaopen	**ldopen**(3X)	Open object file for reading.
ldclose	**ldclose**(3X)	Close object file being processed.
ldfhread	**ldfhread**(3X)	Read file header of object file being processed.

ldgetname	**ldgetname**(3X)	Retrieve the name of an object file symbol table entry.
ldlinit	**ldlread**(3X)	Prepare object file for reading line number entries via **ldlitem**.
ldlitem	**ldlread**(3X)	Read line number entry from object file after **ldlinit**.
ldlread	**ldlread**(3X)	Read line number entry from object file.
ldlseek	**ldlseek**(3X)	Seeks to the line number entries of the object file being processed.
ldnlseek	**ldlseek**(3X)	Seeks to the line number entries of the object file being processed given the name of a section.
ldnrseek	**ldrseek**(3X)	Seeks to the relocation entries of the object file being processed given the name of a section.
ldnshread	**ldshread**(3X)	Read section header of the named section of the object file being processed.
ldnsseek	**ldsseek**(3X)	Seeks to the section of the object file being processed given the name of a section.

ldohseek	**ldohseek(3X)**	Seeks to the optional file header of the object file being processed.
ldopen	**ldopen(3X)**	Open object file for reading.
ldrseek	**ldrseek(3X)**	Seeks to the relocation entries of the object file being processed.
ldshread	**ldshread(3X)**	Read section header of an object file being processed.
ldsseek	**ldsseek(3X)**	Seeks to the section of the object file being processed.
ldtbindex	**ldtbindex(3X)**	Returns the long index of the symbol table entry at the current position of the object file being processed.
ldtbread	**ldtbread(3X)**	Reads a specific symbol table entry of the object file being processed.
ldtbseek	**ldtbseek(3X)**	Seeks to the symbol table of the object file being processed.
sgetl	**sputl(3X)**	Access long integer data in a machine independant format.

sputl	**sputl(3X)**	Translate a long integer into a machine independant format.

2.1 Common Object File Interface Macros (ldfcn.h)

The interface between the calling program and the object file access routines is based on the defined type "LDFILE," which is defined in the header file *ldfcn.h* (see **ldfcn(4)**). The primary purpose of this structure is to provide uniform access to both simple object files and to object files that are members of an archive file.

The function **ldopen(3X)** allocates and initializes the LDFILE structure and returns a pointer to that structure to the calling program. The fields of the LDFILE structure may be accessed individually through the following macros:

1. the **type** macro returns the magic number of the file, which is used to distinguish between archive files and simple object files.
2. The **IOPTR** macro returns the file pointer which was opened by **ldopen(3X)** and is used by the input/output functions of the C library.
3. The **OFFSET** macro returns the file address of the beginning of the object file. This value is non-zero only if the object file is a member of the archive file.
4. The **HEADER** macro accesses the file header structure of the object file.

Additional macros are provided to access an object file. These macros parallel the input/output functions in the C library; each macro translates a reference to an LDFILE structure into a reference to its file descriptor field. The available macros are described in **ldfcn(4)** in the *UniPlus+ User Manual.*

3. Math Library

The math library consists of functions and a header file. The functions may be located and loaded during compile-time if a request is made on the command line:

```
cc file −lm
```

This command will cause the link editor to search the math library. In addition to the request to load the functions, the header file of the math library should be included near the beginning of the (first) file being compiled:

```
#include <math.h>
```

These functions are grouped into the following categories:

- Trigonometric functions
- Bessel functions
- Hyperbolic functions
- Miscellaneous functions.

3.1 Trigonometric Functions

These functions are used to compute angles (in radian measure), sines, cosines, and tangents. All of these values are expressed in double precision.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
acos	**trig(3M)**	Return arc cosine.
asin	**trig(3M)**	Return arc sine.
atan	**trig(3M)**	Return arc tangent.
atan2	**trig(3M)**	Return arc tangent of

		a ratio.
cos	**trig(3M)**	Return cosine.
sin	**trig(3M)**	Return sine.
tan	**trig(3M)**	Return tangent.

3.2 Bessel Functions

These functions calculate bessel functions of the first and second kinds of several orders for real values. The bessel functions are **j0**, **j1**, **jn**, **y0**, **y1**, and **yn**. The functions are located in section **bessel(3M)**.

3.3 Hyperbolic Functions

These functions are used to compute the hyperbolic sine, cosine, and tangent for real values.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
cosh	**sinh(3M)**	Return hyperbolic cosine.
sinh	**sinh(3M)**	Return hyperbolic sine.
tanh	**sinh(3M)**	Return hyperbolic tangent.

3.4 Miscellaneous Functions

These functions cover a wide variety of operations, such as natural logarithm, exponential, and absolute value. In addition, several are provided to truncate the integer portion of double precision numbers.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
ceil	**floor(3M)**	Returns the smallest integer not less than a given value.
exp	**exp(3M)**	Returns the exponential function of a given value.
fabs	**floor(3M)**	Returns the absolute value of a given value.
floor	**floor(3M)**	Returns the largest integer not greater than a given value.
fmod	**floor(3M)**	Returns the remainder produced by the division of two given values.
gamma	**gamma(3M)**	Returns the natural log of the absolute value of the result of applying the gamma function to a given value.
hypot	**hypot(3M)**	Return the square root of the sum of the squares of two numbers.
log	**exp(3M)**	Returns the natural logarithm of a given value.
log10	**exp(3M)**	Returns the logarithm base ten of a given value.
matherr	**matherr(3M)**	Error-handling function.

pow	**exp(3M)**	Returns the result of a given value raised to another given value.
sqrt	**exp(3M)**	Returns the square root of a given value.

In addition, various header files must be included. This is accomplished by including the line:

> **#include** < *stdio.h*>
> **#include** < *a.out.h*>
> **#include** < *ldfcn.h*>

Chapter 6: ASSEMBLER

CONTENTS

LIST OF FIGURES

Chapter 6

MOTOROLA 68000 ASSEMBLER

1. Introduction

This is a reference manual for the UniPlus$^{+®}$ System assembler for the Motorola 68000.

Programmers familiar with the M68000 should be able to program in the assembler by referring to this manual, but this is not a manual for the processor itself. Details about the effects of instructions, meanings of status register bits, handling of interrupts, and many other issues are not dealt with in detail in this document. This manual should be used in conjunction with the **M68000 16-bit Microprocessor User's Manual.**[*]

2. Warnings

A few important warnings should be emphasized at the outset.

For the most part there is a direct correspondence between **as** notation used here and the notation used in **M68000 16-bit Microprocessor User's Manual**. However the following exceptions could lead the unsuspecting user to write incorrect code.

2.1 Compare and Subtract

The order of the operands in compare, **cmp**, instructions follows one convention in the **M68000 16-bit Microprocessor User's Manual**, and the OPPOSITE convention in the UniPlus$^+$ **as**.

[*] **M68000 16-BIT MICROPROCESSOR User's Manual**, Third Edition; Englewood Cliffs, N.J.: Prentice-Hall, 1982.

Using the convention in the **M68000 16-bit Microprocessor User's Manual**, one might write

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

Using the as convention, one would write

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

The UniPlus+ **as** follows the convention used by other assemblers supported in the UNIX™ System (both the 3B20S™ and the VAX™ follow this convention). This convention makes for straightforward reading of **compare-and-branch** instruction sequences. However it does lead to the following peculiarity:

— If a **cmp** instruction is replaced by a **sub** instruction, the effect on the condition codes will be entirely different.

This peculiarity may be especially confusing to programmers who are used to thinking of a comparison as a subtraction whose result is not stored.

2.2 Opcode Overloading

Another issue that users must be aware of arises from the M68000's use of several different instructions to do more or less the same thing.

For example, the **M68000 16-bit Microprocessor User's Manual** lists the instructions **SUB**, **SUBA**, **SUBI** and **SUBQ**, which all have the effect of subtracting their source operand from their destination operand. **as** provides the convenience of allowing all these operations to be specified by a single assembly instruction **sub**. On the basis of the operands given to the **sub** instruction, the **as** assembler selects the appropriate M68000 operation code.

It is important to remember that, even though **sub** can be used, it could leave the misleading impression that all forms of the **SUB** operation are semantically identical — however, they are NOT.

The careful reader of the **M68000 16-bit Microprocessor User's Manual** will notice that while **SUB**, **SUBI**, and **SUBQ** all affect the condition codes in a consistent way, **SUBA** DOES NOT affect the condition codes at all. Consequently, the **as** user must be aware that when the destination of a **sub** instruction is an address register (which causes the **sub** to be mapped into the operation code for **SUBA**), the condition codes will NOT be affected.

3. General Programming Information

3.1 Privilege States

Instructions may be executed in either of two distinct modes or Privilege States:

1. User	This mode is reserved for the execution of most application programs.
2. Supervisor	This mode is reserved for use by the operating system and other system software; To execute some instructions, called "Privileged Instructions," the processor MUST be in supervisor mode.

3.2 Data Organization

All of the M68000 registers, both data and address registers, are 32-bits wide. In the data registers, the low order 8 bits are used by byte operands, the low order 16 bits accomodate word operands and the entire register is used when the operand is a long word. The 0 bit is the "least significant bit" (**lsb**); while bit 31 is the "most significant bit" (**msb**).

Byte-sized operations may not be performed on address registers. When the source operand is an address register, either the low order word or the entire register may be used, depending on the operation size. However, regardless of the operation, when an address register is the destination operand, the entire 32 bits are affected. In fact, the other operand(s) are extended BEFORE the operation to satisfy this requirement.

Bytes may be addressed individually by addressing the high order byte on an even address. The low order byte has an odd address one count higher than the word address.

Words and and long words must be addressed on even boundaries. To access the second word of data stored as a long word at address **x**, the address of the second word would be **x+2**. Naturally, **x** must be even.

If a byte is "pulled" from or "pushed" onto a stack, only the high byte is affected; the lower byte remains unchanged.

The Programmer's Model, Figure 6.1, is common to all implementations of the M68000, and illustrates the following:

- 16 32-bit registers (%d0-%d7, %a0-%a7).
  - 8 data registers (**%d0-%d7**) which may be used for byte (8-bit), word (16-bit) and long word (32-bit) operations.
  - 7 address registers (**%a0-%a6**) and the Stack Pointer (**%a7** or **sp**). The %a7 register may be the User Stack Pointer (**usp**) or the Supervisor Stack Pointer (**ssp**), depending on the status of the "S" bit in the Status Register (**sr**).

    Address registers may be used for word and long word operations ONLY.

 - ANY register may be used as an index register.

- 32-bit Program Counter (**pc**)
- 8-bit Condition Code Register (**ccr**)

The **ccr** is found in the low 8-bits of the Status Register. Only the low order byte of the **sr** is accessible in user mode. The high order byte of the **sr** is considered the "system byte" and the processor must be in supervisor mode to access the high order bits.

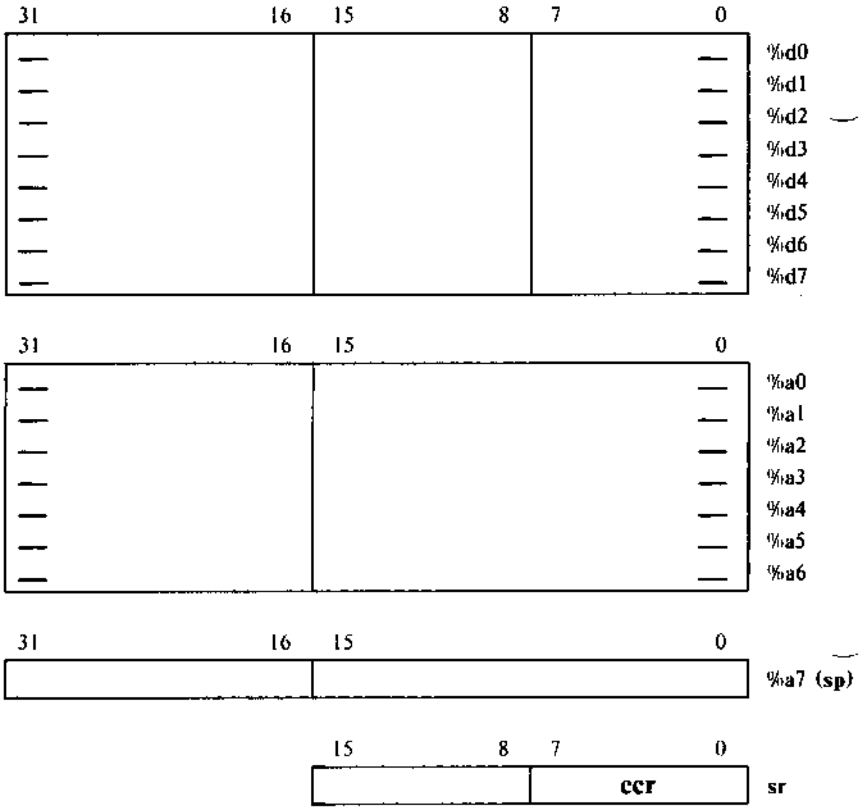


Figure 6.1. Programmer's Model

3.3 Status Register

The Status Register holds the following information:

- 8 levels of interrupt mask are available through the possible combinations of bits 8-10. The interrupt mask, then, is in the "system byte" and only accessible in supervisor mode.
- The condition codes:
  - (V) overflow
  - (Z) zero
  - (N) negative
  - (C) carry
  - (E) extend
- Trace mode status bit (bit 15)
- Supervisor mode status bit (bit 13)

3.4 Data Types

There are five basic data types supported by the 68000:

1. Bits
2. BCD digits (4 bits)
3. Bytes (8 bits)
4. Words (16 bits)
5. Long Words (32 bits)

Memory addresses, status word data, etc. are provided for in the instruction set.

3.5 Error Detection

Several hardware traps, provided to indicate abnormal internal conditions, also detect the following error conditions:

- Word access with odd address

- Illegal instructions
- Unimplemented instructions
- Illegal memory access (bus error)
- Divide by zero
- Overflow condition code (See the **M68000 16-bit Microprocessor User's Manual** regarding the **TRAP** instruction)
- Register out of bounds (See the **M68000 16-bit Microprocessor User's Manual** regarding the **CHK** instruction)
- Spurious interrupt

Sixteen software **trap** instructions are also provided to help detect errors in application programs.

The "Trace mode" provides instruction-by-instruction tracing of a program being debugged by causing a trap to occur after each instruction is executed. The microprocessor must be in the **supervisor state** to enter trace mode.

The supervisor state is especially useful in that it provides a high degree of protection by restricting the privilege to alter selected areas of memory. This is an especially important protection when an external memory management unit is being used.

4. Usage

The following UNIX System commands invoke the assembler:

as5.0 [−o objfile] [−v] [−l] inputfile

or

as [−o objfile] [−n] [−m] [−R] [−V] inputfile

or

ljas [−o objfile] [−n] [−m] [−R] [−V] inputfile

The **as5.0** command produces **a.out** format object files, the **as** command produces "COFF" (Common Object File Format) object files and the **ljas** command is a special version of the **as** command that produces "long jump" instructions rather than (short) branch instructions.

If the **−o** option is given, the following string will be used as the output file name. If no such specification had been made, the output will be left in a file whose name was formed either by appending a **.o** suffix to the end of the input file name, or by replacing the input file's present suffix with the .o suffix.

```
input:   as file.s (or as5.0 file.s or ljas file.s)
output:  file.o

input:   as file (or as5.0 file or ljas file)
output:  file.o
```

4.1 Options

The options have the following significance:

−l This option to **as5.0** produces an assembly listing on a file whose name is formed by adding a **.lst** suffix to the object file name specified with the **−o** option. If the **−l** option is specified, but the **−o** is not, the assembly listing is placed on a.lst.

−m This option for **as** and **ljas** requests that the **m4** macro pre-processor be run on the input to the assembler.

Remember, if using this pre-processor, be careful not use any of the **m4** keywords as variable names, function names or labels in your input file because the **m4** pre-processor is unable to determine which are assembler symbols and which are real **m4** macros.

−n This option for **as** and **ljas** requests that long/short address optimization be turned off. By default, address optimization takes place.

−o This option for all three assembler commands requests that the following string be used as the name of the object file. If this option is not specified for the **as5.0** command, the object file will be placed on a file called **a.out5.0**.

−v This option for the **as5.0** command requests the interpretation of the 68010 mnemonics.

−R This option, for the **as** and **ljas** commands, requests that the input file be removed (unlinked) after assembly is completed. This option is off by default.

−V This option, for the **as** and **ljas** commands, requests that the version number of the assembler being run be written on standard error output.

5. Syntax

Typical **as** assembly code looks like these:

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

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

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

These general points about the example should be noted:

- An identifier occurring at the beginning of a line and followed by a colon (:) is a **label**. One or more labels may precede any assembly language instruction or pseudo-operation.
- A line of assembly code need not include an instruction. It may consist of a comment alone (introduced by #), a label alone (terminated by :), or it may be entirely blank.
- It is good practice to use tabs to align assembly language operations and their operands into columns, but this is **not** a requirement of the assembler. An opcode may appear at the beginning of the line, if desired, and spaces may precede a label. A single blank or tab suffices to separate an opcode from its operands. Additional blanks and tabs are ignored by the assembler.
- It is permissible to write **several** instructions on one line by separating them by semicolons. The semicolon is syntactically equivalent to a newline. A semicolon inside a comment is ignored.

5.1 Comments

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

5.2 Identifiers

An identifier is a string of characters taken from the following set

a-z A-Z _ ~ % 0-9

The first character of an identifier must be a letter (upper or lowercase) or an underscore. Upper and lowercase letters are distinguished. For example:

con35 and **CON35**

are two distinct identifiers.

There is NO LIMIT on the length of an identifier.

The value of an identifier is established by the **set** "pseudo-operation" or by using the identifier as a label.

The character ˜ has special significance to the assembler. A ˜ used alone, as an identifier, means "the current location." A ˜ used as the first character in an identifier becomes a "." in the symbol table. This allows symbols such as **.eos** and **.0fake** to make it into the symbol table, as required by the COFF (Common Object File Format). (See the chapter on COFF in the **UniPlus$^+$ Programming Guide.**)

5.3 Register Identifiers

A register identifier is an identifier preceded by the character "%" and represents one of the available registers. This identifier is used for both data registers and address registers.

The predefined register identifiers recognized by the assembler are:

M68000 REGISTERS		
%d0	**%a0**	**fp** (frame pointer)
%d1	**%a1**	**sp** (system stack pointer)
%d2	**%a2**	**ssp** (supervisor stack pointer)
%d3	**%a3**	**usp** (user stack pointer)
%d4	**%a4**	**sr** (status register)
%d5	**%a5**	**ccr** (condition code register)
%d6	**%a6**	**pc** (program counter)
%d7	**%a7**	

M68010 ONLY	
vbr	vector base register (Accessed by the **movec** instruction)
sfc	alternate function code register (Accessed by the **movec** instruction)
dfc	alternate function code destiniation register (Accessed by the **movec** instruction)

NOTE: The address register **%a6** is equivalent to **fp**, the frame pointer. The address register **%a7** is equivalent to **sp** the System Stack Pointer, which in turn can be either **ssp** (the Supervisor Stack Pointer) or **usp** (the User Stack Pointer), depending upon the processor "Privilege State."

Use of both **%a7** and **sp**, or **%a6** and **fp**, in the same program may result in confusion.

5.4 Constants

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

5.4.1 Numerical Constants

A decimal constant is a string of digits beginning with a non-zero digit.

An octal constant is a string of digits beginning with zero.

A hexadecimal constant consists of the characters **0x** or **0X** followed by a string of characters from the set **0-9**, **a-f**, and **A-F**. In hexadecimal constants, upper and lowercase letters are not distinguished.

Binary numbers consist of a % followed by a binary number.

If the number following is an "immediate," it will be preceded by an ampersand ("&"). In the old syntax, this was indicated by the hash symbol ("#").

BINARY	%001010011010
DECIMAL	666
OCTAL	01232
HEXIDECIMAL	0x29A

5.4.2 Character Constants

An ordinary character constant consists of a single-quote (') followed by an arbitrary ASCII character other than \. The value of the constant is equal to the ASCII code for that arbitrary ASCII character.

Special meanings of characters are overridden when used in character constants. for example, if '# is used, the # is not treated as a comment indicator.

A special character consists of '\ followed by another character. All the special character constants, and examples of ordinary character constants, are listed here:

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

5.5 Segments

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

Primarily to simplify compiler code generation, the assembler permits up to four separate **text** segments and four separate **data** segments named 0, 1, 2, and 3. The assembly language program may switch freely between them by using assembler pseudo-operations. When generating the object file, the assembler concatenates the **text** segments to generate a single **text** segment, and the **data** segments to generate a single **data** segment. Thus, the object file contains only one **text** segment and only one **data** segment.

Since there is never more than one **bss** segment, this segment maps directly into the object file.

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

```
data      1          # segment 1
word      0x1111
data      0          # segment 0
long      0xffffffff
data      1          # segment 1
byte      0x2222
```

then equivalent object code would be generated by

```
data      0
long      0xffffffff
word      0x1111
word      0x2222
```

5.6 Location Counters and Labels

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

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

5.7 Types

Identifiers and expressions may have values of different types.

- In the simplest case, an expression (or identifier) may have an **absolute** value, such as 29, −5000, or 262143.
- An expression, or identifier, may have a value **relative** to the start of a particular segment. Such a value is known as a **relocatable** value. The memory location represented by such an expression cannot be known at assembly time, but the **relative** values, that is the difference between the start of a particular segment and the location of the expression or identifier, can be known if they refer to the same segment.

  Identifiers that appear as labels have **relocatable** values.
- If an identifier is never assigned a value, it is assumed to be an **undefined external**. Such identifiers may be used with the expectation that their values will be defined in another program, and therefore are known at load time. However, the relative values of **undefined externals** cannot be known.

5.8 Expressions

All constants are absolute expressions.

An identifier may be thought of as an expression having the identifier's type.

Expressions may be built up from lesser expressions using the operators + − * and / according to the following type rules:

- An absolute expression + another absolute expression results in an absolute expression.

- Either an absolute expression + a relocatable expression OR A relocatable expression + an absolute expression will result in a relocatable expression.
- Either an absolute expression + an undefined external expression OR an undefined external expression + an absolute expression will result in an undefined external expression.
- An absolute expression − an absolute expression will result in an absolute expression.
- A relocatable expression − a relocatable expression will result in an relocatable expression.
- An undefined external expression − an absolute expression will result in an undefined external expression.
- If two relocatable expressions are relative to the same segment, then:

  A relocatable expression − a relocatable expression will result in an absolute expression.

  However, use of this construction is dangerous, especially when dealing with identifiers from text segments. The problem is that the assembler will determine the value of the expression BEFORE it has resolved all questions about span-dependent optimizations. Use this feature at your own risk!
- An absolute expression * an absolute expression will result in an absolute expression.
- An absolute expression / an absolute expression will result in an absolute expression.
- The complement of an absolute expression is an absolute expression.

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

If the result of a division is a positive non-integer, it will be truncated toward zero. If the result is a negative non-integer, the direction of truncation cannot be guaranteed.

6. Addressing

The first word of an instruction, the "operation word," provides the name and size of the function to be performed. The remaining words specify the operands.

Operand locations may be expressed in one of the following ways:

- Register Specification
- Effective Address
- Implicit Reference

Most instructions specify the location of an operand using the Effective Address.

The Effective Address is composed of two 3-bit fields

1. the mode field, which selects the address mode for the instruction, and
2. the register field, which contains the number of a register.

The Effective Address modes are grouped into three catagories:

1. Register direct, (Data register direct, and Address register direct)
2. Memory addressing, (Address register indirect, address register indirect with postincrement, address register indirect with predecrement, address register indirect with

displacement, address register indirect with index)

3. Special (Absolute short address, absolute long address, program counter with displacement, program counter with index, immediate data)

Implicit references are sometimes made to the program counter, system stack pointer, supervisor stack pointer, user stack pointer or the status register. The implicit references are clearly indicated in the instructions to which they apply. See Figure 6.4 for a complete list of the instructions available.

In addressing memory, there are two classes of reference:

1. Program references, which refer to the memory location of a program; and
2. Data references, which refer to the memory location of data.

ALL operand writes, and most reads, are to data space.

6.1 Addressing Modes

Although the addresses used in **absolute** addressing modes must eventually be filled in with constants, that can be done by the loader, there is no need for the assembler to be able to compute them. Therefore, the **Absolute Long** addressing mode is commonly used for accessing **undefined** external addresses.

Figure 6.2 summarizes the **as** syntax for M68000 addressing modes. The following abbreviations are used in this figure:

%a n	Address register, where **n** specifies the register number.
d	Displacement.
%d n	Data register, where **n** specifies the register number.
pc	Program Counter

%r n	Any register, address or data, where **n** specifies the register number.
%ri	Any register, address or data, used as an index.

Notation for UniPlus+ Version 5.0	Notation for UniPlus+ Version 5.2	Effective Address Mode
Dn	**%d n**	Data Register Direct
An	**%a n**	Address Register Direct
(An)	**(%a n)**	Address Register Indirect
An@+	**(%a n) +**	Address Register Indirect with Postincrement
An@−	**− (%a n)**	Address Register Indirect with Predecrement
An@(d)	**d(%a n)**	Address Register Indirect with Displacement
(**d** is a signed 16-bit absolute displacement)		

An@(d,Ri.W)
An@(d,Ri.L)
	d(%a n,%ri.w)
	d(%a n,%ri.l)
		Address Register Indirect with Index

(**d** signifies s signed 8-bit absolute displacement)		
xxx.W	**xxx**	Absolute Short Address
(**xxx** is an expression yielding a signed 16-bit memory address)		
xxx.L	**xxx**	Absolute Long Address
(**xxx** is an expression yielding a 32-bit memory address)		
PC@(d)	**d(%pc)**	Program Counter with Displacement
(**d** is a signed 16-bit absolute displacement)		

PC@(d,Ri.W)
PC@(d,Ri.L)
	d(%pc,%r n.w)
	d(%pc,%r n.l)
		Program Counter with Index

(**d** signifies a signed 8-bit absolute displacement)		
#xxx	**&xxx**	Immediate Data
(**xxx** signifies an absolute constant expression)		

Figure 6.2. Effective Address Modes

7. Stacks and Queues

The address register indirect postincrement and predecrement addressing modes provide the 68000 with stack and queue data structures. "A stack is a last-in-first-out (LIFO) list, a queue is a first-in-first-out (FIFO) list."*

Many instructions use the system stack implicitly. The programmer may create user stacks and queues by using the appropriate addressing modes.

7.1 System Stack

The system stack pointer (**sp** or %a7) may be either the supervisor stack pointer (**ssp**) or the user stack pointer (**usp**), depending on the state of the "S" bit (bit 15) of the status register. If the S bit indicates supervisor state is set to 1, the **ssp** is the active system stack pointer and the **usp** cannot be addressed as an address register. If the S bit is 0, the **usp** is the active system stack pointer and the **ssp** cannot be addressed.

Since the stack is filled from high to low, to "push" an item on the stack, use the −(S**sp**) address mode. To "pull" an item from the stack, use the (**sp**)+ address mode.

When a subroutine call is made, the program counter is saved on the active system stack, and it is restored when the return from subroutine call is made. When exception processing occurs, both the program counter and the status register are saved on the supervisor stack.

* **M68000 16-bit Microprocessor User's Manual**, page 22.

7.2 User Stacks

User stacks may be filled either from high memory to low memory, or from low memory to high memory. The address register indirect with postincrement and predecrement addressing modes are used to create and manipulate these stacks. The following are some important points regarding these modes:

- When using predecrement, remember that the register is decremented BEFORE its contents is used as the stack pointer.
- When using postincrement remember that the register is incremented AFTER its contents is used as the stack pointer.
- When mixing byte data with word and long word data, be careful to use bytes in pairs to assure even word boundaries. Remember, trying to address an odd word boundary will cause exception processing to occur.
- When implementing a stack from high memory to low memory, use the following:

  ```
  -(%ax) to "push" data onto the stack
   (%ax) to "pull" data off of the stack
  ```

- When implementing a stack from low memory to hihg memory, use the following:

  ```
   (%ax) to "push " data onto the stack
  -(%ax) to "pull" data off of the stack
  ```

7.3 Queues

The address register indirect with postincrement or predecrement addressing modes are used to create and manipulate user queues. User queues can be created to grow from high memory to low memory or from low memory to high memory, just by the correct manipulation of these addressing modes. Two address registers are used as pointers for the queue functions **put** and **get**.

To establish a queue from low to high memory, use the address indirect with postincrement addressing mode.

If the queue is to be implemented as a circular buffer, the address register should be checked BEFORE the **get** or **put** is performed. If necessary, adjust the address register by subtracting the buffer length (in bytes).

Queue growth from high to low memory is implemented with the address indirect with predecrement addressing mode. If the queue is to be implemented as a circular buffer, the address register should be checked AFTER the **get** or **put** is performed. If necessary, adjust the address register by adding the buffer length (in bytes).

8. Pseudo-Operations

8.1 Data Initialization

byte abs, abs, ...	One or more arguments, separated by commas, may be given. The values of the arguments are computed to produce successive bytes in the assembly output.
short *abs*, **abs, ...**	One or more arguments, separated by commas, may be given. The values of the arguments are computed to produce successive 16-bit words in the assembly output.
long *expr*, **expr, ...**	One or more arguments, separated by commas, may be given. Each expression may be **absolute**, **relocatable**, or **undefined external**. A 32-bit quantity is generated for each such argument (for **relocatable** or **undefined external** expressions, the value may not be filled in until load time).
	Alternatively, the arguments may be bit-

field expressions. A bit-field expression has the form:

```
n : value
```

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

```
long4:-1, 16:0x7f, 12:0, 5000
```

and

```
long4:-1, 16:0x7f, 5000
```

are equivalent to

```
long0xf007f000, 5000
```

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

```
long24:0xa, 24:0xb, 24:0xc
```

yields the same thing as

```
long0x00000a00, 0x00000b00, 0x00000c00
```

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

```
space6
```

is equivalent to

```
byte0,0,0,0,0,0
```

8.2 Symbol Definition

set *identifier*, **expr**	The value of **identifier** is set equal to **expr**, which may be absolute or relocatable.
comm *identifier, abs*	The named identifier is to be assigned to a common area of size **abs** bytes. If **identifier** is not defined by another program, the loader will allocate space for it. The type of **identifier** becomes **undefined external**.
lcomm *identifier*, **abs**	The named identifier is assigned to a **local common** of size **abs** bytes. This results in allocation of space in the **bss** segment. The type of **identifier** becomes **relocatable.**
global *identifier*	This causes **identifier** to be externally visible. If **identifier** is defined in the current program, then declaring it global allows the loader to resolve references to **identifier** in other programs. If **identifier** is not defined in the current program, the assembler expects an external resolution; in this case, therefore, **identifier** is global by default.

8.3 Location Counter Control

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

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

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

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

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

8.4 Symbolic Debugging

The assembler allows for symbolic debugging information to be placed into the object code file with special pseudo-operations. The information typically includes line numbers and information about C language symbols, such as their type and storage class. The Motorola 68000 C compiler generates symbolic debugging information when the **−g** option is used. Assembler programmers may also include such information in source files.

8.5 "file" and "ln"

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

file filename

where **filename** consists of one to 14 characters.

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

ln line [,value]

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

8.6 Symbol Attribute

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

```
defname
 .# Attribute
 .# Assigning
 .# Operations
endef
```

NOTE: **def** DOES NOT define the symbol, although it does create a symbol table entry. Because an undefined symbol is treated as external, a symbol that appears in a **def**, but never acquires a value, will eventually result in an error at link edit time.

NOTE: To allow the assembler to calculate the sizes of functions for other tools, each **def/endef** pair that defines a function name must be matched by a **def/endef** pair after the function in which a storage class of **−1** is assigned.

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

val *expr*	Assigns the value **expr** to **name**. The type of the expression **expr** determines with which section **name** is associated. If value is ˜, the current location in the text section is used.
scl *expr*	Declares a storage class for **name**. The expression **expr** must yield an ABSOLUTE value that corresponds to the C

	compiler's internal representation of storage class. The special value **−1** designates the physical end of a function.
type *expr*	Declares the C language type of **name**. The expression **expr** must yield an ABSOLUTE value that corresponds to the C compiler's internal representation of a basic or derived type.
tag *str*	Associates **name** with the structure, enumeration, or union named **str** that must have already been declared with a **def/endef** pair.
line *expr*	Provides the line number of **name**, where **name** is a block symbol. The expression **expr** should yield an ABSOLUTE value that represents a line number.
size *expr*	Gives a size for **name**. The expression **expr** must yield an ABSOLUTE value. When **name** is a structure or an array with a predetermined extent, **expr** gives the size in bytes. For bit fields, the size is in bits.
dim *expr1*, **expr2**, ...	Indicates that **name** is an array. Each of the expressions must yield an ABSOLUTE value that provides the corresponding array dimensions.

8.7 Switch Table

The MC68000 C compiler generates a compact set of instructions for the C language **switch** construct, of which an example is shown below.

```
    sub.l&1,%d0
    cmp.l%d0,&4
    bhiL%21
    add.w%d0,%d0
    mov.w10(%pc,%d0.w),%d0
    jmp6(%pc,%d0.w)
    swbeg&5
  L%22:
    shortL%15-L%22
    shortL%21-L%22
    shortL%16-L%22
    short L%21-L%22
    shortL%17-L%22
```

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

The **swbeg** pseudo-operation takes an argument that looks like an immediate operand. The argument is the number of lines that follow **swbeg** and that contain switch table entries. **Swbeg** inserts two words into text. The first is the ILLEGAL instruction code. The second is the number of table entries that follow. The Motorola 68000 disassembler needs the ILLEGAL instruction as a hint that what follows is a switch table. Otherwise it would get confused when it tried to decode the table entries (differences between two symbols) as instructions.

9. Span-Dependent Optimization

The assembler makes certain choices about the object code it generates based on the distance between an instruction and its operand(s). Choosing the smallest, fastest form is called span-dependent optimization. Span-dependent optimization occurs most obviously in the choice of object code for branches and jumps. It also occurs when an operand may be represented by the program counter relative address mode instead of an absolute 2-word (long) address.

The span-dependent optimization capability is normally enabled; the **−n** command line flag disables it. When this capability is disabled, the assembler makes worst-case assumptions about the types of object code that must be generated.

The compiler generates branch instructions without a specific offset size. When the optimizer is used, it identifies branches that could be represented by the short form and it changes the operation accordingly. The assembler chooses only between long and very-long representation for branches.

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

Although the largest offset specification allowed is a word, large programs could conceivably have need for a branch to a location not reachable by a word displacement. Therefore, equivalent long forms of these instructions might be needed. When the assembler encounters a branch instruction without a size specification, or with a word size specification, it tries to choose between the long and very long forms of the instruction. If the operand can be represented in a word, then the word form of the instruction will be generated. Otherwise the very-long form will be generated. For unconditional branches, (e.g., **br**, **bra** and **bsr**), the very-long form is just the equivalent jump (**jmp** and **jsr**) with an absolute address operand (instead of pc-relative).

For conditional branches, the equivalent very-long form is a conditional branch around a jump, where the conditional test has been reversed.

Figure 6.1 summarizes span-dependent optimizations. The assembler chooses only between the long form and very-long form, while the optimizer chooses between the short and long forms for branches (but not bsr).

INSTRUCTION	SHORT FORM	LONG FORM	VERY LONG FORM
br, bra, bsr	byte offset	word offset	**jmp** or **jsr** with absolute long address
conditional branch	byte offset	word offset	short conditional branch with reversed condition around **jmp** with absolute long address
jmp, jsr	–	pc-relative address	absolute long address
lea.l, pea.l	–	pc-relative address	absolute long address

Figure 6.3. Assembler Span-Dependent Optimizations

10. Machine Instructions

Figure 6.4 shows how MC68000 instructions should be written to be understood correctly by the **as** assembler. In addition to the abbreviations previously described for Figure 6.2 "Effective Address Modes," the following abbreviations are used in the following figure:

CC In the contexts **b***cc*, **db***cc* and **s***cc*, the letters *cc* represent any of the following condition code designations (except that **F** and **T** may not be used in the **b***cc* instruction):

CC	carry clear	**LS**	low or same
CS	carry set	**LT**	less than
EQ	equal	**MI**	minus
F	false	**NE**	not equal
GE	greater or equal	**PL**	plus
GT	greater than	**T**	true
HI	high	**VC**	overflow clear
LE	less or equal	**VS**	overflow set

ccr Condition Code Register (the low 8 bits of the Status register)

EA Effective Address

sr Status Register.

ssp Supervisor Stack Pointer.

usp User Stack Pointer.

Chapter 7: LD — LINK EDITOR

CONTENTS

LIST OF FIGURES

Chapter 7

LD –

THE COMMON LINK EDITOR

1. Introduction

The link editor (see **ld(1)**, in the *UniPlus+® User Manual*) 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. Although the link edit process is controlled in detail through use of this language (described later), most users do not require this degree of flexibility, and the manual page in the *UniPlus+ User Manual* is sufficient instruction in the use of this command.

The command language (described later) supports the ability to

- 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.

1.1 Host and Target

In a cross-compilation system, the host machine is the machine on which the link editor is running, and the target machine is the machine on which the output object file will run. For instance, the b16 link editor will run on the PDP™-11/70, VAX™ or 3B20S™ machines, but the object file will run only on the target machine for the b16 — the Intel 8086.

On a native UNIX™ system, the host and the target are the same. That is, the link editor on a VAX or PDP-11/70 produces an object file that is executable on that machine.

1.2 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 as **illegal** or **nonexistent** with respect to the linking process. Memory

PDP and VAX are trademarks of Digital Equipment Corporation.
UNIX and 3B20S are trademarks of AT&T Bell Laboratories.
UniPlus+ is a registered trademark of UniSoft Corporation.

configurations other than the default must be explicitly specified.

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

1.3 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.

1.4 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.

1.5 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.

1.6 Regions

Regions are currently used only in the link editor for the intel 8086 – the b16 instantiation of the common link editor. A region refers to a range of memory that begins with a virtual address of zero. On the Intel 8086, an address really has two components, a *base* as indicated by a segment register and a 16-bit *offset.* For any memory reference, the contents of one of the four segment registers is added to the *offset* to form an address. Segment registers are set to the *base* address of a region, and all references in the text are resolved with respect to the *virtual* address.

For example, in the simplest case of a single region where everything is loaded between 0 and 64k, the virtual addresses of all symbols are the same as their physical addresses, and both the text and data segment registers are set to the region origin. If, on the other hand, an application has 50k of data and 20k of text, they can form two regions, one at address 0 and the other at 20k. If a global symbol, *gsymbol,* was defined at address 20k+2, and a C program contained the statement:

a = g*symbol*

where **a** is an automatic variable, the virtual address of *gsymbol* would be 2. The link editor always allows an arbitrary number of regions. Relocation is performed only with one region, the code must modify the segment registers. This is done automatically if the program uses a transfer vector for function linkage. If a transfer vector is not used, segment registers must be set explicitly by the user before doing any inter-region transfers.

There are three restrictions on regions:

1. The first is that they must begin at an address aligned to a 16-byte boundary; that is, the low four bits of the address MUST be 0. For example, 0, 16, 32, etc. are valid region origins.

   This is because the 20-bit physical address is stored in a segment BEFORE adding in the offset. For example, if a region begins at 0x12000, the segment register will contain 0x1200.

2. The second restriction is that regions may not exceed 64k in size, or else there can be no direct references to addresses beyond 64k into the region.

3. The third restriction is that the physical memory assigned to user-specified regions may not overlap.

1.7 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:

- **.text**
  The **.text** section contains the instruction text (for example, executable instructions).
- **.data**
  The **.data** section contains initialized data variables.
- **.bss**
  The **.bss** section contains uninitialized data variables.

For example, if a C program contained the following global (i.e., not inside a function) declarations:

```
int i = 100;
char abc[200];
```

and the following 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.)

2. Using the Link Editor

To use the link editor, give the following 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 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, 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. Then the use the folloiwing command:

```
ld  ifile
```

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:

1. file1.o
2. file3.o
3. file4.o,

4. file2.o.

Note from this example that an ifile is read and processed immediately upon being encountered in the command line.

2.1 Options

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, excutable file. Messages are issued when undefined symbols are found, and several special symbols are defined. Unless overridden by the −**r** option, relocation information is stripped from the output file. If neither −**r** or −**a** is specified, the −**a** is assumed.
−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" under the Section heading "Notes and Special Considerations" for a discussion of how the option is used.
−f *bb*	Sets the default fill value. The argument *bb* is a 2-byte constant. 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. If the −**f**

option is not used, the default fill value is zero for all sections except the **.tv** section, whose default fill value is 0xFFFF.

−h *nn*	Set the size of the optional header in the output file to be *nn* bytes. The argument *nn* is an integer constant. If the **−h** option is not used, the optional header will be 0 length. Any part of the optionl header not assigned a value (as a result of using the **−p** or **−X** options) will be cleared to 0.
−i	Generate separate I and D spaces, allowing 64k of instructions and 64k of data, each in a separate region. The default is to have only one address space or region. If there are REGIONS directives in the ifile, they will override the setting of the **−i** option. This option is valid only for those instantiations of the link editor that allow regions (i.e. b16), all other versions ignore it.
−i*ld*	Generate the sections reserved for use by the incremental link editor. This option invokes both **−r** and **−a** options.
−l*x*	Specifies an 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, **−l***c* refers to libc.a, **−l***C* 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 *nn*	Name the output object file. The argument *nn* 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 option *nn* can be a full or partial UNIX path name.
−p *nn*	Generates a patch list. This list is generated in the optional header field of the output file, following

anything built as a result of the **−e**, **−h** or **−X** options. If the optional argument *nn* is used, only *nn* bytes of physical memory will be allocated for patch sections. This conserves memory usage while retaining the ability to later increase patch sections to the size specified in the relevant SECTIONS directive.

−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.

−tv Use the transfer vector linkage convention. When **−tv** is specified, all input object files must have been compiled/assembled using the **−tv** option. This option is valid only for the instantiations of the common link editor that allow transfer vectors. (i.e., 3bld, b16ld, x86ld and mc68ld).

−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 Do not place anything in address zero. This is used to catch references through null pointers. This option is overridden if any section or memory directives are used.

−B *nn* Generates "pad" output sections. The length of each section will be *nn* bytes.

−F Perform alignment necessary for demand paging. Sections will be aligned on stricter boundaries in the address space. Sections will be blocked in the output file so that they begin on file system block boundaries. Also the magic number 0413 will be stored in the UNIX header.

−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.

−L *dir* 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 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 system header. This option is NOT directly recognized by the compiler (cc), so you have to use the −**W** option to pass the version number to the link editor. For example:

```
−W l,−VS num
```

3. Link Editor Command Language

3.1 Expressions

Expressions may contain global symbols, constants, and most of the basic C language operators (see the last section of this document entitled "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.

3.2 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* 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.

3.3 Specifying a Memory Configuration

MEMORY directives are used to specify

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

If no directives are supplied, 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

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

Other attributes may be added in the future if necessary. If no attributes are specified on a MEMORY directive or if no MEMORY directives are supplied, memory areas assume the attributes of 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
}
```

3.4 Region Directives

Region specifications are used when more than one address space or region is to be defined. The syntax of this directive is very similar to memory configuration specifications. An arbitrary name may be assigned to a region, and this name will only be used if a section is to be loaded into the region. Origin and length specifications are exactly like those on the MEMORY directive. Attributes may not be assigned to a region, since the region takes the physical attributes of the physical memory of the region. In addition to the origin and length specification, an optional virtual address may be assigned to the beginning of a region. The physical memory within a user's regions may not overlap, but the virtual spaces of regions may have to overlap in some cases. The virtual address is given by assigning a value to ".", much like changing the location counter in a section.

The syntax is:

```
REGIONS
{
        name1  :  origin = n1, length = n2
        name2  :  origin = n3, length = n4, . = n5
}
```

For example, if initialized data is stored in ROM and then copied into RAM during execution, the beginning of ROM should have the same virtual address as the place in RAM to which the copy is being made. If that virtual address is 0x3000 and the physical address of the ROM is 0x9000, then the virtual zero of the region is at physical address 0x6000 through 0x8FFF could not be used normally. Note that the virtual zero of a region should always correspond to a physical address that is divisible by 16, otherwise the region is not valid region for the 8086.

```
REGIONS
{
        r: o  = 0x9000, 1 = 0x2000, . = 0x3000
}
```

If a transfer vector is not being used, one region is assumed unless REGIONS specification are given or the **−i** option is used. If a transfer vector is used, the link editor combines **.data** and **.bss** into a single region and creates a region for every other output section. Because each text section can validly be a region and the **.data** and **.bss** sections MUST be in the same region, users are strongly discouraged from using REGIONS with a transfer vector. The REGIONS specification overrides the default behavior, and the user assumes full responsibility for placing sections in the appropriate places. The **−i** option is meaningless for transfer vectors.

3.5 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.

3.5.1 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

1. Section sec1 from file file1.o
2. All sections from file2.o, in the order they appear in the file
3. Section sec1 from file file3.o, and then section sec2 from file file3.o

If there are any additional input files that 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.o or file3.o, they will be placed in output sections with the same names as the input sections.

3.5.2 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.

3.5.3 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.).

3.5.4 Grouping Sections Together

The default allocation algorithm for **ld** is:

1. Links all input **.text** sections together into one output section. This output section is called **.text** and is bound to an address of 0x0.
2. Links all input **.data** sections together into one output section. This output section is called **.data** and is bound to an address aligned to a machine dependent constant.
3. 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.

3.5.5 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:

1. 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.
2. The text of input file f2.o begins at 0x100 bytes following the end of f1.o(&.text).
3. 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**."

3.5.6 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:

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

The first case has already been discussed in the previous section.

The second case 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.

3.5.7 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.

3.5.8 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:

1. Explicit initialization options must be used to initialize all **.bss** sections in the output section.
2. 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.

3.6 Transfer Vectors

A *transfer vector* is an ordered list of entries similar to an array of pointers or a jump table. Each entry contains the physical address of an external or static identifier. The address is one word in length, and the address is right-justified within the word. The entire transfer vector is a zero-oriented, one-dimensional array of "long int". The first slot in the transfer vector (slot 0) cannot be used because of a conflict with the null pointer, whose value is zero.

In any C source file compiled using the **cc** compiler, with the **−tv** option, all function references are "*indirect*." Calling a function **funx()** is accomplished by an indirect reference through the transfer vector entry containing the actual address of **funx()**.

Assembler source files can also employ transfer vector linkage, by using the **−tv** option as invocation AND through the use of the indirect assembler language call. The appropriate macros should be used for this purpose. Assembler source can also generate **tv** references to identifiers other than functions.

The **ld** defines the transfer vector to be a separate output section, called **.tv**. Unlike other output sections, the contents of the transfer vector output section are supplied entirely by **ld**, as a result of the link-edit process and certain user commands. The transfer vector is generated by **ld** ONLY when the **−s** option has been (explicitly or implicitly) selected.

Each defined function is assigned a transfer vector slot, and the address of the function is written into that slot in the **.tv** output section. A transfer vector entry is assigned a value in one of two ways:

1. By the user:

Through the use of the ASSIGN field of the TV directive, specific functions can be assigned to specific transfer vector slots.

2. By **ld**:

Any identifier referenced through transfer vector linkage will, if the identifier has no **tv** slot, be assigned the next available slot in the transfer vector.

The **ld** assigns **tv** slots to identifiers on a "first seen, first assigned" basis. These two features of **tv** slot assignment, coupled with the general properties of transfer vectors, makes field update and function replacement applications possible.

Attributes of the transfer vector (also called the **.tv** section) may be specified using a TV directive of the form:

```
TV bond-addr
{
          SPARE     = nbr-slots
          LENGTH    = nbr-slots
          RANGE(first-slot, last-slot)
          ASSIGN    (
                    fcn1  =  slot1
                    fcn2  =  slot2
                          .
                          .
                          .
                    fcnj  =  slotj
                        )
} = fill-fcn
```

where:

bond-addr	The virtual address of the start of the **.tv** section. If supplied, the 3B20D instantiation of the link editor requires this to be 0x760000.
SPARE	the number of **tv** slots to be allocated over and above those actually assigned by the current **ld** run. The default value for **SPARE** is 0.
LENGTH	The total number of **tv** slots to be allocated in the transfer vector. It defaults to the number actually assigned by the current **ld** run.
RANGE	The indices of the first and last slot that **ld** can use when it assigns transfer vector slots to identifiers. "first-slot" defaults to 0, while "last-slot" defaults to the LENGTH value (if specified) or to the largest **tv** slot index given in an ASSIGN directive.

ASSIGN	The **tv** slot index to be assigned to a particular identifier.
fill-fcn	The identifier, the value of which is to be written into **tv** entries which are allocated but not otherwise assigned. The default value written to such **tv** slots is 0xFFFF. Specifying a **fill-fcn** name allows the user to cause automatic branching to, for example, an error handling routine, should an invalid transfer vector reference occur. However, this option functions properly ONLY if the fill function is defined within the subsystem that is being linked, because only then is the fill function's address known by **ld**.

All parts of the TV directive (including the TV directive itself) are optional. Multiple TV directives can be used, in which case the final description of the output **.tv** section is a union of all supplied information. When TV directives are used, there are some restrictions.

- If the SPARE field is specified, then neither LENGTH, RANGE nor ASSIGN fields may be used.
- The LENGTH and RANGE fields can appear at most ONCE, and the LENGTH field, if supplied, must be specified BEFORE the RANGE field.
- Multiple ASSIGN fields may be used. The assignments within the ASSIGN need not be ordered by slot number or function name.
- The **tv** slot indices appearing in the RANGE field must satisfy:

0 <= first-slot <= last-slot <= total-length <= 0x80

- The slot indices appearing in the ASSIGN field MUST satisfy:

0 <= slot <= total-length <= 0x8000

- Any function that is ASSIGNed a slot within the range specified by the RANGE field (or its default values) MUST be defined in the current link edit.
- Any function that is ASSIGNed a slot outside the range specified by the RANGE field (or its default values) MUST NOT be defined in the current link edit.

The SPARE and LENGTH fields are provided as a means of controlling the total number of slots allocated in the transfer vector. The RANGE field permits a transfer vector to be partitioned into disjoint ranges. Each range can then be used for a specific purpose (e.g., subsystem, set of functions, application, etc.). The ASSIGN field permits the user to override the default allocation algorithm used for **tv** slot assignment. The **tv** section can thus be "mapped" in advance and in such a way as to carry over from one **ld** run to the next.

3.7 Subsystem Loading

The following section is applicable ONLY to instantiations of the link editor that use transfer vectors such as b16ld or mc68ld. Even though 3bld uses transfer vectors, this section should be read with caution by a 3bld user, as the terms subsystem and application as used here conflict with the DMERT usage, and as a process must be contained in one process file to run under DMERT. Nevertheless, this section should be read for a better understanding of the capabilities of transfer vectors.

Subsystem loading is an alternative to performing a single link-edit on the entire application. The application is divided into subsystems, each of which can be link-edited independently of the other parts of the application. When the final subsystem link-edits are completed, as subsystem specific *ifile*, which we will call SIFILE, and a common *ifile*, which we will call CIFILE, are sent to the link editor as input. All link-edits within a subsystem loading environment must be performed using transfer vector linkage.

A subsystem *ifile*, SIFILE, specifies the areas of physical memory reserved for each subsystem's **.text, .data** and **.bss** sections, and reserves a range of transfer vector slots with the RANGE directive.

The system *ifile*, CIFILE, specifies the total length of the system transfer vector with a LENGTH directive. The CIFILE contains the address assignments to all global **.data** and **.bss** symbols referenced accross subsystems, and transfer vector slot assignments to all functions referenced across the subsystems.

In the following example of subsystem loading *ifiles*, assume that there are only two subsystems in the application — subsystem A and subsystem B.

The SIFILE for subsystem A is as follows:

```
TV
{
        RANGE(1,200)
}
MEMORY
{
        mtext     : o = 0x01000, 1 = 0x4000
        mdata     : o = 0x40000, 1 = 0x2000
        mbss      : o = 0x50000, 1 = 0x1000
}
SECTIONS
{
        SSAtext:
        {
            A1.o (.text)
            A2.o (.text)
        } > mtext
        .data : { } > mdata
        .bss : { } > mbss
}
```

The SIFILE for subsystem B is:

```
TV
{
        RANGE(201,600)
}
MEMORY
{
        mtext     : o = 0x6000, 1 = 0x8000
        mdata     : o = 0x42000, 1 = 0x8000
        mbss      : o = 0x51000, 1 = 0x4000
}
SECTIONS
{
        SSBtext:
        {
            B1.o (.text)
            B2.o (.text)
            B3.o (.text)
        } > mtext
        .data : { } > mdata
        .bss : { } > bss
}
```

For this example, the common *ifile*, CIFILE, is:

```
TV
{
        LENGTH = 0x8000
        ASSIGN (
            Afunc1 = 1
            Afunc2 = 10
            Afunc3 = 100
        )
        ASSIGN (
            Bfunc1 = 201
            Bfunc2 = 203
            Bfunc3 = 600
        )
}
Aglobal = 0x80000;
Bglobal = 0x80002;
```

Subsystem A contains a total of 200 transfer vector slots. Subsystem A's SIFILE sets the allotment for the transfer vectors. Subsystem B has 400 slots assigned to it.

The maximum number of available system transfer vector slots (0x8000) is allocated within CIFILE. In this example, subsystem A assigns three of its functions to specific slots within its transfer vector slot range. The three subsystem A functions are assigned slots because subsystem B references only these three functions in subsystem A. Similarly, subsystem A references three subsystem B functions. These functions are assigned transfer vector slots within subsystem B's range.

The global, non-function symbols Aglobal and Bglobal are assigned absolute physical addresses in the CIFILE. This is necessary because subsystem A references Bglobal, and subsystem B references Aglobal. Aglobal is defined in subsystem A, and Bglobal is defined in subsystem B. Generally, global **.data** and **.bss** symbols which are shared across subsystems are taken from each subsystem and placed in a separate subsystem, which we will call gdata. The addresses assigned to Aglobal and

Bglobal must be the same addresses that the link editor assigned to them during normal allocation, that is, during the link-edit of gdata. If the addresses assigned are different than those assigned during allocation of gdata, the references to these symbols would be incorrect. For example, if Aglobal was assigned the address 0x80004 when gdata was built but was linked at the address in the CIFILE, the references to Aglobal in subsystem B would be incorrect. This results from the fact that subsystem B references to Aglobal are resolved with the CIFILE.

The CIFILE forms the interface between subsystems A and B. Through the CIFILE, the link editor is aware of the addresses of the global symbols that are not defined within the objects and archives owned by a subsystem. The CIFILE allows each subsystem to be linked separately.

The function that is not referenced by any subsystem, except the subsystem where it is defined, does not need a transfer vector slot assignment in the CIFILE. Similarly, nonfunction symbols that are not referenced, except by the subsystem owning them, do not need to be placed in the gdata subsystem, where non-function global symbol assignments are generated.

To link subsystem A, use the following command:

```
ld −tv CIFILE SIFILEA −o SSA.out
```

The link editor issues error messages for errors that occur in the subsystem loading SIFILE and CIFILE. In the example with subsystems A and B, assume that the transfer vector slot assignment *Afunc1 = 1* is changed to *Afunc1 = 201.* The link editor would issue the following error message:

```
ld SIFILEA : fatal :
Defined symbol Afunc1 assigned a tvslot outside
```

As specified in subsystem A's SIFILE, SIFILEA, the transfer vector slot range for subsystem A is RANGE(1,200). Since slot 201 is obviously outside this range, the error message will be printed. Functions that are defined within a subsystem can be assigned transfer vector slots only within the defining subsystem's range.

NOTE: The error message occurs whenever a function is moved from one subsystem to another without a CIFILE update.

For another example of a link editor error message, assume that 390 of the 400 reserved transfer vector slots are already used by functions defined within subsystem B. Three of these functions are referenced by subsystem A. The link editor requires that the number of slots equal or exceed the number of defined functions within a subsystem to produce a successful final link. Assume that 20 new functions are added to subsystem B without an update of the RANGE directive in SIFILEB. The link editor would then issue the following error message:

```
ld : SIFILEB fatal : tv range allows 400 entries : 410 needed
```

If the RANGE directive in the SIFILE for subsystem B, SIFILEB, is changed from RANGE(201,600) to RANGE(201,610), then the new functions could be successfully added to subsystem B.

For a further example, assume that the line *Bfunc2 = 203* is removed from the CIFILE. Then the link editor, assuming that subsystem A actually references the function Bfunc2 in the file A2.o, would issue the following message, after an attempt to link subsystems A was initiated:

Undefined symbol Bfunc2
First referenced in file A2.0

For an example of a different type of error, assume that *Afunc3* is incorrectly entered into the CIFILE as *Afunc4*. In addition, assume that *Afunc4* is referenced but not defined in subsystem A. In this case, *Afunc4* is not defined in subsystem B either. If an attempt is made to link subsystem A, the link editor will issue the following error message:

```
ld SIFILEA fatal :
Undefined symbol Afunc4 assigned a tvslot within
```

Another common error message issued by the link editor pertains to an inability to allocate sections. Error messages of this type indicate that certain parts of the input objects and archives cannot fit into areas specified in a particular SFILE.

For example, assume that the **.text** size of A1.o is 0x3000 and that the **.text** size of A2.o is 0x1E00. Then the total space needed for subsystem A's text in the output file is 0x4E00. However, as seen in SIFILEA, the **.text** from A1.o and the .text from A2.o is put into the mtext area, whose size is only 0x4000. In this case, the link editor would print the following error message:

```
ld : SIFILEA : Can't allocate section .text in owner mtext
```

In order to solve this problem, the line:

```
     mtext : o = 0x01000, l = 0x4000
```

would have to be changed to:

```
     mtext : o = 0x01000, l = 0x4E00
```

or the size of the **.text** in the input files would have to be reduced so that the **.text** section would fit.

4. Notes and Special Considerations

4.1 Changing the Entry Point

When **ld** is given the **−X** option, a UNIX system a.out header is written to the output file. 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):

1. The value of the symbol specified with the **−e** option, if present, is used.
2. The value of the symbol _start, if present, is used.
3. The value of the symbol main, if present, is used.
4. The value zero is used.

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

```
_start    =    expression;
```

Use of the **−e** option will force the **−X** option to be set automatically. Assigning a value to the symbol **_start** or having a symbol by this name already defined in an input file DOES NOT force the **−X** option to be set, and hence it must be explicitly supplied if the entry point is to be output.

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.

4.2 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:

1. .text
2. .data
3. .bss

Archive libraries are created through the use of the UNIX system **ar** command from object files generated by running the **cc** or **as** compilers.

Each library member has a *magic number*. For object files, there are two magic numbers:

1. one for object files generated with the **−tv** option, and
2. one for the object files generated without the **−tv** option.

The link editor enforces a policy that all input object files must have the same magic number. Any object file that fails this test is not processed and generates a fatal **ld** error. This policy, however, has an important exception:

MEMBERS OF ARCHIVE LIBRARIES WITH THE WRONG MAGIC NUMBER ARE SILENTLY SKIPPED

It is not considered an error, and no message is generated.

This permits an archive library to contain both tv and non-tv versions of its members, and to have the same library used as input to both tv and non-tv link edits. In each instance, members with the "other" magic number are ignored.

An archive library is always processed using *selective inclusion:* Only those members that resolve existing undefined-symbol references are taken from the library for link editing.

Libraries can be placed both inside and outside section definitions. In both cases, a member of a library is included for linking whenever

1. There exists a reference to a symbol defined in that member.
2. The reference is found by the **ld** prior to the actual scanning of the library.
3. The member has the correct magic number.

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

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

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

The ordering of archive libraries is important since for a member to be extracted from the library it MUST satisfy a reference that is KNOWN TO BE UNRESOLVED AT THE TIME THE LIBRARY IS SEARCHED. Archive libraries can be specified more than once. They are searched every time they are encountered. Archive files have a symbol table at the beginning of the archive. The **ld** will cycle through this symbol table until it has determined that it cannot resolve any more references from that library.

There are currently two different archive formats in use:

1. one is a random access library in use on 3B20S and VAX machines running UNIX system 5.0
2. the other is the old format library that must be searched linearly.

The old format library is in use on the PDP 11/70 and all machines running a pre-5.0 UNIX system. The link editor will make one search through a library in the old format, but will continue to search through a library in the new format until it has determined that it can resolve no more references from that library. Due to the different searching algorithms used, programs that are link edited on machines with different archive formats, and that are otherwise the same, may include files from libraries in a different order.

Be careful when using archive libraries in a subsystem loading environment. For a member of an archive, that is, an object file, to be included in a subsystem final load file, there must be a reference (within the subsystem being linked) to a symbol defined in that object file. The **−u** option can be used to create unresolved references that will force the loading of archive members.

Consider the following example:

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

If the **ld** 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.

4.3 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.
}
```

4.4 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. The **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:

1. Any output sections for which explicit bonding addresses were specified are allocated.

2. Any output sections to be included in a specific named memory are allocated. In both this and the succeeding step, each output section is placed into the first available space within the (named) memory with any alignment taken into consideration.

3. Output sections not handled by one of the above steps are allocated.

If all memory is contiguous and configured (the default case), and no SECTIONS directives are given, then output sections are allocated in the order they appear to 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.

4.5 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:

1. Intermediate link edits should contain only SECTIONS declarations and be concerned only with the formation of output sections from input files and input sections. No binding of output sections should be done in these runs.

2. All allocation and memory directives, as well as any assignment statements, are included only in the final **ld** call.

4.6 Space Limitations

Global and external symbols are kept in a symbol table in order to resolve references across input files. On the PDP-11/70, space is limited, so the link editor uses a Software Demand Paging scheme to store the symbols. When the number of global symbols is less than 600, the entire symbol table resides in memory. When the number exceeds 600, however, the symbol table is paged to a file and users will notice a degradation in performance. Structure tag names, structure elements, automatic and static variables do not appear in the symbol table and their number has minimal affect on **ld** performance.

The link editor will also be built as two processes on the PDP-11/70 due to space limitations. On all other machines the link editor is one process.

4.7 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:

1. It does not participate in the memory allocation for output sections. As a result, it takes up no memory and does not show up in the memory map (the **−m** option) generated by the **ld**.
2. It may overlay other output sections and even unconfigured memory. DSECTs may overlay other DSECTs.
3. The global symbols defined within the **dummy section** are relocated normally. That is, they appear in the output file's symbol table with the same value they would have had if the DSECT were actually loaded at its virtual address. DSECT-defined symbols may be referenced by other input sections. Undefined external symbols found within a DSECT cause specified archive libraries to be

searched and any members which define such symbols are link edited normally (i.e., not in the DSECT or as a DSECT).

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

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

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

A section with the **type** of NOLOAD differs in only one respect from a normal output section:

1. 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.

4.8 Output File Blocking

There are two options which can be used to affect the "*physical file offsets*" of the information written to the output file by **ld**:

1. The BLOCK option permits any output section to be aligned in the output fiel at a specified n-byte boundary.
2. The **−B** option causes "padding sections" to be generated in the output file.

Both features were provided explicitly for the use of **ldp**, which constructs pfiles for DMERT. The output sections of a pfile have certain requirements in terms of physical file offsets which can be met by using these two **ld** options.

The BLOCK option, which can be 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).

The **−B** option will cause **ld** to generate special sections in the output file. These sections, called "*padding sections*," take no part in the link edit process. They are supplied to force a certain type of physical file offset alignment of the non-"padding sections."

A "padding section" is an output section consisting of x bytes of zero (where x is supplied by the **−B** option). It has no physical address associated with it. A "padding section" will be output by **ld** after every non-"padding section" which meets either of the following two conditions:

1. It is of zero length.
2. It is comprised entirely of uninitialized **.bss** sections.

Unlike conventional **.bss** sections, the zero bytes making up a "padding section" are actually written to the output file.

"Padding sections" are ignored by **ld** if found in any input file.

4.9 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 NON-RELOCATABLE INPUT FILE.

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

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

4.10 The PATCH List

The **ld** option **−p** indicates that a "PATCH list" is to be built in the optional header field of the output file. The "PATCH list" is a C data structure which looks like this:

```
struct plist
{
    long blk_cnt          /*number of blocks*/
    union pentry
    {
        struct
        {
            long              blk_addr;    /*block address*/
            long              blk_size;    /*size of block*/
            unsigned short    blk_scnum;   /*section*/
            char              blk_type;    /*type of block*/
            char              blk_pad;     /*padding*/
        } type01;            /*FREEE or OLD FCN*/
        struct
        {
            long              blk_addr;    /*block address*/
            long              blk_ndx;     /*fcn tv index*/
            unsigned short    blk_scnum;     /*section*/
            char              blk_type;    /*type of block*/
            char              blk_size;    /*size of block*/
        } type02;            /*DECF*/
    } block[1];
};

#define PLIST       struct plist
#define PENTRY      union pentry
#define PLSIZE      sizeof(PLIST)
#define PESIZE      sizeof(PENTRY)
```

If given the **−p** option, **ld** will serch for all output sections whose name is of the form **.patch***nn*, where *nn* is a two-digit decimal integer. For each such output section, one pentry structure will be built.

The "PATCH list" is put at the end of the optional header field in the output file.

If necessary, so as to prevent any overlapping with the **−X** and **−h** options, the size of the optional header will be increased by the amount of space required by the "PATCH list."

The "PATCH list" is currently built only by 3bldp as part of its pfile construction. This list is used to perform incremental field update and function replacement.

4.11 The −ild Option

When the **−i***ld* option is used, the link editor will create a pair of dummy sections, DSECTs, for each unallocated, configured area of memory. These dummy sections will have unique names in the form of *.i_l_dnn* where *nn* is a two digit decimal integer in the range from 00 to 99, therefore at most 50 pairs of these sections will be created by the link editor. These sections identify the boundaries of the unused memory space. These sections are similar to **.bss** sections in that they do not contain any text or initialized data. The link editor also creates a dummy section named "**.history**." These sections are later used by the incremental link editor.

5. Error Messages

5.1 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.

5.2 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**.

5.3 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 **Id**
- Internal error: negative symbol table id
- Internal error: no symtab entry for DOT
- Internal error: split_scns, size of **sect** exceeds its new displacement.

5.4 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.

5.5 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 UniPlus$^+$ 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.

5.6 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.

5.7 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**.

5.8 Transfer Vector Error Messages

- ASSIGN slot num exceeds total TV size of num
- Attempt to assign tv slot to illegal symbol

- Defined symbol assigned a tv slot outside tv range
- Illegal ASSIGN slot number (0)
- Illegal multiple LENGTH fields in the TV directive
- Illegal multiple RANGE fields in the TV directive
- Illegal RANGE syntax
- Non-tv file name in transfer vector run
- RANGE num exceeds total TV size of num
- Supplied tv origin (num) does not equal the hard-wired tv origin (0x760000). 3bld only.
- Transfer vector file name in non-tv run
- tv fill symbol does not exist
- tv range allows num1 entries; num2 needed
- tv reference to non-tv symbol, addr address, index num, file name
- Two tv slot assignments for function name, slot1 and slot2
- Undefined symbol assigned a tv slot within tv range

5.9 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.

5.10 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.

- Can't create intermediate **ld** file name
- Can't open internal file name

  These two messages are possible only when the link editor is two processes. This would indicate that the temp directory (usually /tmp or /usr/tmp) is out of space, or that the link editor does not have permission to write in it.

- Cannot create output file **name** .

  The user may not have write permission in the directory where the output file is to be written.

- Failure to load pass 2 of **ld**

  This can only occur when the link editor is built as two processes (i.e., on the PDP 11/70). The most likely cause is that the second process is not where the first one thinks it is.

- File name has a section name which is a reserved **ld** identifier: **.tv**
- 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.
- 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.
- 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)**.

  See the part "Notes and Special Considerations."
- 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.

6. Syntax Diagram for Input Directives

The following tables contain a syntax diagram for input directives.

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> }
<section>	->	<name> <sec_options> : { <statement_list> } [<fill>] [<mem_spec>]
<group_options>	->	[<addr>] [<align_option>]

directives	->	expanded directives
<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>	->	* | / | %
	->	+ | -
	->	>> | <<
	->	== | != | > | < | <= | >=
	->	&
	->	|
	->	&&
	->	||

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**

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)

Chapter 8: COFF

CONTENTS

LIST OF FIGURES

Chapter 8
COFF –
THE COMMON OBJECT FILE FORMAT

1. Introduction

This Chapter describes the Common Object File Format (COFF). COFF is the output file produced on some UNIX™ systems by the assembler (**as**) and the link editor (**ld**). Since this format is used on several processors and operating systems, including the UniPlus+® Operating System, the word *common* is both descriptive and widely recognized.

The COFF is flexible enough to meet the demands of most jobs and even simple enough to be easily incorporated into existing projects. The following are some of COFF's key features:

- Applications may add system-dependent information to the object file without causing access utilities to become obsolete.
- Space is provided for symbolic information used by debuggers and other applications
- Users may make some modifications in the object file construction at compile time.

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

UNIX is a trademark of AT&T Bell Laboratories.
UniSoft and UniPlus$^+$ are registered trademarks of UniSoft Corporation.

- A file header
- Optional header information
- A table of section headers
- Data corresponding to the section header
- Relocation information
- Line numbers
- A symbol table
- A string table.

Figure 11.1 below shows the overall structure:

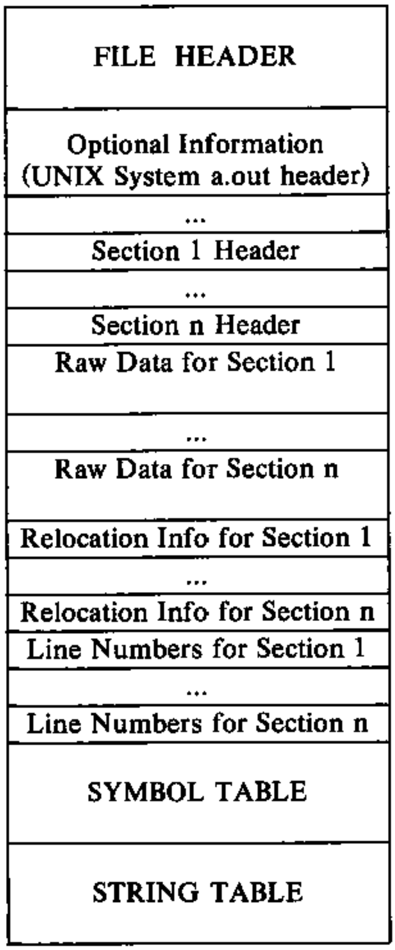

FILE HEADER
Optional Information (UNIX System a.out header)
...
Section 1 Header
...
Section n Header
Raw Data for Section 1
...
Raw Data for Section n
Relocation Info for Section 1
...
Relocation Info for Section n
Line Numbers for Section 1
...
Line Numbers for Section n
SYMBOL TABLE
STRING TABLE

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 link editor (**ld**) or if the relocation (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 compiler's (**cc**) **−g** option. 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.

2. Definitions and Conventions

Section	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 UniPlus$^+$ Operating System loads only the **.text**, and **.data** memory when the file is executed. The kernel clears the **.bss** section.
Physical Address	This is the physical location in memory where a section is loaded.
Virtual Address	This is the offset of a section with respect to the beginning of its segment or region. All relocatable references in a section assume that section occupies the virtual address at execution time.

3. File Header

The file header contains the 20 bytes of information shown in the following table. The last 2 bytes are flags used by **ld** and object file utilities. For more explicit information regarding the C language structure for the file header, see **filehdr(4)** in the *UniPlus*$^+$ *User Manual*, Sections 2-6.

BYTES	DECLARATION	NAME	DESCRIPTION
0-1	unsigned short	f_magic	Magic number as defined by the symbol MAGIC in the file *a.out.h.*
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.
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

Figure 8.2. File Header Contents

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 originally targeted for different systems. On a VAX system, the optional header is 28 bytes.

3.1 Magic Numbers

The "*magic number*" specifies the machine on which the object file is executable. The following is a table of the currently defined magic numbers.

MNEMONIC	MAGIC NUMBER	SYSTEM
MC68MAGIC	0520	M68000
U370WRMAGIC	0530	IBM 370 (writable text segments)
U370ROMAGIC	0535	IBM 370 (read-only sharable text segments)
N3B MAGIC	0550	3B™ 20S computers only.
FBOMAGIC	0560	WE™-32 (forward byte ordering)
RBOMAGIC	0565	WE-32 (reverse byte ordering)
VAXROMAGIC	0575	VAX™-11/750 and VAX-11780 (writable text segments)
VAXWRMAGIC	0570	VAX-11/750 and VAX-11/780 (read-only sharable segments)

Figure 8.3. Magic Numbers

VAX is a trademark of Digital Equipment Corporation.
3B and WE are trademarks of AT&T Bell laboratories.

3.2 Flags

The last 2 bytes of the file header are flags that describe the type of the object file. The UNIX version of COFF has no use for some of these, but they are included here for commonality. The currently defined flags are given in the following table:

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 file
F_LSYMS	00010	Local symbols stripped from file
F_MINMAL	00020	Not used by UNIX
F_UPDATE	00040	Not used by UNIX
F_SWABD	00100	This file has had its bytes *swabbed* (i.e. the bytes of symbol table name entries have been reversed.)
F_AR16WR	00200	Created on an AR16WR* machine, (PDP™-11/70).
F_AR32WR	00400	Created on an AR32WR** machine, (VAX-11/780).
F_AR32W	01000	Created on an AR32W*** machine, (M68000).
F_PATCH	02000	Not used by UNIX

Figure 8.4. File Header Flags

PDP is a trademark of Digital Equipment Corporation.
* AR16WR defines the machine architecture (AR) as 16 bits per word (16), right-to-left byte order with the least significant byte first (WR).
** AR32WR defines the machine architecture (AR) as 32 bits per word (32), right-to-left byte order with the least significant byte first (WR).
*** AR32W defines the machine architecture (AR) as 32 bits per word (32), left-to-right byte order with the most significant byte first (W).

3.3 File Header Declaration

The C structure declaration for the file header is given in the following table. This declaration may be found in the header file **filehdr.h**. See **filehdr(4)** in the *UniPlus+ User Manual*, Sections 2-6.

```
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 pointer to
                               symbol table */
      long        f-nsyms;  /* number entries in the
                               symbol table */
  unsigned short  f_opthdr; /* size of
                               optional header*/
  unsigned short  f_flags;  /* flags */
};

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

Figure 8.5. File Header Declaration

4. Optional Header Information

The template for optional information varies among different systems that use COFF. Applications place ALL system-dependent information into this record. This allows different operating systems access to information that only that operating system uses without forcing all COFF files to save space for that information. General utility programs (for example, the symbol table access library functions), can be made to work properly on any common object file by using the "size of optional header information" in bytes 16-17 of the file header **f_opthdr**.

4.1 Standard UNIX System a.out Header

By default, files produced by the link editor (**ld**) ALWAYS have a standard UNIX System **a.out** header in the optional header field. The VAX version of the optional header is 28 bytes. The fields of the optional header are described in the following figure:

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 uninitial-ized data in bytes
16-19	long int	entry	Entry point
20-23	long int	text_start	Base address of text
24-27	long int	data_start	Base address of data

Figure 8.6. Optional Header Contents

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 UniPlus$^+$ operating system are given in the following table:

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 8.7. UNIX Magic Numbers

The magic number for the UNIX Operating System is a machine-dependent constant that can be found in the header file a.out.h. See the manual page for **a.out(4)** in the *UniPlus⁺ User Manual*, Sections 2-6.

4.2 Optional Header Declaration

The C language structure declaration currently used for the UniPlus+ system **a.out** file header is given in the following table. This declaration may be found in the header file **aouthdr.h**.

```
typedef struct aouthdr {

    short magic;       /* magic number */

    short vstamp;      /* version stamp */

    long  tsize;       /* text size in bytes, padded
                          to full word boundary */

    long  dsize;       /* initialized data size */

    long  bsize;       /* uninitialized data size */

    long  entry;       /* entry point */

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

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

} AOUTHDR;
```

Figure 8.8. Aouthdr Declaration

5. Section Headers

Every object file has a table of section headers to specify the layout of data within the file. Every section in an object file also has its own header. The section header table consists of one entry for every section in the file. Each entry contains the following information about the section:

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 pointer to relocation entries
28-31	long int	s_lnnoptr	File pointer to line number entries
32-33	unsigned short	s_nreloc	Number of relocation entries
34-35	unsigned short	s_nlnno	Number of line number entries
36-39	long int	s_flags	Flags

Figure 8.9. Section Header Contents

The size of a section is always 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 UniPlus$^+$ Operating System function **fseek**(3S).

5.1 Flags

The lower 4 bits of the flag field indicate a section type.

MNEMONIC	FLAG	MEANING
STYP_REG	0x00	Regular section (allocated, relocated, loaded)
STYP_DSECT	0x01	Dummy section (not allocated, relocated, not loaded)
STYP_NOLOAD	0x02	Noload section (allocated, relocated, not loaded)
STYP_GROUP	0x04	Grouped section (formed from input sections)
STYP_PAD	0x08	Padding section (not allocated, not relocated, loaded)
STYP_COPY	0x10	Copy section (for a decision function used in updating fields; not allocated, not relocated, loaded, relocation and line number entries processed normally)
STYP_TEXT	0x20	Section contains executable text ONLY
STYP_DATA	0x40	Section contains initialized data ONLY
STYP_BSS	0x80	Section contains only uninitialized data

Figure 8.10. Section Header Flags

5.2 Section Header Declaration

The C structure declaration for the section headers is described in the following figure. This declaration may be found in the header file **scnhdr.h**. (See **scnhdr(4)** in the *UniPlus+ User Manual*, Sections 2-6 for more information.)

```
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 pointer to
                                   section raw data */

               long  s_relptr;  /* file pointer to
                                   relocation */

               long  s_lnnoptr; /* file pointer 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 8.11. Section Header Declaration

5.3 .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, 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.

6. Sections

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 compiler and the **as** assembler 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 the chapter on "LD – THE COMMON LINK EDITOR" in the *UniPlus*$^{+}$ *Programming Guide.*) allows users to:

- describe how input sections are to be combined;
- direct the placement of output sections; and
- 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 compiler are linked together (each containing the three sections **.text**, **.data**, and **.bss**), the output object file also contains the same three sections.

7. 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 following 10 byte format:

BYTES	DECLARATION	NAME	DESCRIPTION
0-3	long int	r_vaddr	(Virtual) address of reference
4-7	long int	r_symndx	symbol table index
8-9	unsigned short	r_type	Relocation type

Figure 8.12. 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 the following table:

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. (a VAX relocation type)
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 8.13. VAX and M68000 Relocation Types

On 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. In other words, the relative difference between the current segment's start address and the program's load address 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.

7.1 Relocation Entry Declaration

The structure declaration for relocation entries is given in the following table. This declaration may be found in the header file **reloc.h**.

```
struct reloc {
        long r_vaddr;           /* reference virtual address */
        long r_symndx;          /* index into symbol table */
        unsigned short r_type;  /* relocation type */
};

#define RELOC struct reloc
#define RELSZ  10
```

Figure 8.14. Relocation Entry Declaration

8. Line Numbers

When invoked with the **−g** option, the UniPlus$^+$ system compilers (**cc**, **f77**) generate 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.

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

Figure 8.15. 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.

8.1 Line Number Declaration

The following is the structure declaration currently used for line number entries.

```
struct lineno {

        union {
                long l_symndx;/* symbol table index
                                 of function name */
                long l_paddr; /* physical address
                                 of line number */
        } l_addr;
        unsigned short l_lnno;  /* line number */
};

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

Figure 8.16. Line Number Entry Declaration

9. Symbol Table

Because of symbolic debugging requirements, the order of symbols in the symbol table is very important. Symbols appear in the following sequence:

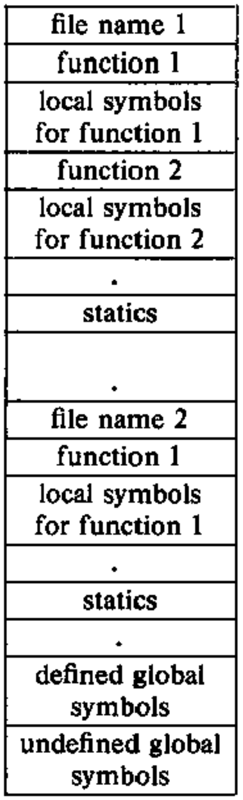


Figure 8.17. COFF Global Symbol Table

The word *statics* 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 name (null-padded), the structure value, the type, and other information.

9.1 Special Symbols

The symbol table contains some special symbols that are generated by the cc compiler, the as assembler, and other tools.

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
.xfake	dummy tag name for structure, union, or enumeran
.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 8.18. Special Symbols in the Symbol Table

Six of these special symbols occur in pairs. The **.bb** and **.eb** symbols indicate the boundaries of inner blocks. A **.bf** and **.ef** pair brackets each function; 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** compiler invents a name to be used in the symbol table. The name chosen for the symbol table is ***.x*fake**, where *x* is an integer. If there are three unnamed structures, unions, or enumerations in the source, their tag names are **.0fake**, **.1fake**, and **.2fake**.

Each of the special symbols has different information stored in the symbol table entry as well as the auxiliary entry.

9.2 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), such as *if*, *while* or *switch*.

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 following table shows this sequence:

.bb
local symbols for that block
.eb

Figure 8.19. Special Symbols

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

```
    {                                   /* 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 8.20. Nested Blocks

The symbol table would then look like the following:

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

Figure 8.21. Example of the Symbol Table

9.3 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 the following table:

function name
.bf
local symbol
.ef

Figure 8.22. 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 the following table:

function name
.target
.bf
local symbols
.ef

Figure 8.23. The Special Symbol .target

The **cc** compiler invents **.target** to store the function-returned structure or union. The symbol **.target** is an automatic variable with **pointer** type. Its value field in the symbol is always 0.

9.4 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 the following table:

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	char	_name	8 character null-padded name of either a pointer or 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 8.24. Symbol Table Entry Format

9.4.1 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 the following table.

The name of a symbol is currently limited to 8 characters, longer names are truncated by the cc compiler, Some special symbols are generated by the compiler and link editor, as discussed under the subheading "Special Symbols." The names of special symbols alwas start with a dot, such as **.file**, **.5fake** and **.bb**.

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 8.25. Name Field

Some special symbols are generated by the cc compiler and **ld** link editor as discussed under the subheading "Special Symbols." The VAX cc compiler prepends an underscore ("_") to all the user defined symbols it generates; the M68000 DOES NOT prepend an underscore. The M68000 prepends a "." to such symbol names (i.e., **.5fake**).

9.4.2 Storage Classes

The storage class field has one of the values described in the following table. 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
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 used only with the UNIX **cprs** utility
C_HIDDEN	106	like static, used to avoid name conflicts

Figure 8.26. Storage Classes

All of these storage classes except for C_ALIAS and C_HIDDEN are generated by the cc compiler or as assembler. These storage classes are not used by any UniPlus$^+$ system tools. The UNIX **cprs** (compress) utility generates the C_ALIAS mnemonic. This utility removes duplicated structure, union and enumeration definitions and puts ALIAS entries in their places.

There are some "dummy" storage classes defined in the header file which are only used internally by the cc compiler and the as assembler. These storage classes are:

- C_EFCN
- C_EXTDEF
- C_ULABEL
- C_USTATIC
- C_LINE

9.4.3 Storage Classes for Special Symbols

Some special symbols are restricted to certain storage classes listed in the following table:

SPECIAL SYMBOL	STORAGE CLASS
.file	C_FILE
.bb	C_BLOCK
.eb	C_BLOCK
.bf	C_FCN
.ef	C_FCN
.target	C_AUTO
.xfake	C_STRTAG, C_UNTAG, C_ENTAG
.eos	C_EOS
.text	C_STAT
.data	C_STAT
.bss	C_STAT

Figure 8.27. Storage Class by Special Symbols

Some storage classes are only used for certain special symbols.

STORAGE CLASS	SPECIAL SYMBOL
C_BLOCK	.bb, .eb
C_FCN	.bf, .ef
C_EOS	.eos
C_FILE	.file

Figure 8.28. Restricted Storage Classes

9.4.4 Symbol Value Field

The meaning of the value of a symbol depends on its storage class. This relationship is summarized in the following table:

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	offset
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 used by UNIX **cprs** utility
C_HIDDEN	relocatable address used by UNIX **cprs** utility

Figure 8.29. Storage Class and Value

If a symbol is the last symbol in the object file and has storage class C_FILE (**.file** symbol), the value of that symbol equals the symbol table entry index of the first global 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.

9.4.5 Section Number Field

Section numbers are listed in the following figure:

MNEMONIC	SECTION NUMBER	MEANING
N_DEBUG	−2	special symbolic debugging symbol
N_ABS	−1	absolute symbol
N_UNDEF	0	undefined external symbol
N_SCNUM	1-077767	section number where symbol was defined

Figure 8.30. 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.

9.4.6 Section Numbers and Storage Classes

Symbols having certain storage classes are also restricted to certain section numbers. They are summarized in the following table:

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 8.31. Section Number and Storage Class

9.4.7 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 and M68000 cc compilers generate 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

d6	d5	d4	d3	d2	d1	typ

Bits 0 through 3, called **typ**, indicate one of the fundamental types given in the following table:

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 8.32. 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 the following table.

MNEMONIC	VALUE	TYPE
DT_NON	0	no derived type
DT_PTR	1	pointer
DT_FCN	2	function
DT_ARY	3	array

Figure 8.33. Derived Types

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

char *func();

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

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

Here **tabptr** is a 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."

9.4.8 Type Entries and Storage Classes

Following are the type entries that are legal for each storage class:

STORAGE CLASS	----------"d" ENTRY----------			"typ" ENTRY BASIC TYPE
	F?*	A?*	P?*	
C_AUTO	no	yes	yes	Any except T_MOE
C_EXT	yes	yes	yes	Any except T_MOE
C_STAT	yes	yes	yes	Any except T_MOE
C_REG	no	no	yes	Any except T_MOE
C_LABEL	no	no	no	T_NULL
C_MOS	no	yes	yes	Any except T_MOE
C_ARG	yes	no	yes	Any except T_MOE
C_STRTAG	no	no	no	T_STRUCT
C_MOU	no	yes	yes	Any except T_MOE
C_UNTAG	no	no	no	T_UNION
C_TPDEF	no	yes	yes	Any except T_MOE
C_ENTAG	no	no	no	T_ENUM

* F? = Function?; * A? = Array? * P? = Pointer?

Figure 8.34. Type Entries by Storage Class (1 of 2)

STORAGE CLASS	----------"d" ENTRY----------			"typ" ENTRY BASIC TYPE
	F?*	A?*	P?*	
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

* F? = Function?; * A? = Array? * P? = Pointer?

Figure 8.35. Type Entries by Storage Class (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.

9.4.9 Structure for Symbol Table Entries

The C language structure declaration for the symbol table entry is given in the following table. 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 symbol table entry */
```

Figure 8.36. Symbol Table Entry Declaration

9.5 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. The following table lists auxiliary table entry formats by type and storage class.

NAME	STORAGE CLASS	TYPE ENTRY d2	TYPE ENTRY typ	AUX. ENTRY FORMAT
.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	Any except T_MOE	function
arrname	C_AUTO C_STAT C_MOS C_MOU C_TPDEF	DT_ARY	Any except T_MOE	array
.bb	C_BLOCK	DT_NON	T_NULL	begin-ning of block
.eb	C_BLOCK	DT_NON	T_NULL	end of block
.bf .ef	C_FCN	DT_NON	T_NULL	begin-ning and end of function
name related to structure union, enumera-tion	C_STAT C_MOS C_MOU C_TPDEF	DT_PTR DT_ARR DT_NON	T_STRUCT T_UNION, T_ENUM	name related to structure union, enumera-tion

Figure 8.37. Auxiliary Symbol Table Entries

In the preceding table, **tagname** means any symbol name including the special symbol **.xfake**, and **fcname** and **arrname** represent any symbol name.

Any symbol that satisfies more than one condition 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.

9.5.1 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.

9.5.2 Sections

The auxiliary table entries for sections have the format as shown in the following table:

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	—	dummy	unused (filled with zeroes)

Figure 8.38. Format for Auxiliary Table Entries

9.5.3 Tag Names

The auxiliary table entries for tag names have the format shown in the following table:

BYTES	DECLARATION	NAME	DESCRIPTION
0-5	–	dummy	unused (filled with zeros)
6-7	unsigned short	x_size	size of strucrt, union,and enumeration
8-11	–	dummy	unused (filled with zeroes)
12-15	long int	x_endndx	index of next entry beyond this structure, union, or enumeration
16-17	–	dummy	unused (filled with zeroes)

Figure 8.39. Tag Names Table Entries

9.5.4 End of Structures

The auxiliary table entries for the end of structures have the format shown in the following table:

BYTES	DECLARATION	NAME	DESCRIPTION
0-3	long int	x_tagndx	tag index
4-5	–	dummy	unused (filled with zeroes)
6-7	unsigned short	x_size	size of struct, union, or enumeration
8-17	–	dummy	unused (filled with zeroes)

Figure 8.40. Table Entries for End of Structures

9.5.5 Functions

The auxiliary table entries for functions have the format shown in the following table:

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 function
16-17	unsigned short	x_tvndx	index of the function's address in the transfer vector table (not used by UNIX Operating System.)

Figure 8.41. Table Entries for Functions

9.5.6 Arrays

The auxiliary table entries for arrays have the format shown in the following table:

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	–	dummy	unused (filled with zeroes)

Figure 8.42. Table Entries for Arrays

9.5.7 Beginning of Blocks and Functions

The auxiliary table entries for the beginning of blocks and functions have the format shown in the following table:

BYTES	DECLARATION	NAME	DESCRIPTION
0-3	–	dummy	unused (filled with zeroes)
4-5	unsigned short	x_lnno	C-source line number
6-11	–	dummy	unused (filled with zeroes)
12-15	long int	x_endndx	index of next entry past this block
16-17	–	dummy	unused (filled with zeroes)

Figure 8.43. Format for Beginning of Block and Function

9.5.8 End of Blocks and Functions

The auxiliary table entries for the end of blocks and functions have the format shown in the following table:

BYTES	DECLARATION	NAME	DESCRIPTION
0-3	–	dummy	used (filled with zeroes)
4-5	unsigned short	x_lnno	C-source line number
6-17	–	dummy	unused (filled with zeroes)

Figure 8.44. End of Block and Function Entries

9.5.9 Names Related to Structures, Unions, and Enumerations

The auxiliary table entries for structure, union, and enumerations symbols have the format shown in the following table:

BYTES	DECLARATION	NAME	DESCRIPTION
0-3	long int	x_tagndx	tag index
4-5	–	dummy	unused (filled with zeroes)
6-7	unsigned short	x_size	size of the structure, union or numeration
8-17	–	dummy	unused (filled with zeroes)

Figure 8.45. 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.

9.5.10 Auxiliary Entry Declaration

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

```
union auxent {

  struct {
        long x_tagndx;
        union {
              struct {
                    unsignedshort x_lnno;
                    unsignedshort x_size;
              } x_lnsz;
              long x_fsize;
        } x_misc;
        union {
              struct {
                    long x_lnnoptr;
                    long x_endndx;
              } x_fcn;
              struct {
                    unsignedshort x_dimen[DNUM];
              } x_ary;
        } x_fcnary;
              unsignedshort x_tvndx;
  } x_sym;
  struct {
        char x_fname[FILNMLEN];
  } x_file;
  struct {
        long x_scnlen;
        unsignedshort x_nreloc;
        unsignedshort x_nlinno;
  } x_scn;
  struct {
        long x_tvfill;
        unsignedshort x_tvlen;
        unsignedshort x_tvran[2];
  } x_tv;
}

#defineFILNMLEN 14
#defineDNUM     4
```

```
#defineAUXENT    union auxent
#defineAUXESZ    18
```

Figure 8.46. Auxiliary Symbol Table Entry

10. 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 shown in the following table:

28			
'l'	'o'	'n'	'g'
'—'	'n'	'a'	'm'
'e'	'—'	'1'	'\0'
'a'	'n'	'o'	't'
'h'	'e'	'r'	'—'
'o'	'n'	'e'	'\0'

Figure 8.47. String Table

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

11. Access Routines

Supplied with every standard UniPlus$^+$ 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, along with a summary of what is available, in the *UniPlus+ User Manual*, Sections 2-6, under **ldfcn(4)**.

Chapter 9: FORTRAN 77

CONTENTS

Chapter 9
FORTRAN 77

1. Introduction

This chapter describes the FORTRAN 77 compiler and run-time system as implemented on the UniPlus+® system.

Also described are the interfaces between procedures and the file formats assumed by the I/O system. (For more information on the I/O system see the chapters entitled "UNIX™ Implementation" and "UNIX I/O" in the *UniPlus+ Administrator Manual*).

2. Usage

2.1 UNIX System Commands

A UniPlus+ System FORTRAN 77 user should be familiar with the following commands:

f77 Usage: **f77 [options] files**
This command invokes the UniPlus+ System FORTRAN 77 compiler.

ratfor Usage: **ratfor [options] [files]**
This command invokes the Ratfor preprocessor.

efl Usage: **efl [options] [files]**
This command compiles a program written in Extended Fortran Language (EFL) into FORTRAN 77.

asa Usage: **asa [files]**
This command interprets the output of FORTRAN programs that utilize ASA carriage control characters.

fsplit Usage: **fsplit options files**
This command splits the named file(s) into separate files, with one procedure per file.

2.2 The f77 Command

The command to run the compiler is

```
f77 options file
```

The UniPlus+ FORTRAN compiler accepts several types of arguments:

1. Arguments whose names end with ".f" are taken to be Fortran 77 source programs; they are compiled, and each object program is left on the file in the current directory whose name is that of the source with a ".o" substituted for the ".f" suffix.
2. Arguments whose names end with ".r" or ".e" are taken to be Ratfor or EFL source programs, respectively.
3. Arguments whose names end with ".c" or ".s" are taken to be C or assembly source programs and are compiled or assembled, producing a ".o" file.

The **f77(1)** command is a general purpose command for compiling and loading FORTRAN and FORTRAN-related files into an executable module.

If EFL (compiler) and Ratfor (preprocessor) source files are given as arguments to the **f77** command, they will be translated into FORTRAN before being presented to this 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 unles the **−c** option is used.

NOTE: The **f77(1)** and **cc(1)** commands have slightly different link editing sequences. FORTRAN programs need two extra libraries — *libI77.a* and *libF77.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.

2.2.1 Options

The following options have the same meaning as in **cc(1)**. (See **ld(1)** for load-time options.)

−c	Suppress loading and produce ".o" files for each source file.
−g	Have the compiler produce additional symbol table information for **sdb(1)**. Also pass the **−lg** flag to **ld(1)**.
−w	Suppress all warning messages. If the option is **−w66**, only Fortran 66 compatibility warnings are suppressed.
−p	Prepare object files for profiling, see **prof(1)**.
−O	Invoke an object-code optimizer.
−S	Compile the named programs, and leave the assembler-language output on corresponding files with a ".s" suffix. (No ".o" is created.).
−o *output*	Name the final output file *output* instead of "a.out" (default).

The following options are peculiar to **f77**:

−onetrip	Compile DO loops that are performed at least once if reached. (Fortran 77 DO loops are not performed at all if the upper limit is smaller than the lower limit.)
−u	Make the default type of a variable "undefined" rather than using the default Fortran rules.
−C	Compile code to check that subscripts are within declared array bounds.
−F	Apply EFL and Ratfor preprocessor to relevant files, put the result in the file with the suffix changed to ".f", but do not compile.
−m	Apply the M4 preprocessor to each ".r" or ".e" file before transforming it with the Ratfor or EFL preprocessor.
−E *x*	Use the string *x* as an EFL option in processing ".e" files.
−R *x*	Use the string *x* as a Ratfor option in processing ".r" files.

Other arguments are taken to be either loader option arguments, F77-compatible object programs (typically produced by an earlier run), or libraries of F77-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name "a.out" (default).

3. 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.

3.1 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.

3.2 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.

3.3 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.

3.4 Recursion

Procedures may call themselves directly or through a chain of other procedures.

3.5 Automatic Storage

Two new keywords recognized are **static** and **automatic**. These keywords may appear in implicit statements or as "types" in type 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.

3.6 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 anywhere EXCEPT in one of the first six positions on the line, is treated as another kind of blank by the compiler.

3.7 Upper Case/Lower Case

In the FORTRAN 77 Standard, there are only 26 letters because FORTRAN is a one-case language, and the new compiler expects **lowercase** input.

By default, the compiler converts all uppercase characters to lowercase except those inside character constants. 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 compiler's **−U** option, keywords will be recognized ONLY if they appear in lowercase.

3.8 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.

3.9 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.

3.10 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)
\\"	Quotation mark (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.

3.11 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.

3.12 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.

3.13 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. For example:

```
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 are in violation of the 1977 Standard, the **−onetrip** compiler option causes loops whose initial value is greater than or equal to the limit value to be performed once.

3.14 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.

3.15 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.

3.16 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 *UniPlus*+ *User 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	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 *UniPlus+ User Manual.*

4. 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.

1. Double Precision Alignment
2. Dummy Procedure Arguments
3. T and TL Formats

4.1 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.

For 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 MUST be assumed on a bad boundary.

To load a double-precision quantity on some machines, it is necessary to use two separate operations.

1. The first operation is to move the upper and lower halves into the halves of an aligned "*temporary*".

2. The second operation is to load that double-precision temporary.

In order to store such a result, it is necessary to perform the above two operations in reverse order.

All double-precision real and complex quantities MUST fall on even word boundaries on machines with corresponding hardware requirements or if the source code must issue a diagnostic if a violation of the odd-boundary rule occurs.

4.2 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**. However, the same code is correct (in this regard) if there are no **character** arguments.

4.3 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 a record which has already been processed.

This compiler's implementation uses "seeks." Therefore, if the standard output 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.

5. Interprocedure Interface

The following sections are included to provide information necessary for writing C language procedures which call or are called by FORTRAN procedures. Specifically, it is important to understand the conventions with regard to the following:

1. Procedure Names
2. Data Representation
3. Return Values
4. Argument Lists

5.1 Procedure Names

On UNIX systems, the name of a common block for 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.

5.2 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.

5.3 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 —

1. a data address, and

2. 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, or *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.

Thus, the statement

```
call nret(*1, *2, *3)
```

is treated exactly as if it were the computed **goto**

```
goto (1, 2, 3),  nret( )
```

5.4 Argument Lists

All FORTRAN arguments are passed by address.

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:

1. Extra arguments for complex and character functions
2. Address for each datum or function
3. 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.

6. File Formats

6.1 File Structure

FORTRAN requires four kinds of external files:

1. sequential formatted
2. sequential unformatted,
3. direct formatted and
4. direct 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 ordinary files on the UNIX system (byte-addressable byte strings). 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.

6.2 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 10: RATFOR

CONTENTS

Chapter 10
RATFOR

1. Introduction

This chapter describes the **RATFOR** preprocessor (**ratfor(1)**). It is assumed that the user is familiar with the current implementation of FORTRAN 77 on the UniPlus$^+$ 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.

2. 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 UniPlus$^+$ system uses *&* in column 1), and **−C**, which causes RATFOR comments to be copied into the generated FORTRAN.

The program **rc(1M)** provides an interface to the **RATFOR(1)** command. This command is similar to **cc(1)**. Thus:

rc options files

compiles the files specified by **files**. Files with names ending in **.r** are **RATFOR** source; other files are assumed to be for the loader. The options **−C** and **−6x** described above are recognized, as are

−c	Compile only; don't load
−f	Save intermediate FORTRAN .f files
−r	RATFOR only; implies **−c** and **−f**
−2	Use big FORTRAN compiler (for large programs)
−U	Flag undeclared variables (not universally available).

Other options are passed on to the loader.

3. Statements

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.

3.1 "if-else"

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(1, 6) a, b }
else
    { sw = 1; write(1, 6) 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.

3.2 Nested "if"

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.

3.3 "switch"

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.

3.4 "do"

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.

3.5 "break" and "next"

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.

3.6 "while"

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.

3.7 "for"

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 RATFOR **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.

3.8 "repeat-until"

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).

3.9 "return"

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.

3.10 "define"

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 COLS 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

```
    write(6, 100); 100 format("hello")
```

is converted into

```
         write(6, 100)
    100  format(5hhello)
```

3.13 Translations

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.

==	is translated to	**.eq.**
!=	is translated to	**.ne.**
>	is translated to	**.gt.**
>=	is translated to	**.ge.**
<	is translated to	**.lt.**
<=	is translated to	**.le.**
&	is translated to	**.and.**
|	is translated to	**.or.**
!	is translated to	**.not.**

In addition, the following translations are provided for input devices with restricted character sets:

[	is translated to	{
]	is translated to	}
$(	is translated to	{
$)	is translated to	}

4. 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.

5. Example 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 11: EFL PROGRAMMING LANGUAGE

CONTENTS

Chapter 11
EFL – A PROGRAMMING LANGUAGE

1. 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. However, in spite of the fact that the name EFL originally stood for "Extended FORTRAN Language." The EFL compiler is much more than a simple preprocessor. The compiler attempts to diagnose all syntax errors, provide readable FORTRAN output, and to avoid a number of niggling FORTRAN restrictions.

EFL is especially useful for numeric programs, and permits the programmer to express complicated ideas in a comprehensible way, while permitting access to the power of the FORTRAN environment.

This is not a tutorial, but a general description and reference manual for the EFL Programming Language. The reader should have a fair degree of familiarity with some procedural language. There will be occasional references to Ratfor and to FORTRAN, but these may be ignored if the reader is unfamiliar with those languages.

2. Conventions

In examples and syntax specifications, **boldface** type is used to indicate literal words and punctuation, such as the EFL keywords **if**, **else**, **while** or **do**. Words in *italic* type indicate an item in a category, such as an *expression* or, as in the following example:

> **define** *name* definition

the italic font was used to indicate that the *name* in the **define** statement was not to be typed literally, but would be replaced with a name chosen by the programmer as appropriate to the definition being written in the program.

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*

3. Lexical Form

3.1 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	, ; : . + − * /
	= < > & ~ | $

Even though all of the examples herein are printed in lower case, letter case (upper or lower) is ignored except within strings. Thus, "a" and "A" are treated as the same character. An exclamation mark ("!") may be used in place of a tilde ("~") as the logical unary operator "complement." Square brackets ("[" and "]") may be used in place of braces ("{" and "}") for punctuation.

3.1.1 White Space

Outside of a character string or comment, any sequence of one or more spaces or tab characters acts as a single space and terminates a "token."

3.2 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 is signaled by an underscore.

3.3 Lines

EFL is a line-oriented language. Except in special cases where a continuation is made explicit by use of an underscore ("_"), 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. Diagnostic messages are labeled with the line number of the file on which they are detected.

3.3.1 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 if unbalanced braces or parentheses exist. Some compound statements are also continued automatically — these points are noted in the sections on executable statements.

3.4 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.

3.5 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. The sharp and succeeding characters on the line are discarded. A blank line is also a comment. Comments have no effect on execution.

NOTE: A sharp inside of a quoted string does NOT mark a comment.

3.6 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.

3.6.1 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.

3.6.2 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'"
```

3.6.3 Integer Constants

An integer constant is a sequence of one or more digits.

```
0
57
123456
```

3.6.4 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:

```
.I
I.
I.J
IE
I.E
.IE
I.JE
```

3.6.5 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.

3.6.6 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.**).

3.7 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.

4. Program Form

4.1 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.

4.2 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.)

4.3 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" for further information on blocks.

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
```

4.4 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. 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.

4.5 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 **error**: in the following:

```
             read(, x)
             if(x < 3) goto error
             ...
     error:  fatal("bad input")
```

5. 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.

5.1 Basic Types

The basic types are

logical	A *logical* quantity may take on the two values *true* and *false.*
integer	An *integer* may take on any whole number value in a machine-dependent range.
field(*m*:*n*)	A *field* quantity is an integer restricted to a particular closed interval ([*m*:*n*]).
real	A *real* quantity is a floating point approximation to a real or rational number. Real quantities are represented as single precision floating point numbers.
complex	A *complex* quantity is an approximation to a complex number, and is represented as a pair of reals.
long real	A *long real* is a more precise approximation to a rational. Long reals are double precision floating point numbers.

long complex	A *long complex* quantity is an approximation to a complex number, and is represented as a pair of long reals.
character(*n*)	A **character** quantity is a fixed-length string of *n* characters.

5.2 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 $(= 7.9 \times 10^{-6})$
14e9 $(= 1.4 \times 10^{10})$

The following are valid **long real** constants

7.9d−6 $(= 7.9 \times 10^{-6})$
5d3

A character constant is a quoted string.

5.3 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.

Each variable has certain attributes:

1. Storage Class
2. Scope
3. Precision

5.3.1 Storage Class

A variable's *storage class* is the association of its name and its location. A storage class can either be *transitory* or *permanent.*

- *Transitory* association is achieved when arguments are passed to procedures.
- Other associations are considered *permanent* or *static*.

5.3.2 Scope of Names

The scope of a variable may be either *global* or *local.*

1. The names of common areas are *global*, and *global* variables may be used anywhere in the program.
2. All other names are considered *local* to the block in which they are declared.

5.3.3 Precision

Floating point variables are either of *normal* or *long* precision. Normal precision is 32 bits; long precision is 64 bits. This attribute may be stated independently of the basic type.

5.4 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.

5.5 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
     }
```

6. 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 the sections "Unary Operators" and "Binary Operators."

```
->   .
**
*  /       unary +   -   ++   --
+  -
<  <=  >  >=  ==  ~=
&  &&
|  ||
$
=  +=  -=  *=  /=  **=   &=  |=  &&=  ||=
```

Examples of expressions are

```
a<b && b<c
-(a + sin(x)) / (5+cos(x))**2
```

6.1 Primaries

Primaries are the basic elements of expressions. They include constants, variables, array elements, structure members, procedure invocations, input/output expressions, coercions, and sizes.

6.1.1 Constants

Constants are described in the section "Constants" under "Data Types and Variables."

6.1.2 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.

6.1.3 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)
```

6.1.4 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)
```

6.1.5 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."

6.1.6 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.

6.1.7 Coercions

An expression of one precision or type may be *coerced*, that is, 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.

6.1.8 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*)

6.2 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.

6.3 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.

6.4 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.

6.4.1 Logical

The only logical unary operator is complement (˜). This operator is defined by the equations

> ˜ true = false
> ˜ false = true

6.5 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.

6.5.1 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   c = complex
l r = long real   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.)

6.5.2 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.

6.6 Relational Operators

There are six relations between arithmetic quantities. These operators are not associative.

EFL Operator	Meaning	
<	$<$	less than
<=	$\leqslant$	less than or equal to
==	$=$	equal to
~=	$\neq$	not equal to
>	$>$	greater than
>=	$\geqslant$	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.

6.7 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.

6.8 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.nth**

refers to the **nth** member of the **st** structure starting at the *i*-th element of the array **place.**

6.9 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)
```

6.10 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.

7. Declarations

Declarations statement describe the meaning, shape, and size of named objects in the EFL language.

7.1 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
    }
```

7.2 Attributes

7.2.1 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 $n > m$.

7.2.2 Arrays

The dimensionality may be declared by an **array** attribute

array($b_1,\ldots,b_n$)

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)

7.2.3 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 **xxs** and a character string.

7.2.4 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**.

7.2.5 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.

7.2.6 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.

7.3 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.

7.4 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.

8. 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.

8.1 Expression Statements

8.1.1 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.

8.1.2 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
```

8.2 Blocks

A block is a compound statement that acts as a single 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)
}
```

8.3 Test Statements

A *test statement* permits execution of another statement or group of statements based on the outcome of a conditional expression.

There are several forms of test statements:

1. if statements
2. if-else statements
3. select statements

8.3.1 If Statement

The simplest of the test statements is the **if** statement, of form

> **if** (*logical-expression*) □ *statement*

First, the logical expression is evaluated; if it is true, then the *statement* is executed. Otherwise *statement* will be skipped.

8.3.2 If-Else

A more general statement is of the form

> **if** (*logical-expression*) □ *statement-1* □
> **else** □ *statement-2*

Just as with the "if" statement, the logical expression is evaluated and 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 **if** which is not already followed by an **else**.

A more common use of the "**if-else**" test statement is the sequential test:

```
if(x==1)
    k = 1
else if(x==3  |  x==5)
    k = 2
else
    k = 3
```

There may be any number of **else if** statements in an "**if-else**" statement to test for several conditions, although if more than 2 **else ifs** are needed, a *select statement* is often used instead.

8.3.3 Select Statement

Much like the **switch** statement in the C shell or **case** statements in many programming languages, a **select statement** is used to direct the branching of a program based on the result of a conditional or arithmetic expression. A select statement 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.

```
select( x)
     {
     case 1:
          k = 1
     case 3,5:
          k = 2
     default:
          k = 3
     }
```

8.4 Loops

The **loop** constructs, (**while**, **for**, **repeat**, **repeat-until** and **do**), provide an efficient way to repeat an operation or series of operations. Termination of a loop is generally initiated by the failure of a logical or iterative test statement. Although the **while** loop is the simplest construct, and consequently the most frequently used, each construct has its own strengths to be exploited in a given application.

8.4.1 While Statement

This construct has the form

> **while** (*logical-expression*) □ *statement*

First, the *logical-expression* is evaluated; if it is true, *statement* is executed, and the *logical-expression* is evaluated again. If *logical-expression* is false, *statement* is not executed and program execution continues at the next statement.

8.4.2 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.

8.4.3 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.

8.4.4 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 expression*; if the *logical expression* 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.

8.4.5 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 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
```

8.5 Branch Statements

It is not considered good programming practice to use branch statements if a loop construct can be used instead. However, if you must use a branch statement, EFL provides a few for your convenience.

8.5.1 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.

8.5.2 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. For example:

```
break 3
```

transfers control to the statement following the third loop and/or **select** surrounding the statement.

It is possible to specify the type of construct to which control is to be transferred, i.e. **for**, **while**, **repeat**, **do**, or **select**. For example:

```
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.

8.5.3 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.

8.5.4 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*)

8.6 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.

8.6.1 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*.

8.6.2 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.

8.6.3 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.

8.6.4 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.

8.6.5 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.

8.6.6 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.

9. Procedures

Procedures are the basic unit of an EFL program, and provide the means of segmenting a program into separately compilable and named parts.

9.1 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.

9.2 End Statement

Each procedure terminates with a statement

> **end**

9.3 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.

9.4 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.

9.5 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.

9.5.1 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)
```

9.5.2 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.

9.5.3 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	$atan2(x,y) = tan^{-1} x/y$

9.5.4 Other Generic Functions

The **sign** function takes two arguments of identical type. The **mod** function yields the remainder of its first argument when divided by its second.

sign$(x,y) = sgn(y)|x|$.
mod(x,y)

These functions accept integer and real arguments.

10. Atavisms

The following constructs are included to ease the conversion of old FORTRAN or Ratfor programs to EFL.

10.1 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. Such a line is called an *escape line* and must begin with a percent sign ("%"). *Escape lines* are copied through to the output without change, except that the percent sign is removed. 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.

10.2 Call Statement

A subroutine call may be preceded by the keyword **call.**

```
call joe
call work(17)
```

10.3 Obsolete Keywords

The following keywords are recognized as synonyms of EFL keywords:

FORTRAN	EFL
double precision	**long real**
function	**procedure**
subroutine	**procedure** (*untyped*)

10.4 Numeric Labels

Standard statement labels are identifiers. A numeric (positive integer constant) label is also permitted; the colon is optional following a numeric label.

10.5 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
```

10.6 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.

10.7 Goto Statement

In unconditional and computed **goto** statements, it is permissible to separate the **go** and **to** words, as in

```
go to xyz
```

10.8 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.

10.9 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)

10.10 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.

10.11 Equivalence

A statement of the form

equivalence $v_1, v_2, \dots, v_n$

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.

10.12 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

11. 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.

11.1 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.**

11.2 Input Language Options

The **dots** option determines whether the compiler recognizes **.lt.** and similar forms. The default setting is **no.**

11.3 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.

11.4 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.

11.5 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

11.6 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.

11.7 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.**

11.8 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.

No Hollerith strings will be passed as subroutine arguments if **hollinc_ll=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.

12. 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.

12.1 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.

12.2 Matrix Multiplication

The following procedure multiplies the m × n matrix a by the $n \times p$ matrix b to give the $m \times p$ matrix c. The calculation obeys the formula $c_{ij} = \Sigma \; a_{ik} \; b_{kj}$.

```
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
```

12.3 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**.

12.4 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 implemented 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 NODE     tree(currentnode)
define STACK     stackframe(stackdepth)

#   nextstate values
define DOWN     1
define LEFT     2
define RIGHT     3

#   initialize stack with root mode
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
```

13. 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).

13.1 Primitives

Certain EFL operations cannot be implemented in portable FORTRAN, so a few machine-dependent procedures must be provided in each environment.

13.1.1 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**.

13.1.2 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.

14. 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:

1. 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.
2. The input/output syntax is quite different in the two languages, and there is no FORMAT statement in EFL.
3. 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
- data structures

EFL permits more general forms for expressions, and provides a more uniform syntax. For example, EFL does not have the restrictions on subscript or DO expressions forms as do FORTRAN and Ratfor.

15. Compiler

15.1 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.

15.2 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.

15.3 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
```

16. 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.

16.1 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.

16.2 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. That is, a procedure name may ONLY be passed as an argument or be invoked — it cannot be stored.

16.3 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, and the language itself simplified considerably, if certain cases could have been handled by pointers. There are several ways of "simulating" pointers by using subscripts, but this raises problems of external variables and initialization.

16.4 Recursion

FORTRAN procedures are not recursive, so it was not practical to permit EFL procedures to be recursive. As in the case of pointers, recursion may be simulated in EFL, but not without considerable effort.

16.5 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 12: LINT

CONTENTS

Chapter 12

LINT –

A C PROGRAM CHECKER

1. Introduction

The C program checker, **lint**, can be used to detect bugs, obscurities, inconsistencies and portability of C programs. It is generally stricter than the C compiler, which accepts constructions without complaint that **lint** considers wasteful or error-prone. The **lint** program is also much stricter with regard to the C language type rules. Also, **lint** accepts multiple files and library specifications and checks them for consistency.

In addition to the many thorough checking mechanisms themselves, **lint** offers the facility of suppressing them if they are not necessary for a given application.

1.1 Usage

The **lint** command has the form:

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

- *options* are optional flags to control **lint** checking and messages
- "files" are the files to be checked by **lint**. Naturally, files containing C language programs must end with a **.c** suffix since this is mandatory for both lint and the C compiler.
- *library-descriptors* are the names of libraries to be used in checking the program.

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.

1.2 Options

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

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

−a	Use this option to suppress messages concerning the assignment of "long" values to variables which are not "long." This option is often useful as there are a number of legitimate reasons for assigning "longs" to "ints."
−b	Use this option to suppress messages concerning "break" statements which are unreachable. For example, programs generated by **yacc** and especially **lex** may have hundreds of unreachable break statements. If the C compiler optimizer were used, these unreached statements would be of little importance, but the resulting messages would clutter up the **lint** output. In this case, the **−b** option is especially useful.
−c	This option is no longer available.
−h	Use this option only to suppress the use of "heuristics." Heuristics is used by default to check for wasteful or error-prone constructions and to detect bugs. For example, by default **lint** prints messages about variables which are declared in inner blocks in a way that conflicts with their use in outer blocks. Though this construction is considered "legal," it remains bad programming style, and frequently a bug.

−l *y*	Use this option to specify libraries you wish included and checked by **lint**. 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 MUST all begin with the comment:

/* LINTLIBRARY */

This comment must then be 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.

−**n**	Use this option to suppress checking for compatibility with either the standard or the portable **lint** library. In effect, this option supresses ALL library checking.
−**O** *name*	Use this option to create a **lint** library from input files named **llib-l***name***.ln**.
−**p**	Use this option to check a program's portability to other dialects of C language. This option checks a file containing descriptions of standard library routines which are expected to be portable.
−**u**	Use this option to suppress messages concerning function and external variables which are either used and not defined or defined and not used.

The comment:

```
/* VARARGS */
```

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

```
/* VARARGS2 */
```

will cause ONLY the first two arguments to be checked.

When **lint** is applied to some but not all files out of a collection which are to be loaded together, information about unused or undefined variables is more distracting than helpful. 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 is especially useful to suppress the spurious messages which might otherwise appear.

−v Use this option to suppress messages concerning unused function arguments. To suppress such messages for one function only, place the following comment in the program before that function:

```
/* ARGSUSED */
```

−x Use this option to suppress messages concerning variables referred to by external declarations but never used.

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.

2. Types of Messages

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

2.1 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.

It is possible to suppress messages regarding variables which are declared through explicit **extern** statements but are never referenced. The statement:

extern *double* sin();

will evoke no comment if **sin** is never used, providing the **−x** option is used. (**Note:** this agrees with the semantics of the C compiler.)

In some cases, these unused external declarations might be of some interest, in which case you can use **lint** without the **−x** option.

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 including 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:

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.

2.2 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 earlier 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.

2.3 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 the section on "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 if given the **−b** option. 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 without the **−b** option.

2.4 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 ;
```

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" when the control flow of a program reaches the end of the function. For example:

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

In this example, if the result of "*a*" is 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.

2.5 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.

2.6 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.

2.7 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:

```
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**

2.8 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 disabled by the **−a** option. However, if using the **−p** option to detect possible portability problems, **lint** may print the message, "warning: conversion from long may lose accuracy," in spite of the use of the **−a** option.

2.9 Strange Constructions

Several perfectly legal, but somewhat strange, constructions are detected by **lint**. The messages hopefully encourage better code quality, clearer style, and may even point out bugs. The **−h** option is used to suppress the majority of these checks.

For example:

```
*p++ ;
```

the * does nothing. This provokes the message "null effect" from **lint**. For example:

```
unsigned x ;
if( x < 0 ) ...
```

results in a test that will never succeed. For another example:

```
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:

```
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.

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. Although this is considered "legal," it remains bad style, usually unnecessary and frequently a bug.

2.10 Old Syntax

Several forms of older syntax are now illegal. These fall into two classes −

1. assignment operators and
2. initialization.

The older forms of assignment operators (e.g., =+, =−, ...) could cause ambiguous expressions. For example:

```
a =−1 ;
```

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:

```
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. For example:

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

This is free of any possible syntactic ambiguity.

2.11 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.

2.12 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:

```
    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 13: SDB

CONTENTS

LIST OF FIGURES

Chapter 13
SDB –
SYMBOLIC DEBUGGING PROGRAM

1. Introduction

This chapter describes the symbolic debugger **sdb(1)** as implemented for C language and Fortran 77 programs on the UniPlus+® 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.

2. 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, **sdb** takes three arguments on the command line:

1. The first argument is the name of the executable file to be debugged; it defaults to **a.out** when not specified. Even with the new **COFF** format, the executable file will be named a.out. However, **sdb** will not work on old a.out format files. Only COFF files may be used with **sdb**.

2. The second argument is the name of the core file, defaulting to *core*;

3. The third argument 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.

The following is a typical example of **sdb** usage. The first display, Figure 13.1, is the source file used to create the output file used to illustrate the use of **sdb**. The second figure, Figure 13.2, is an illustration of a session with **sdb**. Commands in **bold** are to be input explicitly; responses from **sdb** and comments are in roman for clarity.

```
$ 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
```

Figure 13.1. Input File Used for SDB Example

```
$ 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=0x7ffff50,envp=0x7ffff58)[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 13.2. Example of SDB Usage

2.1 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=0x7ffff54,envp=0x7ffff5c)  [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.

2.2 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 cn 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:

a	Print characters starting at the variable's address until a null is reached.
c	Character.
d	Decimal.
f	32-bit single-precision floating point.
g	64-bit double-precision floating point.
i	Interpret as a machine-language instruction.
o	Octal.
p	Pointer to function.
s	Assume variable is a string pointer and print characters starting at the address pointed to by the variable until a null is reached.
u	Decimal unsigned.
x	Hexadecimal.

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.

3. 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 UniPlus$^+$ 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.

3.1 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.

CTRL-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.

3.2 Changing the Source File or Function

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

3.3 Changing the Current Line in the Source File

The **z** and **CTRL-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.

4. A Controlled Testing Environment

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.

4.1 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.

4.2 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.

4.3 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 from a dump.

5. 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.

5.1 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 **CTRL-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**.

5.2 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**.

5.3 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.

Colophon

Composed at UniSoft Systems Inc.
on the UniPlus$^+$ Operating System
Designed by the Documentation Department
Printed in Times Roman on Sequoia Matt