PRO/VENIX™

UNIX™
for

Professional™

Programming Guide

digital software

PRO/VENIX™

for the Professional

Programming Guide

Developed by:

VenturCom, Inc.
215 First Street
Cambridge, MA 02142

Digital Equipment Corporation
Maynard, MA 01754

First Printing

The software described in this manual is distributed as part of Digital Equipment Corporation's Digital Classified Software (DCS) Program. This program enables software developers to submit their software products to Digital for testing according to Digital quality standards for third party software. This software product has met the DCS standard specified in the software product description (SPD) for this product. You should refer to the SPD for information about these standards, the hardware and software required to run this product, and warranties (if any warranty is available).

The software described in this manual is furnished under a license and may only be used or copied in accordance with the terms of that license. This manual is reproduced with the permission of VenturCom, Inc.

Except as may be stated in the SPD for this product, no responsibility is assumed by Digital or its affiliated companies for use or reliability of this software, or for errors in this manual or in the software. Additional support and/or warranty services may be available from the developer of this software product. Digital has no connection with, and assumes no responsibility or liabilities in connection with these services.

This manual is subject to change without notice and does not constitute a commitment by Digital.

VENIX is a trademark of VenturCom, Inc.
UNIX is a trademark of AT&T Technology, Inc.

The following are trademarks of Digital Equipment Corporation:

DEC	DECwriter	Professional	VAX
DECmate	DIBOL	Rainbow	VMS
DECnet	MASSBUS	RSTS	VT
DECsystem-10	PDP	RSX	Work Processor
DECSYSTEM-20	P/OS	UNIBUS	digital
DECUS			

The PRO/VENIX† Documentation Set

The PRO/VENIX documentation set consists of the following manuals:

PRO/VENIX Installation and System Manager's Guide

The set up and maintenance of PRO/VENIX are described in the installation sections. Other articles explain the UNIX-to-UNIX‡ communications systems. The "System Maintenance Reference Manual" contains reference pages for devices and system maintenance procedures (sections (7) and (8)).

PRO/VENIX User Guide

The *User Guide* contains tutorials for newcomers to PRO/VENIX, covering basic use of the system, the editor **vi** and use of the command language interpreters.

PRO/VENIX Document Processing Guide

The line and screen editors and **nroff**-related text formatting utilities are described in the Document Processing Guide. Topics include: line editor **ed**, and stream editor **sed**; the text formatter **nroff**; the **nroff**-preprocessors **tbl** and **neqn**.

PRO/VENIX Programming Guide

The chapters in the *Programming Guide* explicate the different programming languages for VENIX.

---

† VENIX is a trademark of VenturCom, Inc.

‡ UNIX is a trademark of Bell Laboratories.

PRO/VENIX Support Tools Guide

This guide includes tools for programming, such as the compiler-writing languages Yacc and Lex, the M4 Macro processor, the program development utility Make, and the desk calculator programs DC and BC.

PRO/VENIX User Reference Manual

This is a complete and concise reference for the PRO/VENIX system. This volume contains write-ups on all PRO/VENIX commands.

PRO/VENIX Progammer Reference Manual

The reference pages in this volume include system calls, library functions, file formats, miscellaneous functions and games.

Contents

INTRODUCTION

The *Programming Guide* describes the programming languages, libraries and support tools available on the VENIX system. The languages include C, Fortran, Pascal and PDP-11 assembler. The following paragraphs contain a brief description of each chapter.

Chapter 1, **USING THE C COMPILER**, is a short guide to using the **cc** command and provides some tips on programming in the C language.

Chapter 2, **VENIX PROGRAMMING**, describes the programming interface to the operating system and the standard I/O library.

Chapter 3, **C LANGUAGE**, provides a summary of the grammar and rules of the C programming language.

Chapter 4, **CODE MAPPING UNDER VENIX**, describes an overlaying scheme for large programs.

Chapter 5, **PROGRAM CHECKER — "lint"**, describes a program which checks for syntax errors, type violations, and portability problems in C programs.

Chapter 6, **A TUTORIAL INTRODUCTION TO ADB**, describes a symbolic debugging program that is used to debug compiled C language programs.

Chapter 7, **FORTRAN 77**, describes the implementation of Fortran 77 on the VENIX system in terms of the variations from the American National Standard.

Chapter 8, **RATFOR**, is a description of the Ratfor preprocessor. This preprocessor provides a means for writing Fortran in a fashion similar to the C language.

Chapter 9, **USING VU-PASCAL**, is an introduction to the Pascal compiler/interpreter on the VENIX system.

Chapter 10, **VU-PASCAL REFERENCE MANUAL**, is a detailed description of Vu-Pascal as it compares to the 1980 ISO Pascal standard.

Chapter 11, **SCREEN UPDATING AND CURSOR MOVEMENT OPTIMIZATION**, describes the **curses** package that provides the programmer with screen-oriented programming capabilities.

Chapter 12, **VENIX ASSEMBLER REFERENCE MANUAL**, is a brief description of the PDP-11 assembler provided with the VENIX system.

Throughout this document, each reference of the form **name**(7), or **name**(8) refers to entries in the *Installation and System Manager's Guide*. Each reference of the form **name**(1) refers to entries in the *User Reference Manual*. All other references to entries with sections (2) through (6) are contained in the *Programmer Reference Manual*.

Contents

Chapter 1

USING THE C COMPILER

1.1 INTRODUCTION

"C" is the major programming language used with VENIX and other UNIX-derived operating systems. Most UNIX software — from high level applications to the kernel itself — is written in C, and most users find the language powerful and convenient.

The language standard is set by Kernighan and Ritchie's *The C Programming Language*; every VENIX installation should have at least one copy. The following briefly describes how to compile C programs with the **cc** command. Following the introduction there are a few tips on using C. This chapter tries to cover problems that commonly trip new C programmers.

For anything more than minimal C programming, read the **VENIX PROGRAMMING** chapter in this manual, which describes the use of VENIX through C. While *The C Programming Language* is an accurate and (mostly) complete description of C itself, it should be used cautiously as a reference to I/O, command usage, or VENIX interfacing. This chapter and those in the other VENIX manuals cover these areas more accurately.

1.2 USING THE C COMPILER

The simplest way to compile the program **prog1.c** is with the command:

cc prog1.c

All C source file names must end with a ".c". If no errors are found, this produces an executable file called **a.out**, which can be executed by typing simply:

a.out

For example, if **prog1.c** is the program:

```
main(){
        printf("hello, world\n");
}
```

then the sequence of commands:

```
cc prog1.c
a.out
```

produces the output:

```
hello, world
```

If any errors are found in your program, you will receive messages giving the line number at fault, and type of error encountered. In this case, no **a.out** file is produced. Edit your program and try again.

Calling your executable file **a.out** is dangerous, because subsequent compilations of other programs could overwrite it. To prevent this from happening, rename your **a.out** to a unique name which will prevent it from being overwritten, with the **mv** command:

```
mv a.out prog1
```

Alternatively, you can use the **−o** ("rename output") flag with the **cc** command to initially name the executable file something other than **a.out**.

```
cc −o prog1 prog1.c
```

This gives the same net result as the commands:

```
cc prog1.c
mv a.out prog1
```

but is more succinct.*

If you have a medium-to-large sized program, you will probably wish to divide it into several different files. Each of these files may be separately compiled to object-code level, and then linked to executable form later. The advantage to this approach is that you can correct errors and make modifications to your files one by one, without having to recompile the total source after each change. The **−c** ("compile only") flag instructs the **cc** command to compile the source to object form, and not produce an executable file. The object file corresponding to each source file will be given the same name but with a ".o" suffix instead of a ".c". (This is always the case; the **−o** flag can not be used with **−c** to rename object files as it can for executable files.) For example, if a program is divided into the source files **p1.c**, **p2.c**, and **p3.c**, then the command:

```
cc −c p1.c p2.c p3.c
```

will produce three object files **p1.o**, **p2.o**, and **p3.o**. If you now wish to produce an executable file, you can simply enter:

```
cc −o newprog p1.o p2.o p3.o
```

to produce an executable file **newprog**. The **cc** command recognizes that the ".o" files are object code, not source, and acts appropriately. Now if a change needs to be made to **p2.c**, you can recompile it as above with:

```
cc −c p2.c
```

and then run **cc** again with all three object files as before.

---

1. * One subtle difference between renaming the output with **−o** and with **mv** is that in the latter case any previous **a.out** is overwritten, while in the former it is not touched.

There are several other flags that may be useful:

−O This flag causes the compiler to produce optimized output. Since compiling takes slightly longer with this flag, it is usually used only after a program is debugged and in finished form.

−lxxx This flag causes the VENIX library *xxx* (such as the math or other standard library) to be used when linking. Unlike other flags, the **−l** always comes at the end of the command line. For example, if math library routines need be used, then a command line might be something like:

cc p1.c p2.c −lm

−f This flag must be used if your program uses floating point arithmetic, and there is no hardware floating point support on your machine.

−p This causes your program to produce profiling information, indicating the number of times each routine is called. This can be useful for optimizing the program speed. Each time your program is run, a file **mon.out** is produced which contains this information. It can be examined with the **prof**(1) program.

−P Runs only the macro preprocessor on your program, producing for each '.c' source a '.i' file containing the preprocessed file. All '#define's and other '#' lines will have been evaluated and the substitutions done; no '#' lines or comments will be left in the file. To maintain constant line numbering, '#' or comment lines removed will be left blank, but not removed.

−n This option is passed on to the loader, and causes the program's data area to be moved upwards to the first possible 4k word boundary following the end of text. This type of program, known as "pure", can have its text area shared by multiple users running it at one time. This saves on memory when two or more users are running the program; it does, however, reduce the available data space since the data area is moved to the 4k word boundary.

−i This option is passed to the loader, and causes the program text and data space to be placed in separate address spaces. This can only be used on split I/D space computers. This program is "pure" and will have its text shared as described above.

−m Uses code-mapping. This is an advanced option to allow unusually large programs to run. See code-mapping documentation.

1.3 ERROR DIAGNOSTICS

Compiling errors may well be redundant. If a parenthesis or brace is missing on a particular line, for example, a number of different errors may be produced on subsequent lines. Check the *User Reference Manual* pages on **cc** and **ld** for a complete list of messages and what they mean.

The C compiler, unlike Pascal or many other compilers, is not very strict about use of variable types or arguments passing to functions. Generally, it will not complain if pointers of different types are mixed, nor will it protest if a function is called with the wrong number or types of arguments.

This freedom can be convenient, but it can also lead to errors, especially with beginning programmers. For stricter enforcement of the rules, use the lint program, described briefly in the *User Reference Manual* and, in more detail, in Chapter 5 of the *Programming Guide*.

1.4 WHAT cc DOES

The **cc** program does not handle compiling or linking itself, but acts as a master control program to call the appropriate compile/link passes. The following passes are run to produce an executable file from a source file:

1. C preprocessor (**/lib/cpp**) runs handle "#include"s and "#define"s.
2. First phase of C-compiler proper (**/lib/c0**) produces intermediate code.
3. Second phase of C-compiler proper (**/lib/c1**) produces assembly code.

4. Optional optimizing phase of C-compiler (**/lib/c2**) produces better assembly code.
5. Assembler (**/bin/as** and **/lib/as2**) produces binary object code.
6. Linker/loader (**/bin/ld**) links object modules and libraries, and produces an executable file.

1.5 TIPS ON C PROGRAMMING

The following brief hints may help in C programming:

1. Medium or large programs should be divided into separate files for partial compilation, as described earlier. The archive program **ar**(1) can be used to conveniently store object files.
2. When using routines mentioned in sections two and three of the *Programmer Reference Manual* be sure to use any "#include"s given on the page, and to use variable types consistent with those described in the synopsis. For example, the synopsis of the routine **fopen**(3) is:

```
#include <stdio.h>

FILE *fopen(filename, type)
char *filename, *type;
```

This means that **fopen** returns a pointer of type "FILE" (defined in <stdio.h>), and that both of its arguments are pointers to chars (i.e., strings). The statement:

```
#include <stdio.h>
```

should occur at the beginning of any file containing a call to **fopen()**.

Functions that return values other than integers should be declared explicitly in the files that use them. With **fopen()**, for example, you should put a statement:

```
FILE *fopen();
```

before **fopen()**. (Actually, this is done automatically in <stdio.h>, but it's not a bad idea to declare it yourself. Other functions may not have "#include" files to automatically declare them.)

3. Be aware of how you are allocating variable space. For example, the statement:

struct x ndata;

declares one structure called "ndata" of type "x". This allocates storage space for a structure the size of "x". Suppose you have a routine called "fillin()" which takes a pointer to a structure of type "x", and fills the structure with information. Fillin() could then be called as:

fillin(&ndata);

and you can be confident that "&ndata" is a valid place to store data.

BUT, if you declared something as:

struct x *pdata;

you are merely declaring a pointer to type "x", and NOT allocating any space for x. If you don't explicitly assign "pdata" to point to free space (for example, by setting it equal to the address of "ndata", or using a **malloc**(3) to allocate space), then passing it to "fillin()" as in:

fillin(pdata)

won't work. Although you are correctly calling fillin() with a pointer to type "x", you haven't set the pointer to an allocated memory. The routine fillin(), then will blindly place its structure wherever "pdata" happened to address, perhaps right in the middle of your program.

There are various ways to declare strings and pointers:

char a[10];

Allocates 10 bytes for an empty array called "a". The name "a" is a constant; it uses no data space itself, and can not be changed.

```
char *a;
```

Allocates space for pointer "a" only. The name "a" refers to a variable, which can be set to any pointer value.

```
char a[10] = "hello";
```

Allocates 10 bytes for an array called "a", and fills in the first six bytes with the string "hello" followed by a null. The name "a" is a constant.

```
char *a = "hello";
```

Allocates space for pointer "a", and sets it to point to six bytes somewhere filled with the string "hello" followed by a null.

```
char a[] = "hello";
```

Allocates an array "a" just big enough to hold the six bytes for "hello" and a null. The difference between this and the previous declaration is that in this case "a" is a constant and can not be changed; in the declaration above, "a" is a pointer variable which can later be set to point somewhere else.

4. For faster code, use pointers instead of indexes when moving through an array. For example, the code:

```
char a[100];
int i;
   .
   .
   .
for (i = 0; i < 100; ++i)
            a[i] = ...
```

causes two additions on each loop: one to increment "i", and one to add "i" to index "a[]". This could be rewritten:

```
char  a[100];
char  *ptr;
   .
   .
   .
for  (ptr  =  a;  ptr  <  a  +  100;  )
              *ptr++  =  ...
```

Here only one addition need be done (to increment "ptr").

Note that in the "for" loop above, the expression "a + 100" is the sum of two constants. It is therefore evaluated at compile time, not during each execution of the loop. If a variable were used in its place, the expression would have to be evaluated on each loop. If this were the case, the programmer would clearly do better to evaluate the expression once before the loop, and use the result in the "for" statement.

Using pointers instead of arrays does make C code harder to understand for beginners. It is of course your own preference as to which is most important: code efficiency or readability.

5. Register variables can also be used to speed program execution. Up to three registers are available, and they are best used for the most frequently accessed variables. A register can hold an int, char, or pointer.

6. It is better not to declare arrays, particularly large ones, inside a function. Since these kinds of variables are handled on the stack, placing arrays there will cause the stack to expand. This expansion can take a little execution time; it can also (in rare cases) lead to execution errors. (This warning does not apply to variables declared "static" inside a function, since they are not placed on the stack.)

Contents

Chapter 2

VENIX† PROGRAMMING

This chapter is an introduction to programming on the VENIX system. The emphasis is on how to write programs that interface to the operating system, either directly or through the standard I/O library. The topics discussed include

* handling command arguments
* rudimentary I/O; the standard input and output
* the standard I/O library; file system access
* low-level I/O: open, read, write, close, seek
* processes: exec, fork, pipes
* signals — interrupts, etc.

There is also an appendix which describes the standard I/O library in detail.

2.1 INTRODUCTION

This chapter describes how to write programs that interface with the VENIX operating system. This includes programs that use files by name, that use pipes, that invoke other commands as they run, or that attempt to catch interrupts and other signals during execution.

---

† VENIX is a trademark of VenturCom, Inc.

All of the system calls and interface routines mentioned in the following pages can also be found in sections two and three of the *Programmer Reference Manual*. If you will be programming in C, you must be able to read the language roughly up to the level described in the chapter **C LANGUAGE**. Some of the material in the following sections is based on topics covered more carefully there.

2.2 BASICS

2.2.1 Program Arguments

When a C program is run as a command, the arguments on the command line are made available to the function **main** as an argument count **argc** and an array **argv** of pointers to character strings that contain the arguments. By convention, **argv[0]** is the command name itself, so **argc** is always greater than 0.

The following program illustrates the mechanism: it simply echoes its arguments back to the terminal. (This is essentially the **echo** command.)

```
main(argc, argv)        /* echo arguments */
int argc;
char *argv[];
{
        int i;

        for (i = 1; i < argc; i++)
                printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

argv is a pointer to an array whose individual elements are pointers to arrays of characters; each is terminated by **\0**, (null) so they can be treated as strings. The program starts by printing **argv[1]** and loops until it has printed them all. Specifically, the arguments are **argv[1]** to **argv[argc−1]**.

The argument count and the arguments are parameters to **main**. If you want to keep them around so other routines can get at them, you must copy them to external variables.

Note carefully the declaration **char *argv[]**. The **argv[]** indicates that it is an array; the **char *** says that each element of the array is a pointer to type **char**, i.e., a character string. It is important to understand that in C, character strings and pointers to type **char** are exactly the same.

2.2.2 The Standard Input and Standard Output

The simplest input mechanism is to read the "standard input," which is generally the user's terminal. The function **getchar** returns the next input character each time it is called. A file may be substituted for the terminal by using the < convention: if **prog** uses **getchar**, then the command line

prog <file

causes **prog** to read *file* instead of the terminal. **prog** itself need know nothing about where its input is coming from. This is also true if the input comes from another program via the pipe mechanism.

otherprog | prog

provides the standard input for **prog** from the standard output of **otherprog**.

getchar returns the value **EOF** when it encounters the end of file (or an error) on whatever you are reading. The value of **EOF** is normally defined to be -1, but it is unwise to take any advantage of that knowledge. As will become clear shortly, this value is automatically defined for you when you compile a program, and need not be of any concern.

Similarly, **putchar(c)** puts the character **c** on the "standard output," which is also by default the terminal. The output can be captured on a file by using > ; if **prog** uses **putchar**,

prog >outfile

writes the standard output on *outfile* instead of the terminal. *outfile* is created if it doesn't exist; if it already exists, its previous contents are overwritten. And a pipe can be used:

prog | otherprog

puts the standard output of **prog** into the standard input of **otherprog**.

The function **printf**, which formats output in various ways, uses the same mechanism as **putchar**, so calls to **printf** and **putchar** may be intermixed in any order; the output will appear in the order of the calls.

Similarly, the function **scanf** provides for formatted input conversion; it will read the standard input and break it up into strings, numbers, etc., as desired. **scanf** uses the same mechanism as **getchar**, so calls to them may also be intermixed.

Many programs read only one input and write one output; for such programs I/O with **getchar, putchar, scanf**, and **printf** may be entirely adequate, and it is almost always enough to get started. This is particularly true if the VENIX pipe facility is used to connect the output of one program to the input of the next. For example, the following program strips out all ASCII control characters from its input (except for newline and tab).

```
#include <stdio.h>

main()      /* ccstrip: strip non-graphic characters */
{
   int c;
   while ((c = getchar()) != EOF)
     if ((c >= ' ' && c < 0177) || c == '\t' || c == '\n')
        putchar(c);
   exit(0);
}
```

The line

```
#include <stdio.h>
```

should appear at the beginning of each source file. It causes the C compiler to read a file (**/usr/include/stdio.h**) of standard routines and symbols that includes the definition of **EOF**.

If it is necessary to treat multiple files, you can use **cat** to collect the files for you:

```
cat file1 file2 ... | ccstrip >output
```

and thus avoid learning how to access files from a program. By the way, the call to **exit** at the end is not necessary to make the program work properly, but it assures that any caller of the program will see a normal termination status (conventionally 0) from the program when it completes. The section "Signals" discusses status returns in more detail.

2.3 THE STANDARD I/O LIBRARY

The "Standard I/O Library" is a collection of routines intended to provide efficient and portable I/O services for most C programs. The standard I/O library is available on most systems that support C, so programs that confine their system interactions to its facilities can be transported from one system to another essentially without change.

This section discusses the basics of the standard I/O library. The appendix contains a more complete description of its capabilities.

2.3.1 File Access

The programs written so far have all read the standard input and written the standard output, which are assumed to be predefined. The next step is to write a program that accesses a file that is NOT already connected to the program. One simple example is **wc**, which counts the lines, words and characters in a set of files. For instance, the command

```
wc x.c y.c
```

prints the number of lines, words and characters in *x*.c and *y*.c and the totals.

The question is how to arrange for the named files to be read — that is, how to connect the file system names to the I/O statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be "opened" by the standard library function **fopen**. **fopen** takes an external name (like *x*.c or *y*.c), does some housekeeping and negotiation with the operating system, and returns an internal name which must be used in subsequent reads or writes of the file.

This internal name is actually a pointer, called a file pointer, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don't need to know the details, because part of the standard I/O definitions obtained by including **stdio.h** is a structure definition called **FILE**. The only declaration needed for a file pointer is exemplified by

```
FILE        *fp, *fopen();
```

This says that **fp** is a pointer to a **FILE**, and **fopen** returns a pointer to a **FILE**. **FILE** is a type name, like **int**, not a structure tag. In actual programs, the declaration for **fopen** is done for you in **stdio.h**; however, you must declare your own file pointers as type **FILE**.

The actual call to **fopen** in a program is

```
fp = fopen(name, mode);
```

The first argument of **fopen** is the *name* of the file, as a character string. The second argument is the *mode*, also as a character string, which indicates how you intend to use the file. The only allowable modes are read ("**r**"), write ("**w**"), or append ("**a**").

If a file that you open for writing or appending does not exist, it is created (if possible). Opening an existing file for writing causes the old contents to be discarded. Trying to read a file that does not exist is an error, and there may be other causes of error as well (like trying to read a file when you don't have permission). If there is any error, **fopen** will return the null pointer value **NULL** (which is defined as zero in **stdio.h**).

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which **getc** and **putc** are the simplest. **getc** returns the next character from a file; it needs the file pointer to tell it which file. Thus

```
c = getc(fp)
```

places in **c** the next character from the file referred to by **fp**; it returns **EOF** when it reaches end of file. **putc** is the inverse of **getc**:

putc(c, fp)

puts the character c on the file fp and returns c. **getc** and **putc** return **EOF** on error.

When a program is started, three files are opened automatically, and file pointers are provided for them. These files are the standard input, the standard output, and the standard error output; the corresponding file pointers are called **stdin**, **stdout**, and **stderr**. Normally these are all connected to the terminal, but may be redirected at runtime to files or pipes as described in **AN INTRODUCTION TO THE SHELL** in the *User Guide* (which describes in more detail how to redirect different I/O channels). **stdin**, **stdout** and **stderr** are predefined in the I/O library as the standard input, output and error files; they may be used anywhere an object of type **FILE *** can be. They are constants, however, NOT variables, so don't try to assign to them.

With some of the preliminaries out of the way, we can now write **wc**. The basic design is one that has been found convenient for many programs: if there are command-line arguments, they are processed in order. If there are no arguments, the standard input is processed. This way the program can be used stand-alone or as part of a larger process.

```
#include <stdio.h>

main(argc, argv)        /* wc: count lines, words, chars */
int argc;
char *argv[];
{
        int c, i, inword;
        FILE *fp, *fopen();
        long linect, wordct, charct;
        long tlinect = 0, twordct = 0, tcharct = 0;
```

```
        i = 1;
        fp = stdin;
        do {
                if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
                        fprintf(stderr, "wc: can't open %s\n", argv[i]);
                        continue;
                }
                linect = wordct = charct = inword = 0;
                while ((c = getc(fp)) != EOF) {
                        charct++;
                        if (c == '\n')
                                linect++;
                        if (c == ' ' || c == '\t' || c == '\n')
                                inword = 0;
                        else if (inword == 0) {
                                inword = 1;
                                wordct++;
                        }
                }
                printf("%7ld %7ld %7ld", linect, wordct, charct);
                printf(argc > 1 ? " %s\n" : "\n", argv[i]);
                fclose(fp);
                tlinect += linect;
                twordct += wordct;
                tcharct += charct;
        } while (++i < argc);
        if (argc > 2)
           printf("%7ld %7ld %7ld total\n", tlinect, twordct, tcharct);
        exit(0);
}
```

The function **fprintf** is identical to **printf**, except that the first argument is a file pointer that specifies the file to be written.

The function **fclose** is the inverse of **fopen**; it breaks the connection between the file pointer and the external name that was established by **fopen**, freeing the file pointer for another file. Since there is a limit on the number of files that a program may have open simultaneously, it's a good idea to free things when they are no longer needed. There is also another reason to call **fclose** on an output

file — it flushes the buffer in which **putc** is collecting output. **fclose** is called automatically for each open file when a program terminates normally.

2.3.2 Error Handling — Stderr and Exit

stderr is assigned to a program in the same way that **stdin** and **stdout** are. Output written on **stderr** appears on the user's terminal even if the standard output is redirected. **wc** writes its diagnostics on **stderr** instead of **stdout** so that if one of the files can't be accessed for some reason, the message finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program actually signals errors in another way, using the function **exit** to terminate program execution. The argument of **exit** is available to whatever process called it (see section 6), so the success or failure of the program can be tested by another program that uses this one as a sub-process. By convention, a return value of 0 signals that all is well; non-zero values signal abnormal situations.

exit itself calls **fclose** for each open output file, to flush out any buffered output, then calls a routine named **_exit**. The function **_exit** causes immediate termination without any buffer flushing; it may be called directly if desired.

2.3.3 Miscellaneous I/O Functions

The standard I/O library provides several other I/O functions besides those we have illustrated above.

Normally output with **putc**, etc., is buffered (except to **stderr**); to force it out immediately, use **fflush(fp)**.

fscanf is identical to **scanf**, except that its first argument is a file pointer (as with **fprintf**) that specifies the file from which the input comes; it returns **EOF** at end of file.

The functions **sscanf** and **sprintf** are identical to **fscanf** and **fprintf**, except that the first argument names a character string instead of a file pointer. The conversion is done from the string for **sscanf** and into it for **sprintf**.

fgets(buf, size, fp) copies the next line from **fp**, up to and including a newline, into **buf**; at most **size−1** characters are copied; it returns **NULL** at end of file. **fputs(buf, fp)** writes the string in **buf** onto file **fp**.

The function **ungetc(c, fp)** "pushes back" the character **c** onto the input stream **fp**; a subsequent call to **getc**, **fscanf**, etc., will encounter **c**. Only one character of pushback per file is permitted.

setbuf(fp, buf) causes the given buffer to be used instead of an automatically allocated one. When used with terminal I/O, output will be buffered in 512 byte units. This may be desirable for non-interactive programs as it greatly improves the efficiency of output.

2.4 LOW-LEVEL I/O

This section describes the bottom level of I/O on the VENIX system. The lowest level of I/O in VENIX provides no buffering or any other services; it is in fact a direct entry into the operating system. You are entirely on your own, but on the other hand, you have the most control over what happens. And since the calls and usage are quite simple, this isn't as bad as it sounds.

It is important not to confuse this low-level I/O with standard I/O. As described in the next section, the low-level I/O routines connect files to file descriptors, which are analogous to, but quite different from, file pointers. If you try to mix the two, all sorts of problems will occur (this is a common difficulty among new VENIX programmers). All the low-level routines are described in section two of the *Programmer Reference Manual*; the standard I/O routines are found in section three. (Another easy way to distinguish the two routines is to remember that many of the standard I/O routines have the letter "f" in them (like **fopen**, **printf**, **fget**, ...) whereas the low-level I/O routines generally don't. This is not a hard rule, though; the routine **getc**, for example, is standard I/O.)

2.4.1 File Descriptors

In the VENIX operating system, all input and output is done by reading or writing files, because all peripheral devices, even the user's terminal, are files in the file system. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called "opening" the file. If you are going to write on a file, it may also be necessary to create it. The system checks your right to do so (Does the file exist? Do you have permission to access it?), and if all is well, returns a small positive integer called a file descriptor. Whenever I/O is to be done on the file, the file descriptor is used instead of the name to identify the file. (This is roughly analogous to the use of READ(5,...) and WRITE(6,...) in Fortran.) All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

The file pointers discussed in section 3 are similar in spirit to file descriptors, but file descriptors are more fundamental. A file pointer is a pointer to a structure that contains, among other things, the file descriptor for the file in question.

Since input and output involving the user's terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the "shell") runs a program, it opens three files, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error output. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and writes file descriptors 1 and 2, it can do terminal I/O without worrying about opening the files.

If I/O is redirected to and from files with < and >, as in

prog <infile >outfile

the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Similar observations hold if the input or output is associated with a pipe. Normally file descriptor 2 remains attached to the terminal, so error messages can go there. In all cases, the file assignments are changed by the shell, not by the program. The program does not need to know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

2.4.2 Read and Write

All input and output is done by two functions called **read** and **write**. For both, the first argument is a file descriptor. The second argument is a buffer in your program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are

n_read = read(fd, buf, n);

n_written = write(fd, buf, n);

Each call returns a byte count which is the number of bytes actually transferred. On reading, the number of bytes returned may be less than the number asked for, because fewer than *n* bytes remained to be read. (When the file is a terminal, **read** normally reads only up to the next newline, which is generally less than what was requested.) A return value of zero bytes implies end of file, and **−1** indicates an error of some sort. For writing, the returned value is the number of bytes actually written; it is generally an error if this isn't equal to the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two most common values are 1, which means one character at a time ("unbuffered"), and 512, which corresponds to a physical blocksize on many peripheral devices. This latter size will be most efficient, but even character at a time I/O is not inordinately expensive.

Putting these facts together, you can write a simple program to copy its input to its output. This program will copy anything to anything, since the input and output can be redirected to any file or device.

```
#define     BUFSIZE     512 /* best size for PDP-11 VENIX */

main()                              /* copy input to output */
{
            char        buf[BUFSIZE];
            int         n;

            while ((n = read(0, buf, BUFSIZE)) > 0)
                        write(1, buf, n);
            exit(0);
}
```

If the file size is not a multiple of **BUFSIZE**, some **read** will return a smaller number of bytes to be written by **write**; the next call to **read** after that will return zero.

It is instructive to see how **read** and **write** can be used to construct higher level routines like **getchar**, **putchar**, etc. For example, here is a version of **getchar** which does unbuffered input.

```
#define     CMASK     0377          /* for making char's > 0 */

getchar()       /* unbuffered single character input */
{
      char c;

      return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

c must be declared **char**, because **read** accepts a character pointer. However, **getchar** itself returns an integer value, so that the integer constant **EOF** (**−1**) can be differentiated from any character. For this reason, the character being returned must be masked with **0377** to ensure that it is positive; otherwise sign extension may make it negative when it is passed as an integer. (The constant **0377** is appropriate for the PDP-11 but not necessarily for other machines.)

The second version of **getchar** does input in big chunks, and hands out the characters one at a time.

```
#define   CMASK   0377      /* for making char's > 0 */
#define   BUFSIZE 512

getchar()             /* buffered version */
{
          static char buf[BUFSIZE];
          static char *bufp = buf;
          static int  n = 0;

          if (n == 0) {                  /* buffer is empty */
                    n = read(0, buf, BUFSIZE);
                    bufp = buf;
          }
          return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

2.4.3 Open, Creat, Close, Unlink

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system entry points for this, **open** and **creat** [sic].

open is rather like the **fopen** discussed in the previous section, except that instead of returning a file pointer, it returns a file descriptor which is just an **int**.

```
int fd;

fd = open(name, rwmode);
```

As with **fopen**, the **name** argument is a character string corresponding to the external file name. The access mode argument is different, however: **rwmode** is 0 for read, 1 for write, and 2 for read and write access. **open** returns **−1** if any error occurs; otherwise it returns a valid file descriptor.

It is an error to try to **open** a file that does not exist. The entry point **creat** is provided to create new files, or to re-write old ones.

```
fd = creat(name, pmode);
```

returns a file descriptor if it was able to create the file called **name**, and **−1** if not. If the file already exists, **creat** will truncate it to zero length; it is not an error to **creat** a file that already exists.

If the file is brand new, **creat** creates it with the protection mode specified by the **pmode** argument. In the VENIX file system, there are nine bits of protection information associated with a file, controlling read, write and execute permission for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is most convenient for specifying the permissions. For example, 0755 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the VENIX utility **cp** , a program which copies one file to another. (The main simplification is that our version copies only one file, and does not permit the second argument to be a directory.)

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644      /* RW for owner, R for group, others */

main(argc, argv)        /* cp: copy f1 to f2 */
int argc;
char *argv[];
{
        int             f1, f2, n;
        char            buf[BUFSIZE];
```

```
        if (argc != 3)
                error("Usage: cp from to", NULL);
        if ((f1 = open(argv[1], 0)) == -1)
                error("cp: can't open %s", argv[1]);
        if ((f2 = creat(argv[2], PMODE)) == -1)
                error("cp: can't create %s", argv[2]);

        while ((n = read(f1, buf, BUFSIZE)) > 0)
                if (write(f2, buf, n) != n)
                        error("cp: write error", NULL);
        exit(0);
}


error(s1, s2)                   /* print error message and die */
char *s1, *s2;
{
        printf(s1, s2);
        printf("\n");
        exit(1);
}
```

As stated earlier, there is a limit (currently 15) on the number of files which a program may have open simultaneously. Accordingly, any program which intends to process many files must be prepared to re-use file descriptors. The routine **close** breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. Termination of a program via **exit** or return from the main program closes all open files.

The function **unlink(filename)** removes the file **filename** from the file system, analogous to the **rm** command. In fact, the **rm** command calls **unlink** itself.

2.4.4 Random Access — Seek and Lseek

File I/O is normally sequential: each **read** or **write** takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call **lseek** provides a way to move around in a file without actually reading or writing:

```
lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is **fd** to move to position **offset**, which is taken relative to the location specified by **origin**. Subsequent reading or writing will begin at that position. **offset** is a **long**; **fd** and **origin** are **int**'s. **origin** can be 0, 1, or 2 to specify that **offset** is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append to a file, seek to the end before writing:

```
lseek(fd, 0L, 2);
```

To get back to the beginning ("rewind"),

```
lseek(fd, 0L, 0);
```

Notice the **0L** argument; it could also be written as **(long) 0**.

With **lseek**, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file.

```
get(fd, pos, buf, n)   /* read n bytes from position pos */
int fd, n;
long pos;
fchar *buf;
{
        lseek(fd, pos, 0);      /* get to pos */
        return(read(fd, buf, n));
}
```

2.4.5 I/O Control

VENIX allows the user to change terminal and other device driver characteristics, such as line speeds or data collection rates, through the system call

```
ioctl(fildes, request, argp)
struct sgttyb *argp;
```

where **fildes** is the file descriptor for the device being set (which must previously have been opened); **request** is one of a number of requests defined in the

include file **<sgtty.h>**, such as "return current values", "set new values", "return I/O queue character count", etc.; and **argp** is a pointer to a structure defined for terminals as

```
struct sgttyb {
        char sg_ispeed;          /* input speed */
        char sg_ospeed;          /* output speed */
        char sg_erase;           /* erase character */
        char sg_kill;            /* kill character */
        int  sg_flags;           /* mode flags */
};
```

While the remainder of this discussion on I/O control concerns only terminal modes (which have the most extensive control possibilities), other devices such as A/D's, D/A's, or real-time clocks have their own particular modes and flags, and will in general use different structures. Definitions of these structures can be found in the device driver description in **DEVICES**, of the *Installation and System Manager's Guide.*

One frequent use of I/O control is in setting characteristics of a user's terminal, such as line speeds, simulation of tabs, etc.; these can all be specified through different bits in **sg_flags**.

Two **request** values in the **ioctl** call are often used for manipulating the modes: **TIOCGETP** and **TIOCSETP**. The first is used to find out (GET) the current modes, and return them in the buffer pointed to by **argp**. The second SETs the modes to that specified in the pointed-to buffer.

A typical application for this might be to temporarily turn off echoing on the user's terminal while a confidential piece of information is typed. A simple routine to do this follows:

```
#include <sgtty.h>

getcode(){              /* get confidential code */
        struct sgttyb ttybuf;
        int flags, num;

        ioctl(0, TIOCGETP, &ttybuf);      /* get old values */
        flags = ttybuf.sg_flags;      /* save old state of flags */
        ttybuf.sg_flags &= ~ECHO;      /* turn off echoing */
        ioctl(1, TIOCSETP, &ttybuf);      /* reset */
        printf("enter code: ");
        scanf("%d", &num);

        ttybuf.sg_flags = flags;      /* reset old values */
        ioctl(0, TIOCSETP, &ttybuf);

        return(num);                  /* return information */
}
```

A **TIOCGETP** is done to find the previous state of the file with descriptor 0 (the standard input) which is saved; a **TIOCSETP** is then done to turn off echoing. After the information has been read, the original modes are set with another **TIOCSETP**. Note that echoing does not prevent messages being written by the program from appearing on the terminal; it only keeps information the user types from being typed back.

The flags that can be set or read in are as follows:

CRT	**0100000**	**Terminal is a CRT**
SCROLL	**0040000**	**Output stops automatically every 20 lines**
XTABS	**0006000**	**Expand tabs to spaces on output**
RAW	**0000040**	**Raw mode: 8 bit interface (turns off SCROLL, CRT, XTABS, CRMOD, LCASE and CBREAK)**
EVENP	**0000200**	**Enable even parity**
ODDP	**0000100**	**Enable odd parity**
CRMOD	**0000020**	**Map CR into LF; echo LF as CR-LF**
ECHO	**0000010**	**Echo (full duplex)**
LCASE	**0000004**	**Map upper to lower case (Escapes work)**

CBREAK	**0000002**	**Return each character as soon as typed**
TANDEM	**0000001**	**Automatic flow control**

CRT mode indicates that a 'delete' is echoed as a backspace-space-backspace to erase the character from the CRT screen.

SCROLL mode causes output to stop every 20 lines until the user types any character, exactly as if a ^S had been typed by the user. This is useful in preventing information from scrolling off a CRT screen before it can be examined.

XTABS mode causes tab characters to be expanded to the appropriate number of spaces on output, and is used for terminals that do not handle tabs themselves. Tabs are set at every eight character positions.

In RAW mode, the requested number of characters are buffered and then passed to the reading program. (Note that this differs from some other versions of UNIX in which only a single character is returned, regardless of how many are asked for.) No erase or kill processing is done. There is no special treatment of any character; characters are a full 8-bits for both input and output. This is the only mode in which XON/XOFF handling (^S/^Q) can not be done to control output, although by setting the TANDEM mode XON/XOFF can be used to control input. In RAW mode, the modes CRT, SCROLL, XTABS, CRMOD, LCASE and CBREAK are all ignored.

EVENP and ODDP set even and odd parities, respectively, causing parity bits to be sent on output and checked on input. This parity sending and checking is hardware dependent, and not supported by all interfaces (check the device driver). The action if both EVENP and ODDP are set is also hardware dependent.

Mode CRMOD causes input carriage returns to be turned to new-lines; input of either CR or LF causes LF-CR both to be echoed.

ECHO mode causes characters to be re-echoed to the terminal as typed.

CBREAK is a sort of half-cooked (rare?) mode. Programs can read each character as soon as typed, instead of waiting for a full line, but quit and interrupt,

case translation, CRMOD, XTABS, and ECHO work normally. There is no erase or kill.

TANDEM mode causes the system to produce a stop character (^S) whenever the input queue is in danger of overflowing, and a start character (^Q) when the input queue has drained sufficiently. It is useful for flow control when the 'terminal' is actually another machine that obeys the conventions. Note that start/stop (^S/^Q) is handled on the output queue anyway, except in RAW mode.

The C language expressions used to turn on (OR in) these bit-encoded modes are usually something like

```
mode |= CRT;
```

to turn on CRT mode, or

```
mode |= (CRT | XTABS);
```

to turn on CRT and XTABS. The expressions used to mask off these modes are, to parallel the above examples:

```
mode &= ~CRT;
```

or

```
mode &= ~(CRT | XTABS);
```

The previous **getcode** example has at least one deficiency; if the user types a CTRL-C (interrupt) while the terminal is in no-echo mode, the program will exit with the terminal left this way. Since this behavior is often undesirable, programs which change terminal modes usually arrange to catch interrupts and other signals in a routine which resets the terminal mode before exiting. See the section on signals later in this document for directions on how to do this.

The flags byte is reset when a line is first opened by a process; the reset value is driver-dependent (see the driver source for details). The initial setting for active login lines may be further modified by the login process, depending on the terminal type indicated in **/etc/ttys**. Since active login lines are held open by the login process or shell, changes made to the flags by a program doing I/O

control calls will remain in effect after the program exits. In this case, the program should probably save the original mode and reset it before exiting, as the previous **getcode** example does.

However, if the program uses I/O control calls and changes the setting on lines which are not held open by a login or other process, for example a line to a printer, then the changes made will only last for the duration of the program. In this case, the mode will be reset whenever the next program opens that line.

All the flag modes may be used in conjunction with standard I/O routines like **printf**, **scanf**, **getchar** and so on. RAW mode, however, causes input to be buffered in large chunks, so it should not be used for interactive programs.

Other possible commands for terminals, in addition to the previously mentioned **TIOCSETP** and **TIOCGETP**, are described below. The definitions for all these constants may be found in the include file <**sgtty.h**> (i.e., **/usr/include/sgtty.h**).

TIOCSETN Set the parameters but do not delay or flush input. When the usual TIOCSETP is used, the interface delays until output is quiescent, then throws any unread characters away before parameters are changed. TIOCSENT avoids this delaying and flushing of characters.

TIOCEXCL Set "exclusive-use" mode: no other process can open the file until it is closed by the invoking process, or until exclusive mode is turned off (see below). (This is used, for example, by the line-printer spooler **lpr**(1) to prevent people from interfering with output it is sending to the printer.)

A process "A" can set a terminal to exclusive mode even if the line was previously opened by some other process "B" (non-exclusively). Either process can **read** or **write** to the terminal. In this case, if process "A" exits without explicitly turning off "exclusive-use", process "B" inherits the exclusivity, and no further opening of the line is possible until process "B" exits or turns off exclusivity itself. Since this behavior is not usually desirable, it is

usually a good idea for any process which does a **TIOCEXCL** to do a **TIOCNXCL** on the line before exiting.

TIOCNXCL Turn off "exclusive-use" mode.

TIOCHPCL When the file is closed for the last time, hang up the terminal. This is useful when the line is associated with an ACU used to place outgoing calls.

TIOCFLUSH All characters waiting on input or output are flushed.

TIOCQCNT Returns the count of characters currently typed in but not yet read in **sg_ispeed** (0 to 255) and the count of characters on the output queue, which is probably rapidly changing, in **sg_ospeed** (0 to about 100). This mode is used, for example, by some screen editors, which can check to see if further commands have been typed while they process an earlier one.

2.4.6 Error Processing

The routines discussed in this section, and in fact all the routines which are direct entries into the system can incur errors. Usually they indicate an error by returning a value of **−1**. Sometimes it is nice to know what sort of error occurred; for this purpose all these routines, when appropriate, leave an error number in the external cell **errno**. The meanings of the various error numbers are listed in the introduction to Section 2 of the *Programmer Reference Manual*. Your program can, for example, determine if an attempt to open a file failed because it did not exist or because the user lacked permission to read it. Perhaps more commonly, you may want to print out the reason for failure.
The routine **perror** will print a message associated with the value of **errno**; more generally, **sys_errno** is an array of character strings which can be indexed by **errno** and printed by your program.

Severe programming errors, such as attempts to access unallocated memory or execution of illegal instructions, result in a "signal" being generated. This will normally cause your program to terminate and a file called "core" to be created

containing the memory image of the program at the time the error occurred. The shell detects that the program died in such a way, and will supply an error message such as "memory fault — core dumped."

The debugger **adb** may be employed at this point to examine the "core" file and determine the routine and instruction which caused the error. This is known as "post-mortem" debugging. (See **adb**(1) and **A TUTORIAL INTRODUCTION TO ADB**.) The most frequent cause of such a problem is a pointer error, leading to illegal memory access errors.

As discussed in section 6 of this chapter, signals may be "trapped" (intercepted) by a program before they cause program termination. However, there is usually nothing very useful one can do with illegal instruction or memory access errors except to let them cause termination and a core dump, so that the problem may be found.

2.5 PROCESSES

Using a program written by someone else is often easier than inventing one's own. This section describes how to execute a program from within another.

2.5.1 The System Function

The easiest way to execute a program from another is to use the standard library routine **system**. **system** takes one argument, a command string exactly as typed at the terminal (except for the newline at the end), and executes it. For instance, to time-stamp the output of a program,

```
main()
{
        system("date");
        /* rest of processing */
}
```

If the command string has to be built from pieces, the in-memory formatting capabilities of **sprintf** may be useful.

2.5.2 Low-Level Process Creation — Execl and Execv

If you're not using the standard library, or if you need finer control over what happens, you will have to construct calls to other programs using the more primitive routines that the standard library's **system** routine is based on.

The most basic operation is to execute another program *without returning*, by using the routine **execl**. To print the date as the last action of a running program, use

```
execl("/bin/date", "date", NULL);
```

The first argument to **execl** is the file name of the command; so you have to know where it is found in the file system. The second argument is conventionally the program name (that is, the last component of the file name), but this is seldom used except as a place-holder. If the command takes arguments, they are strung out after this. The end of the list is marked by a **NULL** argument.

The **execl** call overlays the existing program with the new one, runs that, then exits. There is no return to the original program.

More realistically, a program might fall into two or more phases that communicate only through temporary files. Here it is natural to make the second pass simply an **execl** call from the first.

The one exception to the rule that the original program never gets control back occurs when there is an error, for example if the file can't be found or is not executable. If you don't know where **date** is located, say

```
execl("/bin/date", "date", NULL);
execl("/usr/bin/date", "date", NULL);
fprintf(stderr, "Someone stole 'date'\n");
```

A variant of **execl** called **execv** is useful when you don't know in advance how many arguments there are going to be. The call is

```
execv(filename, argp);
```

where **argp** is an array of pointers to the arguments; the last pointer in the array must be **NULL** so **execv** can tell where the list ends. As with **execl**, **filename** is the file in which the program is found, and **argp[0]** is the name of the program. (This arrangement is identical to the **argv** array for program arguments.)

Neither of these routines provides the niceties of normal command execution. There is no automatic search of multiple directories — you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like <, >, *, ?, and [] in the argument list. If you want these, use **execl** to invoke the shell **sh**, which then does all the work. Construct a string "commandline" that contains the complete command as it would have been typed at the terminal, then type

execl("/bin/sh", "sh", "−c", commandline, NULL);

The shell is assumed to be at a fixed place, **/bin/sh**. Its argument **−c** says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in commandline.

2.5.3 The Environment

As seen in the last section, when a program is **exec**'ed, a set of commandline arguments (the program name and appropriate flags and arguments) is passed to it. In addition to these arguments, however, a collection of strings known as the "environment" is automatically passed as well. These can contain any information desired; by convention, they are often of the form

"NAME = val"

where "NAME" is an arbitrarily-named environment variable and "val" is its value.

The shell allows users to set up their environment. For example, some full-screen editor programs need to know what kind of terminal is being used (so that they can decide which codes to send to position the cursor and manipulate the screen). Often the variable "TERM" is used for this purpose. The shell commands

```
TERM = VT52
export TERM
```

(often placed in a user's ".profile" file) cause the string "TERM = VT52" to be placed in the user's environment, and passed as part of the environment to all commands the user runs.

A program can find its environment through the external variable "environ", which should be declared as

```
extern char **environ;
```

Like **argv**, **environ** is a pointer to an array of strings. The array is null-terminated. The following program prints out its environment values:

```
#define NULL ((char *) 0)

extern char **environ;
main(argc,argv)
int argc;
char *argv;
{
        int i;

        for (i = 0; environ[i] != NULL; ++i)
                printf("%s\n",environ[i]);
}
```

The routine **getenv**(3) may be used to find the value-part of a particular environment string of the form "NAME = val".

The **execl** and **execv** calls cause the standard **environ** to be passed to the executed program. If a program needs to change the environment before calling another program, a new array of strings should be allocated. The **execle** and **execve** calls are exactly the same as **execl** and **execv**, respectively, except that they allow the program to specify an environment pointer other than **environ**.

2.5.4 Control of Processes — Fork and Wait

The information covered so far isn't really all that useful by itself. So now you will learn how to regain control after running a program with **execl** or **execv**. Since these routines simply overlay the new program on the old one, saving the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a routine called **fork**;

```
proc_id = fork();
```

splits the program into two copies, both of which continue to run. The only difference between the two is the value of **proc_id**, the "process id." In one of these processes (the "child"), **proc_id** is zero. In the other (the "parent"), **proc_id** is non-zero; it is the process number of the child. Thus the basic way to call, and return from, another program is

```
if (fork() == 0)
    execl("/bin/sh", "sh", "-c", cmd, NULL);    /* in child */
```

And in fact, except for handling errors, this is sufficient. The **fork** makes two copies of the program. In the child, the value returned by **fork** is zero, so it calls **execl** which does the **command** and then dies. In the parent, **fork** returns non-zero so it skips the **execl**. (If there is any error, **fork** returns **−1**).

More often, the parent wants to wait for the child to terminate before continuing itself. This can be done with the function **wait**:

```
int status;

if (fork() == 0)
            execl(...);
wait(&status);
```

This still doesn't handle any abnormal conditions, such as a failure of the **execl** or **fork**, or the possibility that there might be more than one child running simultaneously. (The **wait** returns the process id of the terminated child, if you want to check it against the value returned by **fork**.) Finally, this fragment doesn't deal with any funny behavior on the part of the child (which is reported in **status**). Still, these three lines are the heart of the standard library's **system** routine, which we'll show in a moment.

The **status** returned by **wait** encodes in its low-order eight bits the system's idea of the child's termination status; it is 0 for normal termination and non-zero to indicate various kinds of problems. The next higher eight bits are taken from the argument of the call to **exit** which caused a normal termination of the child process. It is good coding practice for all programs to return meaningful status.

When a program is called by the shell, the three file descriptors 0, 1, and 2 are set up pointing at the right files, and all other possible file descriptors are available for use. When this program calls another one, correct etiquette suggests making sure the same conditions hold. Neither the **fork** nor the **exec** call affects open files in any way. If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the **execl**. Conversely, if a caller buffers an input stream, the called program will lose any information that has been read by the caller.

2.5.5 Pipes

A pipe is an I/O channel intended for use between two cooperating processes: one process writes into the pipe, while the other reads. The system looks after buffering the data and synchronizing the two processes. Most pipes are created by the shell, as in:

```
ls | pr
```

which connects the standard output of **ls** to the standard input of **pr**. Sometimes, however, it is more convenient for a process to set up its own plumbing; in this section, we will illustrate how the pipe connection is established and used.

The system call **pipe** creates a pipe. Since a pipe is used for both reading and writing, two file descriptors are returned; the actual usage is like this:

```
int             fd[2];

stat = pipe(fd);
if (stat == -1)
            /* there was an error ... */
```

fd is an array of two file descriptors, where **fd[0]** is the read side of the pipe

and **fd[1]** is for writing. These may be used in **read**, **write** and **close** calls just like any other file descriptors.

If a process **reads** a pipe which is empty, it will wait until data arrives; if a process **writes** into a pipe which is too full, it will wait until the pipe empties some. If the **write** side of the pipe is closed, a subsequent **read** will encounter end of file.

To illustrate the use of pipes in a realistic setting, let us **write** a function called **popen(cmd, mode)**, which creates a process **cmd** (just as **system** does), and returns a file descriptor that will either **read** or **write** that process, according to **mode**. That is, the call

```
fout = popen("pr", WRITE);
```

creates a process that executes the **pr** command; subsequent **write** calls using the file descriptor **fout** will send their data to that process through the pipe.

popen first creates the the pipe with a **pipe** system call; it then **fork**s to create two copies of itself. The child decides whether it is supposed to **read** or **write**, closes the other side of the pipe, then calls the shell (via **execl**) to run the desired process. The parent likewise closes the end of the pipe it does not use. These closes are necessary to make end-of-file tests work properly. For example, if a child that intends to read fails to close the write end of the pipe, it will never see the end of the pipe file, just because there is one writer potentially active.

```
#include <stdio.h>

#define   READ              0
#define   WRITE             1
#define   tst(a, b)   (mode == READ ? (b) : (a))
static              int       popen_pid;
```

```
popen(cmd, mode)
char        *cmd;
int         mode;
{
        int p[2];

        if (pipe(p) < 0)
                return(NULL);
        if ((popen_pid = fork()) == 0) {
                close(tst(p[WRITE], p[READ]));
                close(tst(0, 1));
                dup(tst(p[READ], p[WRITE]));
                close(tst(p[READ], p[WRITE]));
                execl("/bin/sh", "sh", "-c", cmd, 0);
                _exit(1);       /* disaster has occurred if we get here */
        }
        if (popen_pid == -1)
                return(NULL);
        close(tst(p[READ], p[WRITE]));
        return(tst(p[WRITE], p[READ]));
}
```

The sequence of **close**s is a bit tricky. Suppose that the task is to create a child process that will **read** data from the parent. Then the first **close** closes the **write** side of the pipe, leaving the **read** side open. The lines

```
close(tst(0, 1));
dup(tst(p[READ], p[WRITE]));
```

are the conventional way to associate the pipe descriptor with the standard input of the child. The **close** closes file descriptor 0, that is, the standard input. **dup** is a system call that returns a duplicate of an already open file descriptor. File descriptors are assigned in increasing order and the first available one is returned, so the effect of the **dup** is to copy the file descriptor for the pipe (read side) to file descriptor 0; thus the **read** side of the pipe becomes the standard input. (Yes, this is a bit tricky, but it's a standard idiom.) Finally, the old **read** side of the pipe is closed.

A similar sequence of operations takes place when the child process is supposed to **write** from the parent instead of **read**ing. You may find it a useful exercise to step through that case.

The job is not quite done, because you still need a function **pclose** to close the pipe created by **popen**. The main reason for using a separate function rather than **close** is that it is desirable to wait for the termination of the child process. First, the return value from **pclose** indicates whether the process succeeded. Equally important when a process creates several children is that only a bounded number of unwaited-for children can exist, even if some of them have terminated; performing the **wait** lays the child to rest. Thus:

```
#include <signal.h>

pclose(fd)              /* close pipe fd */
int fd;
{
        register r, (*hstat)(), (*istat)(), (*qstat)();
        int status;
        extern int popen_pid;
        close(fd);
        istat = signal(SIGINT, SIG_IGN);
        qstat = signal(SIGQUIT, SIG_IGN);
        hstat = signal(SIGHUP, SIG_IGN);
        while ((r = wait(&status)) != popen_pid && r != -1);
        if (r == -1)
           status = -1;
        signal(SIGINT, istat);
        signal(SIGQUIT, qstat);
        signal(SIGHUP, hstat);
        return(status);
}
```

The calls to **signal** make sure that no interrupts, etc., interfere with the waiting process; this is the topic of the next section.

The routine as written has the limitation that only one pipe may be open at once, because of the single shared variable **popen_pid**; it really should be an array indexed by file descriptor. A **popen** function, with slightly different

arguments and return value is available as part of the standard I/O library discussed below. As currently written, it shares the same limitation.

2.6 SIGNALS — INTERRUPTS

This section is concerned with how to deal with signals from the outside world (like interrupts), and with program faults. Since there's nothing very useful that can be done from within C about program faults, which arise mainly from illegal memory references or from execution of peculiar instructions, the following discussion deals only with the outside-world signals: **interrupt**, which is sent when '^C' is typed; **quit**, generated by '^Z'; **hangup**, caused by hanging up the phone; and **terminate**, generated by the **kill** command. When one of these events occurs, the signal is sent to all processes which were started from the corresponding terminal; unless other arrangements have been made, the signal terminates the process. In the **quit** case, a core image file is written for debugging purposes. See **signal**(2) for a complete list of signals.

The routine which alters the default action is called **signal**. It has two arguments: the first specifies the signal, and the second specifies how to treat it. The first argument is just a number code, but the second is the address and is either a function or a somewhat strange code that requests that the signal either be ignored or be given the default action. The include file **signal.h** gives names for the various arguments, and should always be included when signals are used. Thus:

```
#include <signal.h>
...
signal(SIGINT, SIG_IGN);
```

causes interrupts to be ignored, while:

```
signal(SIGINT, SIG_DFL);
```

restores the default action of process termination. In all cases, **signal** returns the previous value of the signal. The second argument to **signal** may instead be the name of a function (which has to be declared explicitly if the compiler hasn't seen it already). In this case, the named routine will be called when the signal occurs. Most commonly this facility is used to allow the program to clean up unfinished business before terminating, for example to delete a

temporary file:

```
#include <signal.h>

main()
{
        int onintr();

        if (signal(SIGINT, SIG_IGN) != SIG_IGN)
                signal(SIGINT, onintr);

        /* Process ... */

        exit(0);
}
onintr()
{
        unlink(tempfile);
        exit(1);
}
```

Why the test and the double call to **signal**? Recall that signals like interrupt are sent to all processes started from a particular terminal. Accordingly, when a program is to be run non-interactively (started by **&**), the shell turns off interrupts for it so it won't be stopped by interrupts intended for foreground processes. If this program began by announcing that all interrupts were to be sent to the **onintr** routine regardless, that would undo the shell's effort to protect it when run in the background.

The solution, shown above, is to test the state of interrupt handling, and to continue to ignore interrupts if they are already being ignored. The code as written depends on the fact that **signal** returns the previous state of a particular signal. If signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command-processing loop. Think of a text editor: interrupting a long printout should not

cause it to terminate and lose the work already done. The outline of the code for this case is probably best written like this:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf         sjbuf;

main()
{
        int (*istat)(), onintr();

        istat = signal(SIGINT, SIG_IGN);  /* save original status */
        setjmp(sjbuf);              /* save current stack position */
        if (istat != SIG_IGN)
                signal(SIGINT, onintr);

        /* main processing loop */
}


onintr()
{
        printf("\\nInterrupt\\n");
        longjmp(sjbuf);             /* return to saved state */
}
```

The include file **setjmp.h** declares the type **jmp_buf** an object in which the state can be saved. **sjbuf** is such an object; it is an array of some sort. The **setjmp** routine then saves the state of things. When an interrupt occurs, a call is forced to the **onintr** routine, which can print a message, set flags, or whatever. **longjmp** takes as argument an object stored into by **setjmp**, and restores control to the location after the call to **setjmp**, so control (and the stack level) will pop back to the place in the main routine where the signal is set up and the main loop entered. Notice, by the way, that the signal gets set again after an interrupt occurs. This is necessary; most signals are automatically reset to their default action when they occur.

Some programs that want to detect signals simply can't be stopped at an arbitrary point, for example in the middle of updating a linked list. If the routine

called on occurrence of a signal sets a flag and then returns instead of calling **exit** or **longjmp**, execution will continue at the exact point it was interrupted. The interrupt flag can then be tested later.

There is one difficulty associated with this approach. Suppose the program is reading the terminal when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that "execution resumes at the exact point it was interrupted," the program would continue reading the terminal until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading; the user presumably would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the terminal read when execution resumes after the signal, returning an error code which indicates what happened.

Thus programs which catch and resume execution after signals, should be prepared for "errors" which are caused by interrupted system calls. (The ones to watch out for are **reads** from a terminal, **wait**, and **pause**.) A program whose **onintr** program just sets **intflag**, resets the interrupt signal, and returns, should usually include code like the following when it reads the standard input:

```
if (getchar() == EOF)
        if (intflag)
                /* EOF caused by interrupt */
        else
                /* true end-of-file */
```

A final subtlety to keep in mind becomes important when signal-catching is combined with execution of other programs. Suppose a program catches interrupts, and also includes a method (like "!" in the editor) whereby other programs can be executed. Then the code should look something like this:

```
if (fork() == 0)
        execl(...);
signal(SIGINT, SIG_IGN);        /* ignore interrupts */
wait(&status);                  /* until the child is done */
signal(SIGINT, onintr);         /* restore interrupts */
```

Why is this? Again, it's not obvious but not really difficult. Suppose the program you call catches its own interrupts. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read your terminal. But the calling program will also pop out of its wait for the subprogram and read your terminal. Having two processes reading your terminal is very unfortunate, since the system figuratively flips a coin to decide who should get each line of input. A simple way out is to have the parent program ignore interrupts until the child is done. This reasoning is reflected in the standard I/O library function system :

```
#include <signal.h>

system(s)       /* run command string s */
char *s;
{
        int status, pid, w;
        register int (*istat)(), (*qstat)();

        if ((pid = fork()) == 0) {
                execl("/bin/sh", "sh", "-c", s, 0);
                _exit(127);
        }
        istat = signal(SIGINT, SIG_IGN);
        qstat = signal(SIGQUIT, SIG_IGN);
        while ((w = wait(&status)) != pid && w != -1)
                ;
        if (w == -1)
                status = -1;
        signal(SIGINT, istat);
        signal(SIGQUIT, qstat);
        return(status);
}
```

As an aside on declarations, the function **signal** obviously has a rather strange second argument. It is in fact a pointer to a function delivering an integer, and this is also the type of the signal routine itself. The two values **SIG_IGN** and **SIG_DFL** have the right type, but are chosen so they coincide with no possible actual functions. For the enthusiast, here is how they are defined for the PDP-

11; the definitions should be sufficiently ugly and nonportable to encourage use of the include file.

```
#define     SIG_DFL     (int (*)())0
#define     SIG_IGN     (int (*)())1
```

2.7 FILES AND DIRECTORIES

2.7.1 File Attributes — Stat Call

VENIX maintains for each file a header block known as an "inode", holding various pieces of information about the file, such as its length and read/write/execute permissions. Using the **stat** system call, the contents of an i-node can be examined at any time. **stat** returns a buffer with the following structure:

```
struct stat
{
        dev_t       st_dev;
        ino_t       st_ino;
        unsigned short  st_mode;
        short       st_nlink;
        short       st_uid;
        short       st_gid;
        dev_t       st_rdev;
        off_t       st_size;
        time_t      st_atime;
        time_t      st_mtime;
        time_t      st_ctime;
};
```

This structure is defined in the include file <**sys/stat.h**>. The type definitions "dev_t", "off_t" and "time_t" are all given in <**sys/types.h**>. (Since these types may vary with different implementations of UNIX, they are given special definitions in order to make the code using them more portable.)

Here is a brief description of each element:

st_dev contains the major and minor numbers of the device on which the file resides. The macro functions **major()** and **minor()** (given in the include file <**sys/types.h**>), take an **st_dev** as argument and extract one or the other of the two numbers. Together, these two numbers uniquely identify each disk partition; their actual meaning is described in **SETTING UP VENIX** in the *Installation and System Manager's Guide*.

st_ino holds the i-node number (or "i-number") of this file. Each file has an i-node number unique to its file system. Since each file system corresponds with exactly one disk partition, **st_ino** in combination with **st_dev** uniquely identifies the file for the entire system.

st_mode holds (bit-encoded) information on the read/write/execute permission of the file, and a few other items. The encoding is outlined in the **#define** statements in the include file <**sys/stat.h**>. Each condition is given as an octal value.

```
#define S_IFMT      0160000   /* type of file */
#define S_IFDIR     0140000   /* directory */
#define S_IFCHR     0120000   /* character special */
#define S_IFBLK     0160000   /* block special */
#define S_IFREG     0100000   /* regular file */
#define S_ILRG      010000    /* large file */
#define S_ISUID     004000    /* set user id on execution */
#define S_ISGID     002000    /* set group id on execution */
#define S_ISVTX     001000    /* save shared segment after use */
#define S_IREAD     000400    /* read permission, owner */
#define S_IWRITE    000200    /* write permission, owner */
#define S_IEXEC     000100    /* execute permission, owner */
```

The read/write/execute permissions are given in the low nine bits of the file. The file owner's own permissions are encoded in **S_IREAD**, **S_IWRITE** and **S_IEXEC** (i.e., those bits masked by 0700). For example, to test if the owner has execute permission, the expression

((mode & S_IXEC) != 0)

is used. The high-order bits masked by **S_IFMT** indicate the type of file: directory, character device, block device or regular file. For example, the expression:

((mode & S_IFMT) == S_IFDIR)

is true if the file is a directory.

The permissions for the owner's group are given in the bits masked by 070; those for "others" are given in 07. Other bits indicate that the file is "set-UID", "set-GID", and "sticky". The meaning of these conditions is described elsewhere.

st_nlink is the number of links to the file, that is, the number of names by which the file is known. When a file is first created, the link count is one. It is possible to create further links to the file with the **link** system call or the **ln** command. Having multiple links to a file is convenient if different users or programs prefer to call one file by different names.

When a file is removed, its name is said to be "unlinked" (the system call used is **unlink()**). Unlinking a file name breaks the connection between that name and the file; only the name is removed from the file system. The file's link count is decremented by one. The file itself, however, remains around, accessible by all remaining links, until the last link is undone. At that point, the file disappears and its disk space deallocated.

st_uid holds the owner's user ID number, corresponding to the user's name in the **/etc/passwd** file.

st_gid holds the owner's group ID number, corresponding to the user's group name in the **/etc/group** file. Although the user and group ID are given here as type short integer, they are internally stored as char's, and thus are limited to values 0-255.

st_rdev holds major and minor device numbers for files that are block or character devices. These numbers are of the same type that **st_dev** is, but indicate the device these special files "point" to, not the device they reside on.

st_size is the size of the file, measured in bytes.

st_atime is the time the file was last accessed.

st_mtime is the time the file was last modified. Time is measured in seconds since 0:00 January 1, 1970.

st_ctime is identical to **st_atime** (In some UNIX implementations, it holds the time the file was created, but this is not currently done in VENIX.)

With all this in mind, you can write a crude version of the **ls** command. This program, which may be called "list", gives a status report on all files passed it on the command line. The program is written with a **main** routine which processes each command argument, does a **stat** of it, and passes the information to a routine called **list** which prints out the information in readable form. **list** uses the standard library routines **ctime**(3) to convert the time into English, and **getpwuid**(3) to map the owner's user ID number to his name in the **/etc/passwd** file.

```
/* list - list status information on files */

#include <stdio.h>          /* standard I/O */
#include <pwd.h>            /* password header */
#include <sys/types.h>      /* system type defs */
#include <sys/stat.h>       /* stat structure */

char def_perm[] = "rwxrwxrwx";      /* full permission setting */
char perm[9];           /* permission for each file */
struct stat sbuf;       /* stat buf */
```

```
main(argc,argv)
int argc;
char *argv[];
{
        if (argc == 1){                 /* no arguments given */
                fprintf(stderr,"Usage: list file ...\n");
                exit(1);
        }
        while (--argc){
                ++argv;
                if (stat(*argv,&sbuf) < 0){
                        fprintf(stderr,"can't stat %s\n",*argv);
                        continue;
                }
                list(*argv,&sbuf);
        }
        exit(0);
}

/* list routine - print info on pointed-to structure */

list(name,psb)
char *name;                             /* file name */
struct stat *psb;                       /* pointer to stat buffer */
{
      int i;
      int bit = 0400;                   /* high mode bit */
      int lcount;                       /* link count */
      long size;                        /* file size */
      char type;                        /* type of file */
      char *time, *ctime();                     /* file time */
      char *uname;                      /* user name */
      struct passwd *ppwd, *getpwuid();
```

```
        for (i = 0; i < 9; ++i){
            if ((psb->st_mode & bit) == 0)
                    perm[i] = '-';
                else      perm[i] = def_perm[i];
                bit >>= 1;              /* rotate bit to the right */
        }
        switch(psb->st_mode & S_IFMT){
                case S_IFDIR:
                        type = 'd';     /* directory */
                        break;
                case S_IFCHR:
                        type = 'c';     /* character device */
                        break;
                case S_IFBLK:
                        type = 'b';     /* block device */
                        break;
                case S_IFREG:
                        type = '-';     /* plain file */
                        break;
        }
        time = ctime(&(psb->st_mtime));
        ppwd = getpwuid(psb->st_uid);
        uname = ppwd->pw_name;

        lcount = psb->st_nlink;
        size = psb->st_size;

        printf("%c%s %-3d%-7s%-7D%.24s %-14s\n",
            type,perm,lcount,uname,size,time,name);
}
```

2.7.2 Directory Structure

A directory is a special type of file which contains a list of file names and corresponding inode-numbers. Directories may only be written into by VENIX itself; this is done of course whenever a new file is created or removed from the directory. However, it is entirely possible for a user program to read a directory, and the contents can be easily interpreted.

The format of a directory entry is very simple and is defined in the include file **<sys/dir.h>**:

```
#define DIRSIZ 14

struct      direct {        /* structure of directory entry */
        ino_t       d_ino;          /* inode number */
        char        d_name[DIRSIZ];         /* file name */
};
```

The type definition "ino_t" is defined in **<sys/types.h>**; on PDP-11's, it is an integer. **d_ino** is the inode number, and **d_name** is the file name. The size of each file entry then is 2 bytes (inode number) + 14 bytes (name) = 16 bytes total. However, to keep your programs portable, it is always best to refer to the length of each entry as

```
sizeof(struct direct)
```

which evaluates to the right number.

When a file is created, it is assigned a new entry in its directory. When it is removed, the i-node number is zeroed, though the file name itself may remain. Therefore when reading a directory a program must always check that a file has a non-zero i-number and thus really exists. Future entries will overwrite removed ones, but no attempt is ever made to condense a directory to recover removed entries; directories grow as needed, but never shrink.

With this in mind, you can improve the listing program given earlier. In the earlier version, the specifications of each file argument were listed; if a directory name was given as argument, the directory itself was listed, not the files it contained. Since it is frequently convenient to have the contents of each directory argument listed, you can modify the list program to check if each argument is a directory, and if so, to extract and list all its entries, that is, all the files the directory contains.

Since the program was written in a modular fashion, only the **main** need be modified. The **list** routine remains the same, and is not duplicated here.

```
/* list program #2  -  examines contents of directory */
#include <stdio.h>                  /* standard I/O */
#include <pwd.h>                    /* password header */
#include <sys/types.h>              /* system type defs */
#include <sys/dir.h>                /* directory structure */
#include <sys/stat.h>               /* stat structure */
char def_perm[] = "rwxrwxrwx";      /* full permission setting */
char perm[9];                       /* permission for each file */
char name[200];
struct direct dbuf;
struct stat sbuf;                   /* stat buf */

main(argc,argv)
int argc;
char *argv[];
{
        register char *p1, *p2, *q1;
        int i, fd;

        if (argc == 1){
                fprintf(stderr,"Usage: list file ...\n");
                exit(1);
        }
        while (--argc){
           ++argv;
           if (stat(*argv,&sbuf) < 0){
                   fprintf(stderr,"can't stat %s\n",*argv);
                   continue;
           }
```

```
            if ((sbuf.st_mode & S_IFMT) != S_IFDIR)
                list(*argv,&sbuf);          /* normal file */
            else {          /* list directory */
                printf("\ndirectory %s:\n",*argv);
                p1 = name;
                q1 = *argv;
                while (*p1++ = *q1++) /* pocket dir name */
                    ;
                *(p1-1) = '/';          /* cover null with / */
                if ((fd = open(*argv,0)) < 0){
                    fprintf(stderr,"can't open directory %s\n",*argv);
                    exit(1);
                }
                while ((i = read(fd,&dbuf,sizeof(struct direct))) != 0){
                    if (i < 0){
                        fprintf(stderr,"error reading %s\n",*argv);
                        exit(1);
                    }
                    if (dbuf.d_ino != 0){   /* is a real entry? */
                        p2 = p1;            /* dirname */
                        q1 = dbuf.d_name;
                        while (*p2++ = *q1++) /* full name */
                            ;
                    if (stat(name,&sbuf) < 0){
                        fprintf(stderr,"can't stat %s\n",name);
                        exit(1);
                    }
                    list(dbuf.d_name,&sbuf);
                    }
                }
                close(fd);          /* close directory */
                printf("\n");
                }
        }
        exit(0);
    }
```

If an argument is determined to be a directory, it is opened and read. Each entry with a non-zero inode number is extracted, and appended to the directory

name itself so that a **stat** can be done on it. The **list** routine is then called to print out the inode information.

To repeat, this is quite a simple list program. It doesn't attempt to sort its entries, and when listing a directory's contents it doesn't suppress listing of the mandatory "." and ".." entries (referring to the directory itself and its parent).

2.8 SHARED DATA SEGMENTS AND SEMAPHORES

2.8.1 Shared Data Segments

VENIX provides a facility for one or more processes to hook up to a common data segment. This is done through the **sdata** system call, which windows an 8k byte segment of the process' memory into the segment. The **sdata** call is of the form

```
sdata(arg, reg, offset)
char *arg;
int reg, offset;
```

There are two types of shared data segments: "named" segments, which are used when several processes need to hook into the same segment, and "unnamed" segments, for a single process which uses the segment as an extra memory buffer only.

Named segments are opened by calling **sdata** with a file name as **arg**. When the first process hooks up to a named segment, that file is brought in from disk and placed in memory. Subsequent **sdata** calls by other processes will hook up to that file in memory. When a named segment is opened, the **offset** argument indicates the offset in 64 byte units that the file should be windowed into memory. Further **sdata** calls can change this offset.

Unnamed segments are opened by calling **sdata** with **arg** = 1 (more properly, the 1 should be coerced into a character pointer as in **(char *) 1**). Unnamed segments are useful for accessing extra memory, but are unique to each process that opens them; there is no sharing involved (the name "shared data segment" is in this case a misnomer). When an unnamed segment is opened, **offset** specifies the length, in 64 byte units.

The following rules apply to both named and unnamed segments: On opening, **reg** is the number of the Active Page Register (APR) to be used for the window. Each process has 8 APR's (numbered 0 – 7), each of which maps an 8kb section of the process' logical address space into physical memory. The top page of logical memory (APR 7) is reserved for the stack, and the lower pages of memory (APR 0, 1,...) are taken up by the code and data portions of the program;† the number of registers actually used depends on the size of the program. APR 6, however, is usually free for normal-sized programs, and can be used for the shared data segment. This corresponds to virtual address 6 x 8k = 6 x 020000 (octal) = address 0140000 (octal).

The offset, into the shared segment where the window is placed, is set when named segments are opened. The offset may be changed, for both kinds of segments, by additional **sdata** calls with **arg** = 0 and **offset** = the offset that the window is placed into the file, in multiples of 64 bytes. So while only 8kb of the segment is viewable at any time, the particular 8kb piece chosen may begin at any 64 byte boundary in the segment. It is forbidden to move the window base beyond the end of the segment, although it is possible to move the window base up to the end. (If the base is less than 8kb from the end, attempts to address memory logically beyond the segment will cause memory violation errors leading to a core dump unless otherwise trapped.)

The maximum size of named or unnamed segments is installation-dependent, and can be changed by the system administrator. For more information on this refer to the chapter, **SETTING UP VENIX**, in the *Installation and System Manager's Guide*.

2.8.2 Semaphores

Frequently, when one process is reading or writing to a shared segment it has to be careful that another process, looking at the same segment, is not tampering with it; a jumbled, corrupted record might otherwise result. To prevent this, VENIX allows the use of binary semaphores as a means of communicating

---

† On separate I&D machines, programs linked for separate I&D use one set of APR registers dedicated to data space only, with code space controlled by a different set of registers.

between cooperating processes, and this mechanism is especially useful for the problems of managing shared data segments. While reading or writing a particular shared segment, a process first checks an agreed upon semaphore to see if somebody else is playing with it. If nobody is, the process can set the semaphore itself and begin writing. Now, other processes that plan to read or write there will see that the semaphore is set, and will wait. When the first process is finished writing, it clears the semaphore, and another process waiting for it can use it.

The main system calls to set and clear semaphores are:

semset(semnum, priority)

and,

semclear(semnum)

semset sets semaphore number **semnum** if it was clear; otherwise, it causes the calling process to go to sleep. When the semaphore finally clears, the process waiting for this semaphore with the highest priority (lowest number) wakes up; the others remain sleeping. **semclear** clears semaphore **semnum**, and causes the highest priority process waiting for it to wake up. Other calls allow various testing of semaphores; see **semset**(2).

There are 32 possible semaphores, with values −16 to 15. Semaphores with values −16 to −1 are "global" and maintained across the entire system; those with values 0 to 15 are "local" maintained by each process group (that is, the group of processes sharing the same control terminal). Semaphores are not reset when a program starts or exits; local semaphores will be cleared for a new process group, that is, when a user logs in.

The following program sets up a simple shared-data mailing list. The header file is as follows:

```
/* mail.h -- structure definition */

#define     NCHAR        30   /* Number of characters per line */
#define     NNAMES       400  /* Number of slots for names */
#define MLIST     "mailing.list"  /* Name of mailing list file */

struct        mail {
        int         flags;        /* Place for flags, etc. */
        char        line1[NCHAR];       /* Address ... */
        char        line2[NCHAR];
        char        line3[NCHAR];
};

/*
 * Flag bits
 */
#define   F_ALLOC   01 /* Slot is allocated */
#define   F_NUSA    02 /* Address is not in USA */

/*
 * Some implementation dependent parameters
 */
#define   SEM     -1   /* Use global semaphore no. 1 */
#define   REG     6    /* Use PDP-11 mapping register 6 */
#define   SIZE    020000   /* Addressing size of a register */
/*
 * Following are needed because PDP-11's memory management
 * allows relocation only in increments of 64 bytes.
 */
#define     GRAN     077     /* Addressing granularity */
#define     SHIFT    6     /* Shift for granularity */
```

The first program creates a prototype data file, **NNAMES** records long, with one dummy address at the beginning.

```
#include "mail.h"

struct      mail      data[NNAMES];

main(){     /* set up the mailing list data prototype file */
        int       fd;

        if( (fd = creat(MLIST,0666)) < 0 ){
                perror(MLIST);
                exit(1);
        }
        /*
         * Fill in one name
         */
        data[0].flags = F_ALLOC;
        strncpy( &data[0].line1, "John Q. Public", NCHAR);
        strncpy( &data[0].line2, "100 Main Street", NCHAR);
        strncpy( &data[0].line3, "Anytown MA 02000", NCHAR);
        if( write(fd, &data, sizeof data) != sizeof data ){
                perror(MLIST);
                exit(2);
        }

        semclear(SEM);          /* initialize semaphore */
}
```

The **strncpy** function copies the address lines into the data structure, and pads the end of it with nulls (see **string**(3)).

Next you have the routine **getmslot** which reads this file in and finds a empty entry in it.

```
#include "mail.h"

struct mail *getmslot(){        /* find empty slot in mailing list */
    register struct mail *mp;
    register int cnt;
    static int sd_set = 0;   /* flag set when shared data mapped */
```

```
        if( sd__set == 0 ){
            sd__set++;
            if( sdata(MLIST,REG,0) < 0 ){
                perror(MLIST);
                geturn(-1);
            }
        }

        /*
        * Simple minded linear search algorithm.
        */
        cnt = 0;
        for( ;; ){
            /*
            * Map the desired portion.
            */
            sdata( 0, REG, (cnt*sizeof(struct mail))>>SHIFT );
            /*
            * Check for empty slots here.
            */
            mp = REG*SIZE + ((cnt*sizeof(struct mail)&GRAN);
            for( ; mp<((REG+1)*SIZE)-sizeof(struct mail);++mp){
                if( (mp->flags&F__ALLOC)==0 )
                    /*
                    * Found one. Give caller its address.
                    */
                    return(mp);
                if(++cnt>=NNAMES )
                    /*
                    * No empty slots, return NULL.
                    */
                    return(0);
            }
        }
    }
```

The flag **sd__set** indicates whether the **sdata** segment has been read in yet by the process. If it hasn't, an initial **sdata** call is made to hook up to it. Note that after an **sdata** call has been made to a given file, future **sdata** calls passing a

null pointer as a file name will automatically hook up to this file, and will be more efficient than explicitly specifying the file name each time.

getmslot then proceeds to run through the record structure in the file, looking for an empty one (i.e. with **F_ALLOC** flag clear). It scans through the length of the window, address **REG * SIZE** to **(REG + 1) * SIZE**, looking for a free record.

When **mp** points to a structure which crosses the boundary from one window to another, the innermost **for** loop breaks and another **sdata** call is made to bring the window up to a boundary as close as possible below the next structure. The program is unfortunately complicated by the fact that the window must be placed on a 64 byte boundary. The **sdata** in the outer **for** loop moves the window onto the correct boundary; before the inner **for** loop begins, the pointer is set to the bottom of the window (**REG * SIZE**) plus the offset (0 – 63 bytes) necessary to address the beginning of the next structure.

Finally, there is the program **addname** which adds a name to the list.

```
#include "mail.h"

main(){         /* add a name to the mailing list */
        register struct mail *mp;
        char buf1[NCHAR];
        char buf2[NCHAR];
        char buf3[NCHAR];
        printf("Input name: ");
        scanf("%30s",buf1);
        printf("Input street address: ");
        scanf("%30s",buf2);
        printf("Input city, state and ZIP: ");
        scanf("%30s",buf3);

        semset(SEM, 0);         /* lock out other users */
```

```
        if( (mp = getmslot()) == 0 ){
                printf("No empty slots.\n");
                semclear(SEM);          /* clear the semaphore */
                exit(1);
        }
        mp->flags = F_ALLOC;

        strncpy(&mp->line1, &buf1, NCHAR); /* copy in address */
        strncpy(&mp->line2, &buf2, NCHAR);
        strncpy(&mp->line3, &buf3, NCHAR);

        semclear(SEM);          /* clear the semaphore */

        exit(0);
}
```

The program asks the user for the name and address. It then makes a **semset** call to either guarantee that nobody else is playing with the file, or to wait until the file is free. The wake-up priority is arbitrarily set to zero. **getmslot** is called to find a free slot, and the data is copied in there. Finally, the semaphore is cleared and the program finishes.

The beauty of this arrangement is that any number of people can run **addname** simultaneously without interfering with each other. The semaphores guarantee that only one version of the program will update the mailing list at a time.

2.9 REAL-TIME PROGRAMMING

VENIX contains a number of features which are useful for applications requiring high data throughput, or quick response to "real-time" events. Some of these features, such as "exclusive priority" execution, require privileged use of the computer. These features are accessible by the super-user ("root" login, user-ID = zero). To allow access to these features without the more general super-user privileges (such as unrestricted file manipulation), users may be placed in the "super-group", (group-ID = zero). Members of the super-group have only the privileged abilities to run at exclusive priority, lock their processes in memory, and directly access the I/O page. (It must be realized that unrestricted access to the I/O page can lead to total control of the computer; members of the super-group are restricted only at the file-system level.)

No attempt is made here to discuss the use of specific data acquisition devices, such as A/D's or D/A's. These devices can usually be manipulated by **ioctl** (I/O control) calls to control the clock speed, set the acquisition mode, or do anything else under driver control. The use of **ioctl** calls for terminals is described in a previous section; similar calls may be used for many data acquisition devices, although the data structures passed are different for each device. See the device driver write-up in **DEVICES**, section (7) of the *Installation and System Manager's Guide* for particulars.

After an **ioctl** call is done to set these parameters, standard **read** and **write** calls can be made. Devices which use direct memory accessing (DMA) can take advantage of VENIX's asynchronous I/O capabilities described below.

2.9.1 Raw Disk and Tape I/O

Raw I/O can be used when a user wishes to transfer data directly and quickly between his buffer and any mass storage device (disk or tape) which can do direct memory access (DMA). This is possible for most disk and tape devices. Raw I/O is different from ordinary (buffered) I/O, in which the system internally buffers everything between the user and the device. The main advantage of raw I/O is its speed; its disadvantage is that the user must transfer data in the same size blocks that the device physically uses, which is normally 512 bytes at a time. All **read**, **write**, and **lseek** calls must be made in multiples of this block size for raw I/O; in all other ways, they can be used exactly as previously described.

On disks, raw I/O bypasses the file system hierarchy. For this reason, it can not be used with an ordinary file, but instead with an entire disk partition. This disk partition can not have a file system on it; it must be dedicated to raw data. (On tape devices this is irrelevant, since file systems are not used.)

A raw device is opened for I/O just as any ordinary file is, although in this case the "file" is an entire area of the disk or a tape unit. Raw device names are generally the ordinary device names prefixed by an "r"; that is, the buffered device "/dev/rl0.usr" becomes the raw device "/dev/rrl0.usr," "/dev/mt0" becomes "/dev/rmt0," and so on. The devices "rl0.usr" and "rrl0.usr" cover exactly the same disk partition: the only difference is that the first implies buffered I/O, and the second, raw I/O.

Once a raw device is opened, all I/O to it is automatically handled as raw. No special requirements are made, beyond the 512 byte block transfer rules given above.

If a disk partition is too large to be dedicated entirely to raw I/O, or too small to hold the needed data, the system administrator can adjust its size. This is described in **SETTING UP VENIX** in the *Installation and System Manager's Guide*.

2.9.2 Asynchronous I/O

Input and output (raw or ordinary) is normally synchronous; that is, after a call to **read** or **write** returns, the user is guaranteed that the buffer transfer has been completed. If VENIX is unable to complete the transfer immediately, then it waits until resources are available, e.g. the system gets the desired disk block into memory. This is very convenient from the programmer's point of view, and generally only a small and quite acceptable delay is produced.

However, occasionally the I/O may take a significant time to complete, as with an A/D, D/A, or some remote device such as an array processor. Other times, overlapping transfers may be needed to maintain a high throughput, as when taking data from an A/D to the user's buffer, and then to a disk. In these cases, "asynchronous I/O" is required.

The asynchronous I/O in VENIX allows a user process to overlap CPU processing and any I/O which uses DMA to transfer data between the device and the user's buffer. Disk, tape and many A/D and D/A devices are in this class. The penalty for this added flexibility is extra responsibility on the part of the user for multiple buffering and checking to see that an asynchronous transfer is completed. The reward is higher throughput.

Asynchronous I/O is always raw; in fact, it is more accurately referred to as "asynchronous raw I/O". Since it is raw, its use is restricted to entire disk partitions. Again, reading and writing must be in block multiples (usually 512 bytes).

The fact that I/O is to be handled asynchronously is again given in the **open** call, where the asynchronous version of the device is given. Asynchronous device names have an "a" prefixed to them, so the asynchronous version of "/dev/rl0.usr" is "/dev/arl0.usr", and the asynchronous version of "/dev/ad" is "/dev/aad".

Checking of multiple buffers is handled by the special system call **aiowait**(2), which is of the form:

aiowait(fd,level)

where **fd** is the file descriptor returned by the **open** call of the device, and **level** is the number of outstanding I/O requests (i.e., unfilled buffers). **aiowait** causes the calling process to sleep until the number of outstanding I/O requests falls less than or equal to **level**. **aiowait** returns with the number of outstanding requests left. If **level** is negative, **aiowait** just returns with the number of outstanding requests, without any sleeping.

The following is an example of a simple program to read from an A/D (with DMA) and write to an area on disk which has been reserved for raw I/O (i.e. no file system is on the partition). Note that ordinary **read** and **write** calls are used; the fact that the I/O will be handled asynchronously is specified only when the **open** call is made, at which time the asynchronous device name is used. (Here the disk partition used has been given the name "arl0.data"; the ".data" extension is a reminder that this partition is used for data-taking only.)

```
int buf1[256], buf2[256];

main(){     /* simple double buffer program */
        int afd, dfd, i;

        afd = open("/dev/aad",0); /* async A/D */
        dfd = open("/dev/arl0.data",1);   /* async disk partition */

        read(afd, buf1, sizeof(buf1));   /* start the first read */
```

```
        for (i = 0; i < 1500; i+=2){        /* read 1500 blocks */
            read(afd, buf2, sizeof(buf2));
            aiowait(afd,1);
            write(dfd, buf1, sizeof(buf1));
            read(afd,buf1, sizeof(buf1));
            aiowait(afd,1);
            write(dfd, buf2, sizeof(buf2));
        }
}
```

The **aiowait(afd, 1)** after the second buffer transfer has been started causes the user's process to wait until no more than one A/D request remains, thus indicating that the first buffer has been filled. The program continuously fills one of the two buffers asynchronously, while it writes out the other (filled) buffer to the disk.

This program assumes that the disk will always finish the write before the A/D is ready to start the next read. An **aiowait(dfd, 0)** could be done to guarantee the write is really finished (in which case it is a synchronous write, and the disk could just as well have been opened for regular I/O), or a triple buffer scheme could be introduced: one buffer for the read, one for the write, and one for the slop when switching back and forth. Triple or sometimes greater buffering is needed when the A/D transfer rate gets close to the average transfer rate of the disk, especially since there is variation in the disk seek times.

The program also assumes that the A/D is always being read by a least one queued request. If the computer is too slow in initiating write requests to the disk, or is caught up in activity for other users, then all requests may be satisfied before a new one is initiated. The A/D will stop being read during that interval, resulting in a loss of data. This condition can be checked by doing an **aiowait(afd, −1)** (which returns the number of queued buffers) and making sure that there is at least one queued at all times. If this is a problem, then your program may need to use more buffers or run at high priority (see below).

It must also be noted that even when buffers are continuously queued, there is a possibility of data loss between buffers. This occurs at data rates above several kilohertz when VENIX itself can not set up the next queued transfer in time to catch new data. The more sophisticated A/D converters have hardware FIFO's or support for double-buffering, which prevent this problem from occurring.

2.9.3 High Priority Execution

VENIX normally apportions equal shares of processor time to each process running. This can be adjusted, however, with the system call

nice(incr)

nice can be called by any process with a positive **incr** to lower the process's execution priority; super-user processes, and those in the super-group, can call **nice** with a negative **incr** to raise their execution priority. Normal priorities range from −20 (highest priority) to 20 (lowest priority), but even at −20 a process still won't get exclusive use of the machine.

For processes which do require exclusive use of CPU time, for example to handle large amounts of data, the special "exclusive" priority can be set by calling **nice** with an increment of −100. Once a process has set itself to this priority, it has total use of CPU time. All normal-priority processes are frozen until the exclusive process exits, lowers its priority, or goes to sleep. Two or more "exclusive" priority processes will compete evenly for CPU time.

2.9.4 Sleeping

Processes will implicitly sleep (that is, suspend execution for an interval) for several reasons: normal (synchronous) read/write calls will cause a process to sleep until the block of data is transferred; **aiowait** calls with a positive **level** will cause the process to sleep until the number of outstanding asynchronous requests reach that level or below; the **semset** call (set semaphore) will cause the process to sleep until the given semaphore is clear. These sleeps are automatically induced by these calls, but it is also possible for a process to explicitly cause a sleep though the **sleep** routine, of the form

sleep(+seconds)

or

sleep(−clockticks)

Sleep takes one argument to indicate the amount of time to suspend execution. If it is positive, the sleep is measured in seconds; if it is negative, it is measured in clock-ticks (60ths of seconds). Sleeping can be very useful for exclusive processes which wish to allow other system activity to take place during non-critical intervals. **Sleep**, incidentally, is composed of two system calls: an **alarm** to schedule an alarm signal in whatever amount of time, and then a **pause** to suspend execution until the alarm. It is possible to schedule alarms without sleeping; see the discussion of signals earlier in this document, and **alarm**(2).

A running process set to exclusive-priority will give up control of the CPU when it sleeps, and regain it when the sleep finishes. There may, however, be some delay — usually small — involved in waking up. If there is much other activity on the system, the process could get swapped out onto disk during its sleep; in this case there will be a delay after the sleep is over while the process is brought into memory. This behavior can be prevented by locking the process in memory with the **lock** call. **lock** takes one argument: if it is non-zero, the process will be locked into memory; if it is zero, it will be unlocked and subject to swapping. **Lock**ing is restricted to super-users or users in the super-group.

Even if a process is not swapped, some delay may be suffered when a process finishes sleeping. If experience shows that this delay is too great, the process should sleep for fewer clock-ticks and then simply loop for the remaining time. In an extreme case, the process should avoid sleeping altogether and just loop.

Programs doing asynchronous I/O can avoid sleeping by calling **aiowait** with an argument of **−1** to check the number of queued buffers without initiating a sleep. For example, in the previous example, the line:

aiowait(afd,1);

causes a sleep to occur until one or fewer requests remain outstanding. To avoid sleeping, this can be rewritten as:

```
while(aiowait(afd,-1) > 1)
        ;
```

which will continuously check the number of requests, without any sleeping.

2.9.5 Addressing the I/O Page

Device drivers are usually responsible for controlling the device registers. There is, however, some overhead involved in accessing the driver from the user-level (through the system calls **read**, **write**, or **ioctl**), and real-time applications may sometimes need to map the I/O page into their own memory and toggle the bits directly. This is done through the **phys** ("physical address") call.

Since the ability to remap memory can completely defeat system security and corrupt I/O, the **phys** can be used only by the super-user, or, in the special case of mapping the I/O page, by users in the super-group. The call is of the form:

```
phys(segreg, size, physaddr)
```

Segreg is the number of the segmentation register used; each segment is 8kb or octal 020000 bytes long, so the virtual starting address of the segment is 020000 * (multiplied by) **segreg**. **Size** indicates the length of the mapped area, in units of 64 (octal 0100) bytes. **Physaddr** is the physical address to be mapped, also in units of 64 (octal 0100) bytes.

It is only permissible to map an area of virtual memory not already in use by the program. High memory (**segreg** = 7) is always used by the stack, and lower memory (**segreg** = 0,1...) is used by the code and data portions of the program itself, but the page at location 0140000 (**segreg** = 6) is usually available, except when used by maximal size processes.

For example, to map the PDP-11 console, the call:

```
phys(6, 1, 0177775);
```

can be used. This will map virtual memory at location 6 * 020000 = 0140000 for 1 * 0100 = 0100 bytes, to the physical address of the I/O page: 0177775 *

0100 = 017777500. The I/O page at addresses 017777500 – 017777600 now resides at location 0140000 – 0140100 in virtual memory, and the console in particular can be found at 0140060. A pointer can be set to this address, and the register manipulated as needed.

Typical code to manipulate some device looks something like this:

```
#define SEGREG 6        /* virtual segmentation register to use */
#define PAGLEN 020000       /* length of a page of memory */

#define MAPADDR 0177600       /* address to pass to phys() */
#define VIRTADDR ((SEGREG * PAGLEN) + 020)

#define READY 0000200       /* set when device is ready */
#define ERR    0100000       /* error */

struct device {       /* structure of device register */
   int csr;       /* command/status register */
   int data;       /* data register */
};

devcom(info)       /* write out data to device */
int info;
{
   if (phys(SEGREG, 1, MAPADDR) < 0){
         fprintf(stderr,"bad phys call\n");
         return(1);
   }
   nice (-100);       /* set real-time priority */
```

```
    while ( (VIRTADDR->csr & READY) == 0)
                                    /* wait for device to be free */
        sleep(-1);

    VIRTADDR->data = info;          /* write data */

    while ( (VIRTADDR->csr & READY) == 0)
                                /* wait for command to complete */
        sleep(-1);

    if (VIRTADDR->csr & ERR){
        fprintf(stderr,"device error\n");
        return(1);
    }
    nice(+100);             /* relinquish real-time priority */
    return(0);
}
```

The physical address of the device register is 017760020 (the high four bits are set on to work properly for 22-bit addressing), which comes to 0177600 when divided by 64 (0100). The virtual device address is calculated as the beginning of page 6 plus the offset (in this case 020) of the register address from the 64 (0100) byte boundary on which the **phys()** call maps memory.

Devices such as A/D's can be similarly controlled, although interrupts from these devices must be handled by kernel device drivers which users write and install. (Sources for many typical devices drivers are supplied with VENIX.) Alternatively, a user program can avoid using interrupts and poll a real-time device directly through the hardware, as is done in the above example. After execution of the **phys** system call and a **nice(-100)** call, the "ready" bit can be polled in between **sleep** calls of appropriate lengths. This will guarantee instant response to a device, at the expense of processor time for other tasks. If no **sleep** calls are made, the system becomes de facto single-user and single-tasking. This facility can be used only by members of the super-group and the super-user.

2.10 Appendix

2.10.1 THE STANDARD I/O LIBRARY

The standard I/O library was designed with the following goals in mind.

1. It must be as efficient as possible, both in time and in space, so that there will be no hesitation in using it no matter how critical the application.
2. It must be simple to use, and also free of the magic numbers and mysterious calls whose use mars the understandability and portability of many programs using older packages.
3. The interface provided should be applicable on all machines, whether or not the programs which implement it are directly portable to other systems, or to other machines running a version of UNIX.

2.10.2 GENERAL USAGE

Each program using the library must have the line

```
#include <stdio.h>
```

which defines certain macros and variables. The routines are in the normal C library, so no special library argument is needed for loading. All names in the include file intended only for internal use begin with an underscore _ to reduce the possibility of collision with a user name. The names intended to be visible outside the package are

stdin
The name of the standard input file

stdout
The name of the standard output file

stderr
The name of the standard error file

EOF
is actually **−1**, and is the value returned by the read routines on end-of-file or error.

NULL
is a notation for the null pointer, returned by pointer-valued functions to indicate an error

FILE
expands to **struct _iob** and is a useful shorthand when declaring pointers to streams.

BUFSIZ
is a number (viz. 512) of the size suitable for an I/O buffer supplied by the user. See **setbuf**, below.

getc, getchar, putc, putchar, feof, ferror, fileno
are defined as macros. Their actions are described below; they are mentioned here to point out that it is not possible to redeclare them and that they are not actually functions; thus, for example, they may not have breakpoints set on them.

The routines in this package offer the convenience of automatic buffer allocation and output flushing where appropriate. The names **stdin**, **stdout**, and **stderr** are in effect constants and may not be assigned to.

2.10.3 CALLS

FILE *fopen(filename, type) char *filename, *type;
opens the file and, if needed, allocates a buffer for it. **filename** is a character string specifying the name. **type** is a character string (not a single character). It may be **"r"**, **"w"**, or **"a"** to indicate intent to read, write, or append. The value returned is a file pointer. If it is **NULL** the attempt to open failed.

FILE *freopen(filename, type, ioptr) char *filename, *type; FILE *ioptr;
The stream named by **ioptr** is closed, if necessary, and then reopened as if by **fopen**. If the attempt to open fails, **NULL** is returned, otherwise **ioptr**, which will now refer to the new file. Often the reopened stream is **stdin** or **stdout**.

int getc(ioptr) FILE *ioptr;
returns the next character from the stream named by **ioptr**, which is a pointer to a file such as returned by **fopen**, or the name **stdin**. The integer **EOF** is returned on end-of-file or when an error occurs. The null character **\0** is a legal character.

int fgetc(ioptr) FILE *ioptr;
acts like **getc** but is a genuine function, not a macro, so it can be pointed to, passed as an argument, etc.

putc(c, ioptr) FILE *ioptr;
putc writes the character **c** on the output stream named by **ioptr**, which is a value returned from **fopen** or perhaps **stdout** or **stderr**. The character is returned as value, but **EOF** is returned on error.

fputc(c, ioptr) FILE *ioptr;
acts like **putc** but is a genuine function, not a macro.

fclose(ioptr) FILE *ioptr;
The file corresponding to **ioptr** is closed after any buffers are emptied. A buffer allocated by the I/O system is freed. **fclose** is automatic on normal termination of the program.

fflush(ioptr) FILE *ioptr;
Any buffered information on the (output) stream named by **ioptr** is written out. Output files are normally buffered if and only if they are not directed to the terminal; however, **stderr** always starts off unbuffered and remains so unless **setbuf** is used, or unless it is reopened.

exit(errcode); terminates the process and returns its argument as status to the parent. This is a special version of the routine which calls **fflush** for each output file. To terminate without flushing, use **_exit**.

feof(ioptr) FILE *ioptr;
returns non-zero when end-of-file has occurred on the specified input stream.

ferror(ioptr) FILE *ioptr;
returns non-zero when an error has occurred while reading or writing the named stream. The error indication lasts until the file has been closed.

getchar(); is identical to **getc(stdin)**.

putchar(c); is identical to **putc(c, stdout)**.

char *fgets(s, n, ioptr) char *s; FILE *ioptr;
reads up to **n−1** characters from the stream **ioptr** into the character pointer **s**. The read terminates with a newline character. The newline character is placed in the buffer followed by a null character. **fgets** returns the first argument, or **NULL** if error or end-of-file occurred.

fputs(s, ioptr) char *s; FILE *ioptr;
writes the null-terminated string (character array) **s** on the stream **ioptr**. No newline is appended. No value is returned.

ungetc(c, ioptr) FILE *ioptr;
The argument character **c** is pushed back on the input stream named by **ioptr**. Only one character may be pushed back.

printf(format, a1, ...) char *format;

fprintf(ioptr, format, a1, ...) FILE *ioptr;

char *format; sprintf(s, format, a1, ...)char *s, *format;
printf writes on the standard output. **fprintf** writes on the named output stream. **sprintf** puts characters in the character array (string) named by **s**. The specifications are as described in section **printf**(3) of the *Programmer Reference Manual*.

scanf(format, a1, ...) char *format;

fscanf(ioptr, format, a1, ...) FILE *ioptr;

char *format; sscanf(s, format, a1, ...) char *s, *format;
scanf reads from the standard input. **fscanf** reads from the named input stream. **sscanf** reads from the character string supplied as **s**. **scanf** reads characters, interprets them according to a format, and stores the results in its arguments. Each routine expects as arguments a control string **format**, and a set of

arguments, each of which must be a pointer, indicating where the converted input should be stored.

scanf returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, **EOF** is returned; note that this is different from 0, which means that the next input character does not match what was called for in the control string.

fread(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;
reads **nitems** of data beginning at **ptr** from file **ioptr.** No advance notification that binary I/O is being done is required; when, for portability reasons, it becomes required, it will be done by adding an additional character to the mode-string on the **fopen** call.

fwrite(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;
Like **fread**, but in the other direction.

rewind(ioptr) FILE *ioptr;
rewinds the stream named by **ioptr.** It is not very useful except on input, since a rewound output file is still open only for output.

system(string) char *string;
The **string** is executed by the shell as if typed at the terminal.

getw(ioptr) FILE *ioptr;
returns the next word from the input stream named by **ioptr.** **EOF** is returned on end-of-file or error, but since this a perfectly good integer **feof** and **ferror** should be used. A "word" is 16 bits on the PDP-11.

putw(w, ioptr) FILE *ioptr;
writes the integer **w** on the named output stream.

setbuf(ioptr, buf) FILE *ioptr; char *buf;
setbuf may be used after a stream has been opened but before I/O has started. If **buf** is **NULL**, the stream will be unbuffered. Otherwise the buffer supplied will be used. It must be a character array of sufficient size:

char buf[BUFSIZ];

fileno(ioptr) FILE *ioptr;

returns the integer file descriptor associated with the file.

fseek(ioptr, offset, ptrname) FILE *ioptr; long offset;

The location of the next byte in the stream named by **ioptr** is adjusted. **offset** is a long integer. If **ptrname** is 0, the offset is measured from the beginning of the file; if **ptrname** is 1, the offset is measured from the current read or write pointer; if **ptrname** is 2, the offset is measured from the end of the file. The routine accounts properly for any buffering. (When this routine is used on non-UNIX systems, the offset must be a value returned from **ftell** and the ptrname must be 0).

long ftell(ioptr) FILE *ioptr;

The byte offset, measured from the beginning of the file, associated with the named stream is returned. Any buffering is properly accounted for. (On non-UNIX systems the value of this call is useful only for handing to **fseek**, so as to position the file to the same place it was when **ftell** was called.)

getpw(uid, buf) char *buf;

The password file is searched for the given integer user ID. If an appropriate line is found, it is copied into the character array **buf**, and 0 is returned. If no line is found corresponding to the user ID then 1 is returned.

char *malloc(num);

allocates **num** bytes. The pointer returned is sufficiently well aligned to be usable for any purpose. **NULL** is returned if no space is available.

char *calloc(num, size);

allocates space for **num** items each of size **size**. The space is guaranteed to be set to 0 and the pointer is sufficiently well aligned to be usable for any purpose. **NULL** is returned if no space is available .

cfree(ptr) char *ptr;
Space is returned to the pool used by **calloc**. Disorder can be expected if the pointer was not obtained from **calloc**.

The following are macros whose definitions may be obtained by including **<ctype.h>**.

isalpha(c) returns non-zero if the argument is alphabetic.

isupper(c) returns non-zero if the argument is upper-case alphabetic.

islower(c) returns non-zero if the argument is lower-case alphabetic.

isdigit(c) returns non-zero if the argument is a digit.

isspace(c) returns non-zero if the argument is a spacing character: tab, newline, carriage return, vertical tab, form feed, space.

ispunct(c) returns non-zero if the argument is any punctuation character, i.e., not a space, letter, digit or control character.

isalnum(c) returns non-zero if the argument is a letter or a digit.

isprint(c) returns non-zero if the argument is printable — a letter, digit, or punctuation character.

iscntrl(c) returns non-zero if the argument is a control character.

isascii(c) returns non-zero if the argument is an ASCII character, i.e., less than octal 0200.

toupper(c) returns the upper-case character corresponding to the lower-case letter **c**.

tolower(c) returns the lower-case character corresponding to the upper-case letter **c**.

Contents

Chapter 3

C LANGUAGE

3.1 LEXICAL CONVENTIONS

There are six classes of tokens — identifiers, keywords, constants, strings, operators, and other separators. Blanks, tabs, new-lines, and comments (collectively, "white space") as described below are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

3.1.1 Comments

The characters /* introduce a comment which terminates with the characters */. Comments do not nest.

3.1.2 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. The external name sizes include:

PDP-11	**7 characters, 2 cases**
VAX-11	**>100 characters, 2 cases**
WECo 3B 20	**>100 characters, 2 cases**

3.1.3 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**	**void**
continue	**external**	**long**	**struct**	**while**
default	**float**	**register**	**switch**	

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

3.1.4 Constants

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

3.1.4.1 Integer Constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with **0** (digit zero). An octal constant consists of the digits **0** through **7** only. A sequence of digits preceded by **0x** or **0X** (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include **a** or **A** through **f** or **F** with values 10 through 15. Otherwise, the integer constant is taken to be decimal. A decimal constant whose value exceeds the largest signed machine integer is taken to be **long**; an octal or hex constant which exceeds the largest unsigned machine integer is likewise taken to be **long**. Otherwise, integer constants are **int**.

3.1.4.2 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.

3.1.4.3 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	*ddd*	*ddd*

The escape *ddd* consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is **\0** (not followed by a digit), which indicates the character **NUL**. If the character following a backslash is not one of those specified, the behavior is undefined. A new-line character is illegal in a character constant. The type of a character constant is **int**.

3.1.4.4 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 (not both) may be missing. Either the decimal point or the **e** and the exponent (not both) may be missing. Every floating constant has type **double**.

3.1.4.5 Enumeration Constants

Names declared as enumerators (see "Structure, Union, and Enumeration Declarations" under "DECLARATIONS") have type **int**.

3.1.5 Strings

A string is a sequence of characters surrounded by double quotes, as in "...". A string has type "array of **char**" and storage class **static** (see "NAMES") and is initialized with the given characters. The compiler places a null byte (**\0**) at the end of each string so that programs which scan the string can find its end. In a string, the double quote character (") must be preceded by a \; in addition, the same escapes as described for character constants may be used.

A \ and the immediately following new-line are ignored. All strings, even when written identically, are distinct.

3.1.6 Hardware Characteristics

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

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

DEC PDP-11 HARDWARE CHARACTERISTICS

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

DEC VAX-11 HARDWARE CHARACTERISTICS

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

WECO 3B HARDWARE CHARACTERISTICS

3.2 SYNTAX NOTATION

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

{ $expression_{opt}$ }

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

3.3 NAMES

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

3.3.1 Storage Class

There are four declarable storage classes:

- Automatic
- Static
- External
- Register.

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

3.3.2 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 **short int**, **int**, and **long int**, are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers or long integers, or both, equivalent to plain integers. "Plain" integers have the natural size suggested by the host machine architecture. The other sizes are provided to meet special needs.

The properties of **enum** types (see "Structure, Union, and Enumeration Declarations" under "DECLARATIONS") are identical to those of some integer types. The implementation may use the range of values to determine how to allot storage.

Unsigned integers, declared **unsigned,** obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation. (On the PDP-11, unsigned long quantities are not supported.)

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

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

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.

3.4 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.

3.5 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 under "Arithmetic Conversions." The summary will be supplemented as required by the discussion of each operator.

3.5.1 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

VAX-11 sign-extend. On these machines, **char** variables range in value from −128 to 127.

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.

3.5.2 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.

3.5.3 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.

3.5.4 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.

3.5.5 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.

3.5.6 Arithmetic Conversions

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

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

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

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

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

e. 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.

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

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

3.6 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 + (see "Additive Operators") are those expressions defined under "Primary Expressions", "Unary Operators", and "Multiplicative Operators". Within each subpart, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators are summarized in the grammar of "SYNTAX SUMMARY".

Otherwise, the order of evaluation of expressions is undefined. In particular, the compiler considers itself free to compute subexpressions in the order it believes most efficient even if the subexpressions involve side effects. The order in which subexpression evaluation takes place is unspecified. Expressions involving a commutative and associative operator (*, +, &, |, ^) may be rearranged arbitrarily even in the presence of parentheses; to force a particular order of evaluation, an explicit temporary must be used.

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

3.6.1 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."

3.6.2 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.**

3.6.3 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).

3.6.4 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.

3.6.5 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.

3.6.6 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; 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.

3.6.7 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.

3.6.8 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.

3.6.9 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.

3.6.10 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.

3.6.11 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**.

3.6.12 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**.

3.6.13 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.

3.6.14 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.

3.6.15 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.

3.7 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.

3.7.1 Storage Class Specifiers

The sc-specifiers are:

sc-specifier:
auto
static
extern
register
typedef

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

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

A **register** declaration is best thought of as an **auto** declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few such declarations in each function are effective. Moreover, only variables of certain types will be stored in registers; on the PDP-11, they are **int** or pointer. One other restriction applies to register variables: the address-of operator **&** cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately, but future improvements in code generation may render them unnecessary.

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

3.7.2 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
    void

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

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

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

3.7.3 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.

3.7.4 Meaning of Declarators

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

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

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

Now imagine a declaration

T D1

where **T** is a type-specifier (like **int**, etc.) and **D1** is a declarator. Suppose this declaration makes the identifier have type "... **T**," where the "..." is empty if **D1** is just a plain identifier (so that the type of **x** in **"int x"** is just **int**). Then if **D1** has the form

***D**

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

If **D1** has the form

D()

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

If **D1** has the form

D[*constant-expression***]**

or

D[]

then the contained identifier has type "... array of **T**". In the first case, the constant expression is an expression whose value is determinable at compile time, whose type is **int**, and whose value is positive. (Constant expressions are defined precisely in "Constant Expressions".) When several "array of" specifications are adjacent, a multidimensional array is created; the constant expressions which specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful when the array is external and the actual definition, which allocates storage, is given elsewhere. The first constant expression may also be omitted when the declarator is followed by initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multidimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays or functions although they may return pointers; there are no arrays of functions although there may be arrays of pointers to functions. Likewise, a structure or union may not contain a function; but it may contain a pointer to a function.

As an example, the declaration

```
int i, *ip, f(), *fip(), (*pfi)();
```

declares an integer **i**, a pointer **ip** to an integer, a function **f** returning an integer, a function **fip** returning a pointer to an integer, and a pointer **pfi** to a function which returns an integer. It is especially useful to compare the last two.

The binding of ***fip()** is ***(fip())**. The declaration suggests, and the same construction in an expression requires, the calling of a function **fip**. Using indirection through the (pointer) result to yield an integer. In the declarator **(*pfi)()**, the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called; it returns an integer.

As another example,

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

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

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

declares a static 3-dimensional array of integers, with rank $3 \times 5 \times 7$. 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**.

3.7.5 Structure and Union Declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object which may, at a given time, contain any one of several members. Structure and union specifiers have the same form.

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

struct-or-union:
struct
union

The struct-decl-list is a sequence of declarations for the members of the structure or union:

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

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

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

In the usual case, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *field*; its length, a non-negative constant expression, is set off from the field name by a colon.

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

Within a structure, the objects declared have addresses which increase as the declarations are read left to right. Each nonfield member of a structure begins on an addressing boundary appropriate to its type; therefore, there may be unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field which does not fit into the space remaining in a word is put into the next word. No field may be wider than a word.

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

A struct-declarator with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, a field with a width of 0 specifies alignment of the next field at an implementation dependant boundary.

The language does not restrict the types of things that are declared as fields, but implementations are not required to support any but integer fields. Moreover, even **int** fields may be considered to be unsigned. On the PDP-11, fields are not signed and have only integer values; on the VAX-11, fields declared with **int** are treated as containing a sign. For these reasons, it is strongly recommended that fields be declared as **unsigned**. In all implementations, there are no arrays of fields, and the address-of operator **&** may not be applied to them, so that there are no pointers to fields.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most, one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

struct *identifier* { *struct-decl-list* }
union *identifier* { *struct-decl-list* }

declares the identifier to be the *structure* tag (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

struct *identifier*
union *identifier*

Structure tags allow definition of self-referential structures. Structure tags also permit the long part of the declaration to be given once and used several times. It is illegal to declare a structure or union which contains an instance of itself, but a structure or union may contain a pointer to an instance of itself.

The third form of a structure or union specifier may be used prior to a declaration which gives the complete specification of the structure or union in situations in which the size of the structure or union is unnecessary. The size is unnecessary in two situations: when a pointer to a structure or union is being declared and when a **typedef** name is declared to be a synonym for a structure or union. This, for example, allows the declaration of a pair of structures which contain pointers to each other.

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

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

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

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

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

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

```
sp->count
```

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

s.left

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

s.right – >tword[0]

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

3.7.6 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}.

3.7.7 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.

3.7.8 Type Names

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

type-name:
    type-specifier abstract-declarator

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

To avoid ambiguity, in the construction

    (abstract-declarator **)**

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

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

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

3.7.9 Typedef

Declarations whose "storage class" is **typedef** do not define storage but instead define identifiers which can be used later as if they were type keywords naming fundamental or derived types.

typedef-name:
    identifier

Within the scope of a declaration involving **typedef**, each identifier appearing as part of any declarator therein becomes syntactically equivalent to the type keyword naming the type associated with the identifier in the way described in "Meaning of Declarators". For example, after

```
typedef int MILES, *KLICKSP;
typedef struct { double re, im; } complex;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

are all legal declarations; the type of **distance** is **int**, that of **metricp** is "pointer to **int**", and that of **z** is the specified structure. The **zp** is a pointer to such a structure.

The **typedef** does not introduce brand-new types, only synonyms for types which could be specified in another way. Thus in the example above **distance** is considered to have exactly the same type as any other **int** object.

3.8 STATEMENTS

Except as indicated, statements are executed in sequence.

3.8.1 Expression Statement

Most statements are expression statements, which have the form

expression ;

Usually expression statements are assignments or function calls.

3.8.2 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.

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

3.8.4 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.

3.8.5 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.

3.8.6 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.

3.8.7 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 is executed.

The prefixes **case** and **default** do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see "Break Statement".

Usually, the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, but initializations of automatic or register variables are ineffective.

3.8.8 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.

3.8.9 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 (...)
{                  {               {
    ...                ...             ...
contin: ;          contin: ;       contin: ;
}                  } while (...);  }
```

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

3.8.10 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.

3.8.11 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.

3.8.12 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."

3.8.13 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**.

3.9 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.

3.9.1 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:
decl-specifiers$_{opt}$ *function-declarator function-body*

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 ...."

3.9.2 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**.

3.10 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: first, what may be called the lexical scope of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

3.10.1 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**.

3.10.2 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**.

3.11 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.

3.11.1 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.

3.11.2 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.

3.11.3 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.

3.11.4 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.

3.12 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**."

3.13 TYPES REVISITED

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

3.13.1 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) ...
```

3.13.2 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 (.

3.13.3 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 $i \times j \times \ldots \times k$, then **E** appearing in an expression is converted to a pointer to an $(n-1)$-dimensional array with rank $j \times \ldots \times k$. 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.

3.13.4 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.

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

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

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

3.14 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.

3.15 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.

3.16 SYNTAX SUMMARY

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

3.16.1 Expressions

The basic expressions are:

expression:
primary
* *expression*
&*lvalue*
– expression
! expression
~ expression
++ lvalue
– –lvalue
lvalue ++
lvalue – –
sizeof]*R *expression*
sizeof (*type-name*)
(type-name) expression
expression binop expression
expression ? expression : expression
lvalue asgnop expression
expression , expression

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

lvalue:

identifier
primary [expression]
lvalue . identifier
primary −> identifier
** expression*
(lvalue)

The primary-expression operators

() [] . −>

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

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

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

binop:

* / %
\+ −
>> <<
< > <= >=
== !=
&
^
|
&&
||

The conditional operator groups right to left.

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

asgnop:

= += −= *= /= %= >>= <<= &= ^= |=

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

3.16.2 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
    void

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

enum-list:
enumerator
enum-list , enumerator

enumerator:
identifier
identifier = constant-expression

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

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

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

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

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

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

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

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

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

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

type-name:
type-specifier abstract-declarator

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

typedef-name:
identifier

3.16.3 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
;

3.16.4 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* **;**

3.16.5 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"*

Contents

Chapter 4

CODE-MAPPING UNDER VENIX

4.1 INTRODUCTION

Code-mapping is a memory overlaying scheme for expanding the address space available to programs. It requires no source-code modification, and thus is entirely transparent to the programmer. However, since code-mapping does slow program execution somewhat, it is recommended only for programs whose size absolutely requires it.

4.2 DESCRIPTION

Program address space is divided into two parts: the code (or text) space, containing the program instructions, and the data space, containing the values on which the instructions operate. While PDP-11 computers may be configured with large amounts of memory (often 512kb or more), the address space allowed any individual program is ordinarily much less. Code-mapping allows a program easy access to the memory which would otherwise be out of bounds.

Code-mapping is currently implemented only for non-split I/D, machines such as the DEC Professional and other LSI-11/23 computers, the PDP-11/24, 11/34, and 11/60. On these machines, the maximum program size is ordinarily at most 64 kilobytes (kb). (As discussed later, the 8kb always reserved for the stack brings the effective program size down to 56kb). If code-mapping is used, however, the code portion of the program can become very large, and is limited only by the amount of memory physically on the computer. The data space remains restricted to 48kb.

Code-mapping works by dynamically mapping 8kb segments of the program's code in and out of the standard 64kb address space. At any given time, a single 8kb segment of the program code will be mapped in and available for execution, while all the other 8kb segments of code are mapped out of the 64kb address space. In addition to the mapped segments, there is a single "resident" 8kb segment which is not mapped but instead maintained permanently inside the 64kb space.

The mapping is done entirely with PDP-11 memory management hardware. Unlike some other overlaying schemes, code-mapping does not copy program segments to disk.

4.3 WHEN TO USE CODE-MAPPING

There are two circumstances in which a user may require code-mapping. First, a given program may simply be too large to load into memory. When the user tries to execute it, the shell immediately responds with:

a.out: too big

or

a.out: Not enough core

depending upon which shell is being used. At this point, the **size**(1) command can be run on the user's program. The result might look like this:

```
size a.out
44278 + 14466 + 12244 = 70988b = 0212514b
```

The first number given (44278) is the code space in bytes; the second two numbers (14466 + 12244) make up the data space. The total program size is given in decimal (70988b) and then in octal (0212514).

For a program to run without code-mapping, the total size, as reported by **size**, must be less than or equal to 56kb: (57344 bytes decimal, or 0160000 octal). The reason that this limit is not the full 64kb, is that **size** does not report the 8kb segment at the top of the address space which is always reserved for the stack. The size of the above program was greater than 56kb, thus code-mapping is required.

A second circumstance requiring code-mapping occurs when a program meets the 56kb limit described above, but runs out of memory while executing. This occurs when a program dynamically requests more memory, (such as through a **malloc**(3) call), and then reaches the 56kb limit. If the program is written to catch such errors, it will report the condition, typically saying something like "out of memory" and then exit. (Programs which do not check for this type of error when allocating memory may crash mysteriously, often with a core dump.) Code-mapping oftens alleviates this problem. The reason is that a code-mapped program uses exactly 16kb worth of code space, leaving exactly 40kb available for data space. The following tables illustrate this.

Memory usage of sample program before mapping:

00 – 43kb code space
44 – 56kb data space
57 – 64kb stack space

Memory usage of program after code-mapping:

00 – 08kb resident code space
08 – 16kb mapped space (mapped in and out of remaining 35kb code)
17 – 56kb data space
57 – 64kb stack space

After code-mapping, the program data space available was 40kb instead of the initial 13kb. The code portion of a code-mapped program will use exactly 16kb out of the 64kb available.

4.4 USAGE

Programs which are to be code-mapped are compiled and linked with the **–m** flag. A typical compilation and linking might look like this:

cc –m –o prog file1.c file2.c file3.c

or, if the user wishes to compile and link separately:

```
cc -c -m file1.c file2.c file3.c
cc -m -o prog
```

Note that in the latter case the **-m** flag was used both times that **cc** was executed.

In the example above, the **cc** command automatically passed the **-m** flag on to the loader (ld(**1**)) to indicate that code-mapping was desired.

4.5 SPECIAL CASE ROUTINES

There are certain cases in which the loader must be called directly. These occur if the program uses

1. **printf()** routines called with double or float type data
2. **signal()** calls
3. floating point simulation (machine has no floating-point hardware)

Each of these cases requires that the loader be called directly, in order to force certain routines to be placed in the "resident" portion of code space. This is done by passing a **-u** flag to the loader, to force certain routines to be immediately considered "unresolved", and thus loaded in first.

The loader command line should be of the following form:

```
ld -X -u f1 -u f2 ... -m /lib/crt0.o file1.o file2.o file3.o -lc
```

where *f1*, *f2*, etc., are the names of critical symbols which must be loaded first and kept resident. In the case of **printf()** calls using double or floating types, the symbols to use are **__doprnt** and *fltused*, as in

```
ld -X -u __doprnt -u fltused [-u ...] -m /lib/crt0.o ....
```

If **signal()** is used, the symbol to pass is **__signal**, as in

```
ld -X -u __signal [-u ...] -m /lib/crt0.o ...
```

Finally, if floating point simulation is used, the following command should be used:

ld −X −u fptrap [−u ...] −m −lfpsim /lib/fcrt0.o ....

Note that the floating-point simulator start-off **/lib/fcrt0.o** was used above instead of **/lib/crt0.o**. The **−u** options can be combined if the program falls under more than one of the above categories.

4.5.1 Code-Mapping Fortran Programs

The loader must also be called directly when linking Fortran programs. Fortran modules may be compiled to object form with the command:

f77 −c file1.o file2.o ....

and the linked form with code-mapping using the command:

ld −X −m /lib/crt0.o file1.o file2.o .... −lF77−lI77 −lm −lc

4.6 LIMITATIONS AND NOTES

The following is a summary of code-mapping limitations:

- The maximum size of a single object module (.o) is 8kb.
- The maximum available data space is 48kb.
- The maximum available code space is determined by the amount of physical memory minus kernel size (~48kb) minus the program's data size.
- The maximum number of functions in code-mapped space is ~1500.
- The code of a code-mapped program is automatically shared if the program is run by more than one user. This saves on the amount of physical memory being used.

4.7 PROBLEMS

Errors will arise at load-time if the number of routines which need to be mapped exceed the amount pre-allocated by the loader. The user can cause the loader to pre-allocate more room by specifying a number immediately after the **−m** flag. See "Options" below.

The most common problem found when executing a code-mapped program is a memory fault and core dump, caused by illegal memory references. This is often due to a pointer bug — specifically, an attempt to write data in code space. Type 407 programs (non-code-mapped, non-pure) will not automatically give core dumps with such a bug (because code space is not protected) and may even run successfully; however, code-mapped programs are pure, and will respond to this kind of memory violation with a core dump. In this case, code-mapping is only revealing an existing bug, not introducing a new one.

If the routines mentioned under "Special Case Routines" are not loaded as described, programs will crash with core dumps. The user should be very careful that the specified routines are indeed forced into the resident segment. A symbol table listing that uses the **nm** utility can verify that the routines are indeed in this segment. See "Optimization" for more details.

The code-mapping routine assumes that the standard C (or Fortran) calling sequence is used by the program, and that registers R0 and R1 are therefore free for mapping purposes whenever a function is called. In the rare cases when a routine is coded in assembly language, the user must either avoid using these registers on function calls, or force the assembly language routines in the resident code space. (This is the reason that users are told to use the **−u** flag with the **signal** and other routines, as described earlier.)

Certain C constructions (specifically, the passing of function-pointers) may cause code-mapping to fail if the program was assembled on a non-VENIX PDP-11 assembler. It is strongly recommended that code-mapping only be used with programs compiled under VENIX, as the VENIX assembler avoids creating these problems.

Currently, **adb** will not provide correct addresses for code-mapped program routines. This makes it useless for analysis of core dumps or break-point debugging. **adb** will not locate any symbol correctly on programs whose code segment is larger than 64kb.

4.8 OPTIMIZATION

Once a code-mapped program is working, the user may wish to delve deeper into the mapping process in order to improve the program performance. The overhead of code-mapping comes when remapping is required to reach a function. The user may speed program execution by understanding the rules which the loader uses to map the program, and adjusting the location of object modules so that as little remapping as possible is needed.

The 8kb resident code segment holds special code to aid in the mapping itself, as well as any of the "special case" routines described above which were specifically mentioned as "unresolved" with **−u** flags. There frequently remains room in the resident segment for other routines as well. Since there is no mapping required to access the resident segment, routines stored there can be reached with no overhead. In the interest of efficiency, therefore, the most commonly called routines from the C library (**/lib/libc.a**) should be forced into the resident segment, by using the **−u** flag to force them "unresolved." (Note that C-language routines are always preceded with an underscore '_'). The **−m** flag causes an automatic scan of the C library to search for any such routines. Frequently, these will be ones in the Standard I/O package, such as **printf()** or **malloc()**. In addition to those specified by the user, a number of small but very common routines are automatically loaded into the resident segment.

After scanning the C library to bring in any "unresolved" routines, the loader begins loading in the modules specified after the **−m** flag, in the order given on the command line. It fills the remaining space in the 8kb resident area with as many modules as will fit (modules are never split up). Therefore, the modules containing the most commonly used code should be specified first on the command line, so that they have a better chance of fitting into the resident segment.

After filling the resident segment, the loader assigns the remaining modules to 8kb mapped segments. Continuing to read the modules in the order given on the command line, the loader fits as many as possible into each segment before

starting a new one. No single module can exceed 8kb in code size. Since no remapping is required for function calls from routines in the same mapped segment, it makes sense to try and order the object modules so that modules containing routines which frequently call each other, wind up in the same segment. If this is not possible, the user may wish to juggle the placement of routines in source files to create object modules which fit better for code-mapping.

The important numbers to be aware of, then, are the sizes of each module's code areas. These can be determined by the **size**(1) command, for example:

size file.o

which will produce an output like:

7604 + 830 + 3852 = 12286b = 027776b

The first number (7604) indicates the module's code size. Since **size** must be done on object modules, you will have to compile your C files before running it.

The **nm** utility can be used on your linked program to determine where routines end up; it produces a list of symbols (function and variable names) and their addresses. **nm** is particularly useful in determining which routines are placed in the resident segment, and which are mapped.

nm −ng prog

will produce a list of global symbols in **prog** ordered by address. These include both mapped and resident functions.

In a listing produced by **nm**, the address of all functions will lie between 000000 and 020000 (octal). Routines which are mapped can be distinguished because their addresses all lie in the jump table, which is used to "jump" from one code-mapped routine to another. All the addresses in the jump table are 6 bytes apart, making them easy to spot. Resident functions, on the other hand, are directly addressed above the jump table. Their address will be greater than 6 bytes apart, since their addresses correspond to their actual position in the code. There will be no function addresses above 020000.

The jump table is normally positioned in the resident segment, as described above. However, if there are many functions which the user wishes to place in the resident segment, and there is free space in the data area, then the table can be located there. This is indicated by a **−md** flag instead of a **−m** (see below). If this is done, the addresses of code-mapped routines will appear above address 040000.

4.9 OPTIONS

There are several variations to the loader **−m** flag. The **−m** by itself causes the loading of the **/lib/libcmap.a** which handles the mapping system call and sets up a jump table. As described above, this table is normally placed in the resident code segment. However, the **−md** flag can be used to place the table in the data segment.

The table normally has room for roughly 500 entries, where each entry corresponds to a single function. This size can be changed by specifying the number of entries to be used, as in **−m750** or **−md200**. If the table is too small, an error message will be given. (Be aware that a small number of entries will be used in overhead, so don't try to cut the size too finely.) Currently, the maximum number of entries allowed is 1500.

Contents

Chapter 5

A C PROGRAM CHECKER— lint

5.1 GENERAL

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

5.1.1 Usage

The **lint** command has the form:

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

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

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

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

−b	Suppress messages about break statements that cannot be reached.
−c	Only check for intra-file bugs; leave external information in files suffixed with **.ln**.
−h	Do not apply heuristics (which attempt to detect bugs, improve style, and reduce waste).
−n	Do not check for compatibility with either the standard or the portable **lint** library.
−0 name	Create a lint library from input files named **llib−l***name***.ln**.
−p	Attempt to check portability to other dialects of C language.
−u	Suppress messages about function and external variables used and not defined or defined and not used.
−v	Suppress messages about unused arguments in functions.
−x	Do not report variables referred to by external declarations but never used.

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

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

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

−ly

These arguments specify libraries that contain functions used in the C language program. The source code is tested for compatibility with these libraries. This

is done by accessing library description files whose names are constructed from the library arguments. These files all begin with the comment:

/* LINTLIBRARY */

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED comments can be used to specify features of the library functions.

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

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

5.2 TYPES OF MESSAGES

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

5.2.1 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. An exception is variables which are declared through explicit **extern** statements but are never referenced; thus the statement

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

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

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The **−v** option is available to suppress the printing of messages about unused arguments. When **−v** is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments. This can be considered an active (and preventable) waste of the register resources of the machine.

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

```
/* ARGSUSED */
```

to the program before the function. This has the effect of the **−v** option for only one function. Also, the comment:

```
/* VARARGS */
```

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

```
/* VARARGS2 */
```

will cause only the first two arguments to be checked.

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

5.2.2 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 physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use", since the actual use may occur at any later time, in a data dependent fashion.

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

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

5.2.3 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 "Function Values"). If a particular place in the program cannot be reached but it is not apparent to **lint**, the comment

/* NOTREACHED */

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

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

5.2.4 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 ;

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

function *name* **contains return(e) and return**

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

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

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

In practice, some potentially serious bugs have been discovered by this feature.

On a global scale, **lint** detects cases where a function returns a value that is sometimes or never used. When the value is never used, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

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

5.2.5 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.

5.2.6 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 or her 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.

5.2.7 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, the fragment

```
char c;
    ...
if( (c = getchar()) < 0 ) ...
```

will work on one machine but will fail on machines where characters always take on positive values. The real solution is to declare *c* as an integer since **getchar** is actually returning integer values. In any case, **lint** will print the message "nonportable character comparison".

A similar issue arises with bit fields. When assignments of constant values are made to bit fields, the field may be too small to hold the value. This is especially true because on some machines bit fields are considered as signed quantities. While it may seem logical to consider that a two-bit field declared of type **int** cannot hold the value 3, the problem disappears if the bit field is declared to have type **unsigned**

5.2.8 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 enabled by the **−a** option.

5.2.9 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 supress these checks. For example, in the statement

```
*p++ ;
```

the * does nothing. This provokes the message "null effect" from **lint**. The following program fragment:

```
unsigned x ;
if( x < 0 ) ...
```

results in a test that will never succeed. Similarly, the test

```
if( x > 0 ) ...
```

is equivalent to

```
if( x != 0 )
```

which may not be the intended action. The **lint** program will print the message "degenerate unsigned comparison" in these cases. If a program contains something similar to

```
if( 1 != 0 ) ...
```

lint will print the message "constant in conditional context" since the comparison of 1 with 0 gives a constant result.

Another construction detected by **lint** involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statement

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

or

```
x<\h'-.3m'<2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and **lint** encourages this by an appropriate message.

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

5.2.10 Old Syntax

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

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

```
a = -1;
```

which could be taken as either

```
a = - 1;
```

or

```
a = -1;
```

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

A similar issue arises with initialization. The older language allowed

```
int x 1;
```

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

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

looks somewhat like the beginning of a function definition:

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

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

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

This is free of any possible syntactic ambiguity.

5.2.11 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.

5.2.12 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, the statement

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

will cause **lint** to print the message

warning: i evaluation order undefined

in order to call attention to this condition.

Contents

Chapter 6

A TUTORIAL INTRODUCTION TO ADB

6.1 INTRODUCTION

adb is a debugging program that is available on VENIX. It enables you to look at "core" files resulting from aborted programs, print output in a variety of formats, patch files, and run programs with embedded breakpoints. This chapter covers some of the most useful features of **adb**. Many of these features are explained with the aid of figures which are located at the end of this chapter. Also at the end, there is a summary of **adb** commands, formats and expressions.

6.2 OVERVIEW

6.2.1 Invocation

adb is invoked as:

adb *objfile corefile*

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

adb a.out core

or more simply:

adb

where the defaults are **a.out** and **core** respectively. The filename minus (−)

means ignore this argument as in:

adb – core

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

address ? format

OR

address / format

6.2.2 Current Address

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

0126?i

sets dot to octal 126 and prints the instruction at that address. The request:

.,10/d

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

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

6.2.3 Formats

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

b	**one byte in octal**
c	**one byte as a character**
o	**one word in octal**
d	**one word in decimal**
f	**two words in floating point**
i	**PDP-11 instruction**
s	**a null terminated character string**
a	**the value of dot**
u	**one word as unsigned integer**
n	**print a newline**
r	**print a blank space**
^	**backup dot**

(Format letters are also available for "long" values, for example, **D** for long decimal, and **F** for double floating point.) For other formats see **adb**(1) in the *User Reference Manual.*

6.2.4 General Request Meanings

The general form of a request is:

address,count command modifier

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

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

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

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

6.3 DEBUGGING C PROGRAMS

6.3.1 Debugging A Core Image

Consider the C program in Figure 1 (located at the end of this chapter). The program is used to illustrate a common error made by C programmers. The object of the program is to change the lower case 't' to upper case in the string pointed to by **charp** and then write the character string to the file indicated by argument 1. The bug shown is that the character 'T' is stored in the pointer **charp** instead of the string pointed to by **charp**. Bugs of this kind will frequently produce a **core** file automatically because of an out of bounds memory reference; in this case, an **abort()** call [see **abort**(3)] was inserted to force a core dump.

adb is invoked by:

```
adb a.out core
```

The first debugging request:

```
$c
```

is used to give a C backtrace through the subroutines called. As shown in Figure 2, only one function (**main**) was called and the arguments **argc** and **argv** have octal values 02 and 0177654 respectively. Both of these values look reasonable;

02 = two arguments, 0177654 = address on stack of parameter vector. The next request:

$C

is used to give a C backtrace plus an interpretation of all the local variables in each function and their values in octal. The backtrace shows that the last function to be called was, as expected, **abort()**.

The next request:

$r

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

The request:

$e

prints out the values of all external variables. The symbols with two underscores (__) in front of them are used internally by standard I/O routines. Symbols preceded by single underscores are either externals like **environ** and **errno** which are always present, or programs variables like **charp**. (The C compiler always inserts an underscore in front of symbols.)

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

$m

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

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

***charp/s**

which says use **charp** as a pointer in the **core** file and print the information as a character string. This printout clearly shows that the character buffer was incorrectly overwritten (there doesn't appear to be anything there) and helps identify the error. Printing the locations around **charp** shows that the buffer is unchanged but that the pointer is destroyed. It is highly suspicious that the value of **charp** is 0124; it should be pointing to a string (usually in higher memory). This indicates the bug: **charp,** rather than ***char** was set to 'T' (ASCII 0124).

Using **adb** similarly, we could print information about the arguments to a function. The request:

main.argc/d

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

***main.argv,3/o**

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

0177672/s

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

***"/s**

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

The request:

.=o

prints the current address (not its contents) in octal which has been set to the

address of the first argument. The current address, dot, is used by **adb** to "remember" its current location. It allows the user to reference locations relative to the current address, for example:

.−10/d

6.3.2 Multiple Functions

Consider the C program illustrated in Figure 3. This program calls functions **f**, **g**, and **h** until the stack is exhausted and a core image is produced.

Again you can enter the debugger via:

adb

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

$c

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

The request:

,5$C

prints the five most recent activations.

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

The request:

fcnt/d

prints the decimal value of the counter for the function **f**. Similarly **gcnt** and **hcnt** could be printed. To print the value of an automatic variable, for example

the decimal value of **x** in the last call of the function **h**, type:

h.x/d

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

6.3.3 Setting Breakpoints

Consider the C program in Figure 5. This program changes tabs into blanks. We will run this program under the control of **adb** (see Figure 6a) by:

adb a.out –

Breakpoints are set in the program as:

address:b [request]

The requests:

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

set breakpoints at the start of these functions. C does not generate statement labels. Therefore it is currently not possible to plant breakpoints at locations other than function entry points without a knowledge of the code generated by the C compiler. The above addresses are entered as **symbol + 4** so that they will appear in any C backtrace since the first instruction of each function is a call to the C save routine (**csv**). Note that some of the functions are from the C library.

To print the location of breakpoints one types:

$b

The display indicates a count field. A breakpoint is bypassed count – 1 times

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

By displaying the original instructions at the function **settab** we see that the breakpoint is set after the **jsr** to the C save routine. We can display the instructions using the **adb** request:

settab,5?ia

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

settab,5?i

which displays the instructions with only the starting address.

Notice that we accessed the addresses from the **a.out** file with the **?** command. In general when asking for a printout of multiple items, **adb** will advance the current address the number of bytes necessary to satisfy the request; in the above example five instructions were displayed and the current address was advanced 18 (decimal) bytes.

To run the program, one simply types:

:r

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

settab + 4:d

To continue execution of the program from the breakpoint, type:

:c

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

$C

to display a stack trace, or:

tabs,3/8o

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

6.3.4 Advanced Breakpoint Usage

We continue execution of the program with:

:c

See Figure 6b. **getc** is called three times and the contents of the variable **c** in the function **main** are displayed each time. The single character on the left hand edge is the output from the C program. On the third occurrence of **getc** the program stops. We can look at the full buffer of characters by typing:

ibuf+6/20c

When we continue the program with:

:c

we hit our first breakpoint at **tabpos** since there is a tab following the "This" word of the data.

Several breakpoints of **tabpos** will occur until the program has changed the tab into equivalent blanks. Since we feel that **tabpos** is working, we can remove the breakpoint at that location by:

tabpos+4:d

If the program is continued with:

:c

it resumes normal execution after **adb** prints the message

a.out:running

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

```
:c
```

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

```
:c  0
```

is typed.

Now let us reset the breakpoint at **settab** and display the instructions located there when we reach the breakpoint. This is accomplished by:

```
settab+4:b    settab,5?ia
```

It is also possible to execute the **adb** requests for each occurrence of the breakpoint but only stop after the third occurrence by typing:

```
getc+4,3:b    main.c?C    *
```

This request will print the local variable **c** in the function **main** at each occurrence of the breakpoint. The semicolon is used to separate multiple **adb** requests on a single line.

Warning: setting a breakpoint causes the value of dot to be changed; executing the program under **adb** does not change dot. Therefore:

```
settab+4:b    .,5?ia
fopen+4:b
```

will print the last thing dot was set to (in the example **fopen+4**) not the current location (**settab+4**) at which the program is executing.

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

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

could be entered after typing the above requests.

Now the display of breakpoints:

```
$b
```

shows the above request for the **settab** breakpoint. When the breakpoint at **settab** is encountered the **adb** requests are executed. Note that the location at **settab+4** has been changed to plant the breakpoint; all the other locations match their original value.

Using the functions, **f**, **g** and **h** shown in Figure 3, we can follow the execution of each function by planting non-stopping breakpoints. We call **adb** with the executable program of Figure 3 as follows:

```
adb  ex3  -
```

Suppose we enter the following breakpoints:

```
h+4:b      hcnt/d;   h.hi/;      h.hr/
g+4:b      gcnt/d;   g.gi/;      g.gr/
f+4:b      fcnt/d;   f.fi/;      f.fr/
:r
```

Each request line indicates that the variables are printed in decimal (by the specification **d**). Since the format is not changed, the **d** can be left off all but the first request.

The output in Figure 7 illustrates two points. First, the **adb** requests in the breakpoint line are not examined until the program under test is run. That means any errors in those **adb** requests is not detected until run time. At the location of the error, **adb** stops running the program.

The second point is the way **adb** handles register variables. **adb** uses the symbol table to address variables. Register variables, like **f.fr** above, have pointers to uninitialized places on the stack. Therefore the message "symbol not found" appears.

Another way of getting at the data in this example is to print the variables used in the call as:

```
f+4:b      fcnt/d;  f.a/;      f.b/;  f.fi/
g+4:b      gcnt/d;  g.p/;      g.q/;  g.gi/
:c
```

The operator / was used instead of **?** to read values from the **core** file. The output for each function, as shown in Figure 7, has the same format. For the function **f**, for example, it shows the name and value of the **external** variable **fcnt**. It also shows the address on the stack and value of the variables **a**, **b** and **fi**.

Notice that the addresses on the stack will continue to decrease until no address space is left for program execution at which time (after many pages of output) the program under test aborts. A display with names would be produced by requests like the following:

```
f+4:b      fcnt/d;  f.a/"a="d;  f.b/"b="d;  f.fi/"fi="d
```

In this format, the quoted string is printed literally and the **d** produces a decimal display of the variables. The results are shown in Figure 7.

6.3.5 Other Breakpoint Facilities

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

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

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

- The program being debugged can be single stepped by:

  :s

  If necessary, this request will start up the program being debugged and stop after executing the first instruction.

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

  address:r

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

  ,n:r

  The request:

  ,n:c

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

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

  address:c

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

  :k

6.4 MAPS

VENIX supports several executable file formats. These are used to tell the loader how to load the program file. File type 407 is the most common and is generated by a C compiler invocation such as **cc pgm.c**. A 410 file is produced by a C compiler command of the form **cc −n pgm.c**, whereas a 411 file is produced by **cc −i pgm.c**. **adb** interprets these different file formats and provides access to the different segments through a set of maps (see Figure 8). To print

the maps type:

$m

In 407 files, both text (instructions) and data are intermixed. This makes it impossible for **adb** to differentiate data from instructions and some of the printed symbolic addresses look incorrect; for example, printing data addresses as offsets from routines.

In 410 files (shared text), the instructions are separated from data and **?*** accesses the data part of the **a.out** file. The **?*** request tells **adb** to use the second part of the map in the **a.out** file. Accessing data in the **core** file shows the data after it was modified by the execution of the program. Notice also that the data segment may have grown during program execution.

In 411 files (separated I & D space), the instructions and data are also separated. However, in this case, since data is mapped through a separate set of segmentation registers, the base of the data segment is also relative to address zero. In this case since the addresses overlap it is necessary to use the **?*** operator to access the data space of the **a.out** file. In both 410 and 411 files the corresponding **core** file does not contain the program text.

Figure 9 shows the display of three maps for the same program linked as a 407, 410, 411 respectively. The b, e, and f fields are used by **adb** to map addresses into file addresses. The "f1" field is the length of the header at the beginning of the file (020 bytes for an **a.out** file and 02000 bytes for a **core** file). The "f2" field is the displacement from the beginning of the file to the data. For a 407 file with mixed text and data this is the same as the length of the header; for 410 and 411 files this is the length of the header plus the size of the text portion.

The "b" and "e' fields are the starting and ending locations for a segment. Given an address, A, the location in the file (either **a.out** or **core**) is calculated as:

$$b1 \leq A \leq e1 \Rightarrow \text{file address} = (A - b1) + f1$$
$$b2 \leq A \leq e2 \Rightarrow \text{file address} = (A - b2) + f2$$

A user can access locations by using the **adb** defined variables. The **$v** request

prints the variables initialized by **adb**:

b	**base address of data segment**
d	**length of the data segment**
s	**length of the stack**
t	**length of the text**
m	**execution type (407,410,411)**

In Figure 9 those variables not present are zero. Use can be made of these variables by expressions such as

<b

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

02000>b

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

adb reads the header of the **core** image file to find the values for these variables. If the second file specified does not seem to be a **core** file, or if it is missing then the header of the executable file is used instead.

6.5 ADVANCED USAGE

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

6.5.1 Formatted Dump

The line:

<b,-1/4o4^8Cn

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

<b
The base address of the data segment.

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

The format **4o4^8Cn** is broken down as follows:

4o Print 4 octal locations.

4^ Backup the current address 4 locations (to the original start of the field).

8C Print 8 consecutive characters using an escape convention; each character in the range 0 to 037 is printed as @ followed by the corresponding character in the range 0140 to 0177. An @ is printed as @@.

n Print a newline.

The request:

<b,<d/4o4^8Cn

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

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

adb a.out core < dump

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

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

The request **120$w** sets the width of the output to 120 characters (normally, the width is 80 characters). **adb** attempts to print addresses as:

symbol + offset

The request **4095$s** increases the maximum permissible offset to the nearest symbolic address from 255 (default) to 4095. The request = can be used to print literal strings. Thus, headings are provided in this **dump** program with requests of the form:

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

that spaces three lines and prints the literal string. The request **$v** prints all non-zero **adb** variables (see Figure 8). The request **0$s** sets the maximum offset for symbol matches to zero thus suppressing the printing of symbolic labels in favor of octal values. Note that this is only done for the printing of the data segment. The request:

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

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

Figure 11 shows the results of some formatting requests on the C program of Figure 10.

6.5.2 Directory Dump

As another illustration (Figure 12) consider a set of requests to dump the contents of a directory (which is made up of an integer *inumber* followed by a 14 character name):

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

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

6.5.3 Ilist Dump

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

```
adb /dev/src -
02000>b
?m <b
<b,-1? "flags"8ton"links,uid,gid"8t3bn",size"8tbrdn"addr"8t8un"times"8t2Y2na
```

In this example the value of the base for the map was changed to 02000 (by saying **?m<b**) since that is the start of an **ilist** within a file system. An artifice (**brd** above) was used to print the 24 bit size field as a byte, a space, and a decimal integer. The last access time and last modify time are printed with the **2Y** operator. Figure 12 shows portions of these requests as applied to a directory and file system.

6.5.4 Converting Values

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

```
072 = odx
```

will print

072 58 #3a

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

'a' = co

prints

a 0141

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

6.6 PATCHING

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

?l value

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

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

adb −w file1 file2

When called with this option, *file1* and *file2* are created if necessary and opened for both reading and writing.

For example, consider the C program shown in Figure 10. We can change the word "This" to "The " in the executable file for this program, **ex7**, by using the following requests:

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

The request **?l** starts at dot and stops at the first match of "Th" having set dot to the address of the location found. Note the use of **?** to write to the **a.out** file. The form **?*** would have been used for a 411 file.

More frequently the request will be typed as:

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

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

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

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

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

6.7 ANOMALIES

Users should be aware of the following **adb** anomalies:

1. Function calls and arguments are put on the stack by the C save routine. Putting breakpoints at the entry point to routines means that the function appears not to have been called when the breakpoint occurs.

2. When printing addresses, **adb** uses either text or data symbols from the **a.out** file. This sometimes causes unexpected symbol names to be printed with data (e.g. **savr5+022**). This does not happen if **?** is used for text (instructions) and / for data.

3. **adb** cannot handle C register variables in the most recently activated function.

Figure 1: C program with pointer bug

```
#include <stdio.h>
char *charp "this is a sentence.";

main(argc,argv)
int argc;
char **argv;
{
    FILE *fp;
    char    cc;

    if(argc < 2) {
      printf("Input file missing\n");
      exit(8);
    }

    if((fp = fopen(argv[1],"w")) == 0){
      printf("%s : can't open for writing\n", argv[1]);
      exit(8);
    }
    charp = 'T';
    abort();
    while(cc =  *charp++)
      fputc(cc,fp);
}
```

Figure 2: ADB output for C program of Figure 1

```
adb a.out core
$c
__abort()
~main(02,0177654)
$C
__abort()
~main(02,0177654)
                argc:       02
                argv:       0177654
                fp:         05634
                cc:         0
$r
ps              0170004
pc              03520       __abort + 06
sp              0177612
r5              0177622
r4              0
r3              0
r2              0
r1              0
r0              03516
__abort + 06:               clr             r0
$e
__environ:  0177662
__charp:    0124
____iob:    07120
____sobuf:  0
____lastbu: 06044
__errno:    0
____sibuf:  0
__end:      0
$m
? map       'a.out'
b1 = 0      e1 = 06116  f1 = 020
b2 = 0      e2 = 06116  f2 = 020
/ map       'core'
b1 = 0      e1 = 010200 f1 = 02000
b2 = 0175400  e2 = 0200000      f2 = 012200
*charp/s
0124:
charp/s
__charp:    T

__charp + 02:     this is a sentence.

__charp + 026:    Input file missing
main.argc/d
0177646:    2
```

```
*main.argv/3o
0177654:  0177672    0177700    0
0177672/s
0177672/s:    a.out
*main.argv/3o
0177654:  0177672    0177700    0
*"/s
0177672:      a.out
.=o
0177672
.-10/d
0177756:  2
$q
```

Figure 3: Multiple function C program for stack trace illustration

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

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

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

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

Figure 4: ADB output for C program of Figure 3

```
adb
$c
~h(04452,04451)
~g(04453,011124)
~f(02,04451)
~h(04450,04447)
~g(04451,011120)
~f(02,04447)
~h(04446,04445)
~g(04447,011114)
~f(02,04445)
~h(04444,04443)
HIT DEL KEY
adb
,5$C
~h(04452,04451)
            x:          04452
            y:          04451
            hi:         ?
~g(04453,011124)
            p:          04453
            q:          011124
            gi:         04451
            gr:         ?
~f(02,04451)
            a:          02
            b:          04451
            fi:         011124
            fr:         04453
~h(04450,04447)
            x:          04450
            y:          04447
            hi:         04451
            hr:         02
~g(04451,011120)
            p:          04451
            q:          011120
            gi:         04447
            gr:         04450
fcnt/d
__fcnt:     1173
gcnt/d
__gcnt:     1173
hcnt/d
__hcnt:     1172
h.x/d
022004:     2346
$q__end
```

Figure 5: C program to decode tabs

```
#define MAXLINE   80
#define YES       1
#define NO        0
#define TABSP     8

char      input[] "data";
char      ibuf[518];
int       tabs[MAXLINE];

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

    ptab = tabs;
    settab(ptab);    /*Set initial tab stops */
    col = 1;
    if(fopen(input,ibuf) < 0) {
        printf("%s : not found\n",input);
        exit(8);
    }
    while((c = getc(ibuf)) != -1) {
        switch(c) {
            case '\t':    /* TAB */
                while(tabpos(col) != YES) {
                    putchar(' ');    /* put BLANK */
                    col++ ;
                }
                break;
            case '\n':    /*NEWLINE */
                putchar('\n');
                col = 1;
                break;
            default:
                putchar(c);
                col++ ;
        }
    }
}

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

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

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

Figure 6a: ADB output for C program of Figure 5

```
adb a.out -
settab+4:b
fopen+4:b
getc+4:b
tabpos+4:b
$b
breakpoints
count   bkpt        command
1   ~tabpos+04
1   __getc+04
1   __fopen+04
1   ~settab+04
settab,5?ia
~settab:    jsr         r5,csv
~settab+04:             tst         -(sp)
~settab+06:             clr         0177770(r5)
~settab+012:            cmp         $0120,0177770(r5)
~settab+020:            blt         ~settab+076
~settab+022:
settab,5?i
~settab:    jsr         r5,csv
        tst     -(sp)
        clr     0177770(r5)
        cmp     $0120,0177770(r5)
        blt     ~settab+076
:r
a.out: running
breakpoint ~settab+04:          tst         -(sp)
settab+4:d
:c
a.out: running
breakpoint __fopen+04:          mov         04(r5),nulstr+012
$C
__fopen(02302,02472)
~main(01,0177770)
    col:    01
    c:      0
    ptab:   03500
tabs,3/8o
03500:      01      0       0       0       0       0       0       0
            01      0       0       0       0       0       0       0
            01      0       0       0       0       0       0       0
```

Figure 6b: ADB output for C program of Figure 5

```
:c
a.out: running
breakpoint __getc + 04:          mov      04(r5),r1
ibuf + 6/20c
____cleanu + 0202:    This   is   a test   of
:c
a.out: running
breakpoint ~tabpos + 04:         cmp      $0120,04(r5)
tabpos + 4:db + 4:b  settab,5?ia
settab + 4:bsettab,5?ia;  0
getc + 4,3:bmain.c?C;  0
settab + 4:bsettab,5?ia;  ptab/o;  0
$b
breakpoints
count      bkpt       command
1          ~tabpos + 04
3          __getc + 04 main.c?C;0
1          __fopen + 04
1          ~settab + 04 settab,5?ia;ptab?o;0
~settab:   jsr        r5,csv
~settab + 04:          bpt
~settab + 06:          clr      0177770(r5)
~settab + 012:         cmp      $0120,0177770(r5)
~settab + 020:         blt      ~settab + 076
~settab + 022:
0177766:   0177770
0177744:   @`
T0177744:  T
h0177744:  h
i0177744:  i
s0177744:  s
```

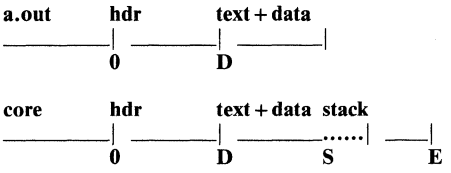
Figure 7: ADB output for C program with breakpoints

```
adb ex3 -
h+4:b hcnt/d; h.hi/; h.hr/
g+4:b gcnt/d; g.gi/; g.gr/
f+4:b fcnt/d; f.fi/; f.fr/
:r
ex3: running
_fcnt:              0
0177732:  214
symbol not found
f+4:b fcnt/d; f.a/; f.b/; f.fi/
g+4:b gcnt/d; g.p/; g.q/; g.gi/
h+4:b hcnt/d; h.x/; h.y/; h.hi/
:c
ex3: running
_fcnt:              0
0177746:  1
0177750:  1
0177732:  214
_gcnt:              0
0177726:  2
0177730:  3
0177712:  214
_hcnt:              0
0177706:  2
0177710:  1
0177672:  214
_fcnt:              1
0177666:  2
0177670:  3
0177652:  214
_gcnt:              1
0177646:  5
0177650:  8
0177632:  214
HIT DEL
f+4:b fcnt/d; f.a/"a = "d; f.b/"b = "d; f.fi/"fi = "d
g+4:b gcnt/d; g.p/"p = "d; g.q/"q = "d; g.gi/"gi = "d
h+4:b hcnt/d; h.x/"x = "d; h.y/"h = "d; h.hi/"hi = "d
:r
ex3: running
_fcnt:              0
0177746:  a = 1
0177750:  b = 1
0177732:  fi = 214
_gcnt:              0
0177726:  p = 2
0177730:  q = 3
0177712:  gi = 214
```

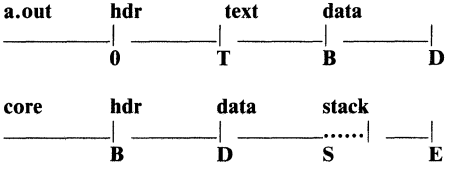
```
__hcnt:                 0
0177706:   x = 2
0177710:   y = 1
0177672:   hi = 214
__fcnt:                 1
0177666:   a = 2
0177670:   b = 3
0177652:   fi = 214
HIT DEL
$q
```

Figure 8: ADB address maps

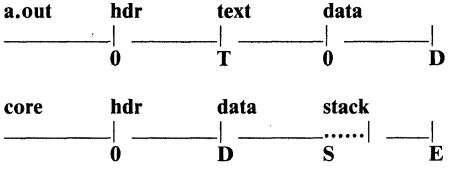
407 files



410 files (shared text)



411 files (separated I and D space)



The following *adb* variables are set.

		407	410	411
b	base of data	0	B	0
d	length of data	D	D−B	D
s	length of stack	S	S	S
t	length of text	0	T	T

Figure 9: ADB output for maps

```
adb map407 core407
$m
text map    `map407'
b1 = 0                  e1 = 0256       f1 = 020
b2 = 0                  e2 = 0256       f2 = 020
data map    `core407'
b1 = 0                  e1 = 0300       f1 = 02000
b2 = 0175400            e2 = 0200000    f2 = 02300
$v
variables
d = 0300
m = 0407
s = 02400
$q


adb map410 core410
$m
text map    `map410'
b1 = 0                  e1 = 0200       f1 = 020
b2 = 020000             e2 = 020116     f2 = 0220
data map    `core410'
b1 = 020000             e1 = 020200     f1 = 02000
b2 = 0175400            e2 = 0200000    f2 = 02200
$v
variables
b = 020000
d = 0200
m = 0410
s = 02400
t = 0200
$q


adb map411 core411
$m
text map    `map411'
b1 = 0                  e1 = 0200       f1 = 020
b2 = 0                  e2 = 0116       f2 = 0220
data map    `core411'
b1 = 0                  e1 = 0200       f1 = 02000
b2 = 0175400            e2 = 0200000    f2 = 02200
```

```
$v
variables
d = 0200
m = 0411
s = 02400
t = 0200
$q
```

Figure 10: Simple C program for illustrating formatting and patching

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

Figure 11: ADB output illustrating fancy formats

```
adb map410 core410
<b,-1/8ona
020000:          0       064124  071551  0644400201630201410641430711 41

__str1+016:      061541  062564  020162  07216306456206355 6002

__number:
__number:        0710    0       02322   040240006412407155 1064440

__str2+06:       020163  064164  020145  062563067543062156061440060550

__str2+026:      060562  072143  071145  071440711640671510 1470

savr5+02:        0       0       0       00000

<b,20/4o4^8Cn
020000:                  0       064124  071551064440@`@`This i
                 020163  020141  064143  071141s a char
                 061541  062564  020162  072163acter st
                 064562  063556  0       02ring@`@`@b@`

__number:        0710    0       02322   040240H@a@`@`R@d @@
                 0       064124  071551  064440@`@`This i
                 020163  064164  020145  062563s the se
                 067543  062156  061440  060550cond cha
                 060562  072143  071145  071440racter s
                 071164  067151  0147    0tring@`@`@`
                 0       0       0       0@`@`@`@`@`@`@`@`
                 0       0       0       0@`@`@`@`@`@`@`@`
data address not found
<b,20/4o4^8t8cna
020000:                  0       064124  071551064440This i
__str1+06:       020163  020141  064143  071141s a char
__str1+016:      061541  062564  020162  072163acter st
__str1+026:      064562  063556  0       02ring
__number:
__number:        0710    0       02322   040240HR
__fpt+02:        0       064124  071551  064440This i
__str2+06:       020163  064164  020145  062563s the se
__str2+016:      067543  062156  061440  060550cond cha
__str2+026:      060562  072143  071145  071440racters
__str2+036:      071164  067151  0147    0tring
savr5+02:        0       0       0       0
savr5+012:       0       0       0       0
data address not found
<b,10/2b8t^2cn
020000:                  0       0
```

```
__str1:      0124   0150      Th
      0151   0163      is
      040    0151      i
      0163   040       s
      0141   040       a
      0143   0150      ch
      0141   0162      ar
      0141   0143      ac
       0164                  0145                  te
$Q
```

Figure 12: Directory and inode dumps

```
adb dir -
=nt"Inode"t"Name"
0,-1?ut14cn

                        Inode   Name
0:                      652     .
                        82      ..
                        5971    cap.c
                        5323    cap
                        0       pp



adb /dev/src -
02000>b
?m<b
new map                         `/dev/src'
b1 = 02000                      e1  = 0100000000        f1 = 0
b2 = 0                              e2      = 0         f2 = 0
$v
variables
b = 02000
<b,-1?"flags"8ton"links,uid,gid"8t3bn"size"8tbrdn"addr"8t8un"times"8t2Y2na
02000:                  flags   073145
        links,uid,gid   0163    0164    0141
        size            0162    10356
        addr            28770   8236    25956   27766254558236259562520 6
        times           1976 Feb 5 08:34:56     1975 Dec 28 10:55:15

02040:                  flags   024555
        links,uid,gid   012     0163    0164
        size            0162    25461
        addr            8308    30050   8294    25130152162689029806107 84
        times           1976 Aug 17 12:16:51    1976 Aug 17 12:16:51

02100:                  flags   05173
        links,uid,gid   011     0162    0145
        size            0147    29545
        addr            25972   8306    28265   83082564215216231425970
        times           1977 Apr 2 08:58:01     1977 Feb 5 10:21:44
```

6.7.1 ADB Summary

6.7.1.1 Command Summary

a) formatted printing

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

b) breakpoint and program control

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

c) miscellaneous printing

$b	print current breakpoints
$c	C stack trace
$e	external variables
$f	floating registers

$m	print ADB segment maps
$q	exit from ADB
$r	general registers
$s	set offset for symbol match
$v	print ADB variables
$w	set output line width

d) calling the shell

!	call *shell* to read rest of line

e) assignment to variables

>name	assign dot to variable or register *name*

6.7.1.2 Format Summary

a	the value of dot
b	one byte in octal
c	one byte as a character
d	one word in decimal
f	two words in floating point
i	PDP 11 instruction
o	one word in octal
n	print a newline
r	print a blank space
s	a null terminated character string
nt	move to next *n* space tab
u	one word as unsigned integer
x	hexadecimal
Y	date
^	backup dot
...	print string

6.7.1.3 Expression Summary

a) expression components

decimal integer	e.g. 256
octal integer	e.g. 0277
hexadecimal	e.g. #ff

symbols	e.g. flag __main main.argc
variables	e.g. <b
registers	e.g. <pc <r0
(expression)	expression grouping

b) dyadic operators

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

c) monadic operators

~	not
*	contents of location
–	integer negate

Contents

Chapter 7

FORTRAN 77

This chapter describes the compiler and run-time system for Fortran 77 as implemented on the VENIX system. This chapter also describes the interfaces between procedures and the file formats assumed by the I/O system.

7.1 USAGE

The command to run the compiler is

f77 *options file*

The **f77**(1) command is a general purpose command for compiling and loading Fortran and Fortran-related files into an executable module. Ratfor (preprocessor) source files will be translated into Fortran before being presented to the Fortran compiler. The **f77** command invokes the C compiler to translate C source files and invokes the assembler to translate assembler source files. Object files will be link edited. [The **f77**(1) and **cc**(1) commands have slightly different link editing sequences. Fortran programs need two extra libraries (libI77.a, libF77.a) and an additional startup routine.] The following file name suffixes are understood:

.f	Fortran source file
.r	Ratfor source file
.c	C language source file

.s	Assembler source file
.o	Object file

7.2 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.

7.2.1 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.

7.2.2 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.

7.2.3 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.

7.2.4 Recursion

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

7.2.5 Automatic Storage

Two new keywords recognized are **static** and **automatic.** These keywords may appear as "types" in type statements and in **implicit** statements. Local variables are static by default; there is exactly one copy of the datum, and its value is retained between calls. There is one copy of each variable declared **automatic** for each invocation of the procedure. Automatic variables may not appear in **equivalence, data,** or **save** statements.

7.2.6 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 elsewhere on the line is treated as another kind of blank by the compiler.

In the Fortran 77 Standard, there are only 26 letters — Fortran is a one-case language. Consistent with ordinary system usage, the new compiler expects lowercase input. By default, the compiler converts all uppercase characters to lowercase except those inside character constants. However, if the **−U** compiler option is specified, uppercase letters are not transformed. In this mode, it is possible to specify external names with uppercase letters in them and to have distinct variables differing only in case. Regardless of the setting of the option, keywords will only be recognized in lowercase.

7.2.7 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.

7.2.8 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.

7.2.9 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.

7.2.10 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.

7.2.11 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.

7.2.12 One-Trip DO Loops

The Fortran 77 American National Standard requires that the range of a **do** loop not be performed if the initial value is already past the limit value, as in

```
do 10 i = 2, 1
```

The 1966 Standard stated that the effect of such a statement was undefined, but it was common practice that the range of a **do** loop would be performed at least once. In order to accommodate old programs though they were in violation of the 1966 Standard, the **−onetrip** compiler option causes nonstandard loops to be generated.

7.2.13 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.

7.2.14 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.

7.2.15 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**).

For more information on the Fortran intrinsic function commands, see the *User Reference Manual*.

7.3 VIOLATIONS OF THE STANDARD

The following paragraphs describe only three known ways in which the VENIX system implementation of Fortran 77 violates the new American National Standard.

7.3.1 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, as in the following example:

```
real a(4)
double precision b,c
equivalence (a(1),b), (a(4),c)
```

Some machines require that double precision quantities be on double word boundaries; other machines run inefficiently if this alignment rule is not observed. It is possible to tell which equivalenced and common variables suffer from a forced odd alignment, but every double-precision argument would have to be assumed on a bad boundary. To load such a quantity on some machines, it would be necessary to use two separate operations. The first operation would be to move the upper and lower halves into the halves of an aligned temporary. The second would be to load that double-precision temporary. The reverse would be needed to store a result. All double-precision real and complex quantities are required to fall on even word boundaries on machines with corresponding hardware requirements and to issue a diagnostic if the source code demands a violation of the rule.

7.3.2 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.** Code is correct if there are no **character** arguments.

7.3.3 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 the record which has already been processed. The implementation uses "seeks"; so if the unit is not one which allows seeks (such as a terminal) the program is in error. A benefit of the implementation chosen is that there is no upper limit on the length of a record nor is it necessary to predeclare any record lengths except where specifically required by Fortran or the operating system.

7.4 INTERPROCEDURE INTERFACE

To be able to write C language procedures that call or are called by Fortran procedures, it is necessary to know the conventions for procedure names, data representation, return values, and argument lists that the compiled code obeys.

7.4.1 Procedure Names

On VENIX systems, the name of a common block or a Fortran procedure has an underscore appended to it by the compiler to distinguish it from a C language procedure or external variable with the same user-assigned name. Fortran library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

7.4.2 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.

7.4.3 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 — a data address and a length. Thus,

```
character*15 function g( . . . )
```

is equivalent to

```
char result[ ];
long int length;
g__(result, length, . . .)
 . . .
```

and could be invoked in C language by

```
char chars[15];
 . . .
g__(chars, 15L, . . . );
```

Subroutines are invoked as if they were **integer**-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function but are used to do an indexed branch in the calling procedure. (If the subroutine has no entry points with alternate return arguments, the returned value is undefined.) The statement

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

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

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

7.4.4 Argument Lists

All Fortran arguments are passed by address. In addition, for every argument that is of type character or that is a dummy procedure, an argument giving the length of the value is passed. (The string lengths are **long int** quantities passed by value.) The order of arguments is then:

Extra arguments for complex and character functions
Address for each datum or function
A long int for each character or procedure argument

Thus, the call in

```
external f
character*7 s
integer b(3)
 . . .
call sam(f, b(2), s)
```

is equivalent to that in

```
int f();
char s[7];
long int b[3];
 . . .
sam__(f, &b[1], s, 0L, 7L);
```

Note that the first element of a C language array always has subscript 0, but Fortran arrays begin at 1 by default. Fortran arrays are stored in column-major order; C language arrays are stored in row-major order.

7.5 FILE FORMATS

7.5.1 Structure of Fortran Files

Fortran requires four kinds of external files: *sequential formatted* and *unformatted*, and *direct formatted* and *unformatted*. On VENIX systems, these are all implemented as ordinary files which are assumed to have the proper internal structure.

Fortran I/O is based on "records." When a direct file is opened in a Fortran program, the record length of the records must be given; and this is used by the Fortran I/O system to make the file look as if it is made up of records of the given length. In the special case that the record length is given as 1, the files are not considered to be divided into records but are treated as byte-addressable byte strings; i.e., as ordinary files on the VENIX system. (A read or write request on such a file keeps consuming bytes until satisfied rather than being restricted to a single record.)

The peculiar requirements on sequential unformatted files make it unlikely that they will ever be read or written by any means except Fortran I/O statements. Each record is preceded and followed by an integer containing the record's length in bytes.

The Fortran I/O system breaks sequential formatted files into records while reading by using each new-line as a record separator. The result of reading off the end of a record is undefined according to the Fortran 77 American National Standard. The I/O system is permissive and treats the record as being extended by blanks. On output, the I/O system will write a new-line at the end of each record. It is also possible for programs to write new-lines for themselves. This is an error, but the only effect will be that the single record the user thought was written will be treated as more than one record when being read or backspaced over.

7.5.2 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 **open**ed 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.

Contents

Chapter 8

RATFOR

8.1 GENERAL

This chapter describes the **Ratfor**(1) preprocessor. It is assumed that the user is familiar with the current implementation of Fortran 77 on the VENIX 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.

8.2 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 VENIX system uses *&* in column 1), **−h**, which causes quoted strings to be turned into *n*H constructs and **−C**, which causes Ratfor comments to be copied into the generated Fortran.

8.3 STATEMENT GROUPING

The Ratfor language provides a statement grouping facility. A group of statements can be treated as a unit by enclosing them in the braces { and }. For example, the Ratfor code

```
if (x > 100)
    { call error("x>100"); err = 1; return }
```

will be translated by the Ratfor preprocessor into **Fortran** equivalent to

```
      if (x .le. 100) goto 10
          call error(5hx>100)
          err = 1
          return
10    ...
```

which should simplify programming effort. By using { and }, a group of statements can be used instead of a single statement.

Also note in the previous Ratfor example that the character > was used instead of **.GT.** in the **if** statement. The Ratfor preprocessor translates this C language

type operator to the appropriate Fortran operator. More on relationship operators later.

In addition, many Fortran compilers permit character strings in quotes (like "*x>100*"). But others, like ANSI Fortran 66, do not. Ratfor converts it into the right number of *H*s.

The Ratfor language is free form. Statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. The previous example could also be written as

```
if (x > 100) {
    call error("x>100")
    err = 1
    return
}
```

which shows grouped statements spread over several lines. In this case, no semicolon is needed at the end of each line because Ratfor assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the **if** is a single statement, no braces are needed.

8.4 THE "if-else" CONSTRUCTION

The Ratfor language provides an **else** statement. The syntax of the **if-else** construction is:

if (*legal Fortran condition*)
    ratfor statement
else
    ratfor statement

where the **else** part is optional. The *legal Fortran condition* is anything that can legally go into a Fortran Logical **IF** statement. The Ratfor preprocessor does not check this clause since it does not know enough Fortran to know what is permitted. The "ratfor" *statement* is any Ratfor or Fortran statement or any collection of them in braces. For example:

```
if (a < = b)
    { sw = 0; write(6, 1) a, b }
else
    { sw = 1; write(6, 1) b, a }
```

is a valid Ratfor **if-else** construction. This writes out the smaller of *a* and *b*, then the larger, and sets *sw* appropriately.

As before, if the statement following an **if** or an **else** is a single statement, no braces are needed.

8.4.1 Nested "if" Statements

The statement that follows an **if** or an **else** can be any Ratfor statement including another **if** or **else** statement. In general, the structure

if *(condition) action*
else if *(condition) action*
else *action*

provides a way to write a multibranch in Ratfor. (The Ratfor language also provides a **switch** statement which could be used instead, under certain conditions.) The last **else** handles the "default" condition. If there is no default action, this final **else** can be omitted. Thus, only the actions associated with the valid condition are performed. For example:

```
if (x < 0)
    x = 0
else if (x > 100)
    x = 100
```

will ensure that x is not less than 0 and not greater than 100.

Nested **if** and **else** statements could result in ambiguous code. In Ratfor when there are more **if** statements than **else** statements, **else** statements are associated with the closest previous **if** statement that currently does not have an associated **else** statement. For example:

```
if (x > 0)
if (y > 0)
write(6,1) x, y
else
write(6,2) y
```

is interpreted by the Ratfor preprocessor as

```
if (x > 0) {
    if (y > 0)
        write(6, 1) x, y
    else
        write(6, 2) y
}
```

in which the braces are assumed. If the other association is desired it *must* be written as

```
if (x > 0) {
    if (y > 0)
        write(6, 1) x, y
}
else
    write(6, 2) y
```

with the braces specified.

8.5 THE "switch" STATEMENT

The **switch** statement provides a way to express multiway branches which branch on the value of some *integer*-valued expression. The syntax is

```
switch (expression) {
    case expr1 :
    statements
    case expr2, expr3 :
    statements
    ...
    default:
    statements
}
```

where each **case** is followed by an integer expression (or several integer expressions separated by commas). The **switch** *expression* is compared to each **case** *expr* until a match is found. Then the *statements* following that **case** are executed. If no **cases** match *expression*, then the *statements* following **default** are executed. The **default** section of a **switch** is optional.

When the *statements* associated with a **case** are executed, the entire **switch** is exited immediately. This is different from C language.

8.6 THE "do" STATEMENT

The **do** statement in Ratfor is quite similar to the **DO** statement in Fortran except that it uses no statement number (braces are used to mark the end of the **do** instead of a statement number). The syntax of the **ratfor do** statement is

```
do legal-Fortran-DO-text {
   ratfor statements
}
```

The *legal-Fortran-DO-text* must be something that can legally be used in a Fortran **DO** statement. Thus if a local version of Fortran allows **DO** limits to be expressions (which is not currently permitted in ANSI Fortran 66), they can be used in a **ratfor do** statement. The *ratfor statements* are enclosed in braces; but as with the **if**, a single statement need not have braces around it. For example, the following code sets an array to zero:

```
do i = 1, n
    x(i) = 0.0
```

and the code

```
do i = 1, n
   do j = 1, n
      m(i, j) = 0
```

sets the entire array *m* to zero.

8.7 THE "break" AND "next" STATEMENTS

The Ratfor **break** and **next** statements provide a means for leaving a loop early and one for beginning the next iteration. The **break** causes an immediate exit from the **do**; in effect, it is a branch to the statement after the **do**. The **next** is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this code skips over negative values in an array

```
do i = 1, n {
   if (x(i) < 0.0)
      next
   process positive element
}
```

The **break** and **next** statements will also work in the other Ratfor looping constructions and will be discussed with each looping construction.

The **break** and **next** can be followed by an integer to indicate breaking or iterating that level of enclosing loop. For example:

```
break 2
```

exits from two levels of enclosing loops, and

```
break 1
```

is equivalent to **break**. The

```
next 2
```

iterates the second enclosing loop.

8.8 THE "while" STATEMENT

The Ratfor language provides a **while** statement. The syntax of the **while** statement is

while (*legal-Fortran-condition*)
  ratfor statement

As with the **if**, *legal-Fortran-condition* is something that can go into a Fortran Logical **IF**, and *ratfor statement* is a single statement which may be multiple statements enclosed in braces.

For example, suppose *nextch* is a function which returns the next input character both as a function value and in its argument. Then a **while** loop to find the first nonblank character could be

```
while (nextch(ich) == iblank)
    ;
```

where a semicolon by itself is a null statement (which is necessary here to mark the end of the **while**). If the semicolon were not present, the **while** would control the next statement. When the loop is exited, *ich* contains the first nonblank.

8.9 THE "for" STATEMENT

The **for** statement is another Ratfor loop. A **for** statement allows explicit initialization and increment steps as part of the statement.

The syntax of the **for** statement is

for (*init* ; *condition* ; *increment*)
  ratfor statement

where *init* is any single Fortran statement which is executed once before the loop begins. The *increment* is any single Fortran statement that is executed at the end of each pass through the loop before the test. The *condition* is again anything that is legal in a Fortran Logical **IF**. Any of *init*, *condition*, and

increment may be omitted although the semicolons must always be present. A nonexistent *condition* is treated as always true, so

```
for (;;)
```

is an infinite loop.

For example, a Fortran **DO** loop could be written as

```
for (i = 1; i <= n; i = i + 1) ...
```

which is equivalent to

```
i = 1
while (i <= n) {
    ...
    i = i + 1
}
```

The initialization and increment of *i* have been moved into the **for** statement.

The **for**, **do**, and **while** versions have the advantage that they will be done zero times if *n* is less than 1. In addition, the **break** and **next** statements work in a **for** loop.

The *increment* in a **for** need not be an arithmetic progression. The program

```
sum = 0.0
for (i = first; i > 0; i = ptr(i))
    sum = sum + value(i)
```

steps through a list (stored in an integer array *ptr*) until a zero pointer is found while adding up elements from a parallel array of values. Notice that the code also works correctly if the list is empty.

8.10 THE "repeat-until" STATEMENT

There are times when a test needs to be performed at the bottom of a loop after one pass through. This facility is provided by the **repeat-until** statement. The syntax for the **repeat-until** statement is

repeat
  ratfor statement
until (*legal-Fortran-condition*)

where *ratfor-statement* is done once, then the *condition* is evaluated. If it is true, the loop is exited; if it is false, another pass is made.

The **until** part is optional, so a **repeat** by itself is an infinite loop. A **repeat-until** loop can be exited by the use of a **stop, return,** or **break** statement or an implicit stop such as running out of input with a **READ** statement.

As stated before, a **break** statement causes an immediate exit from the enclosing **repeat-until** loop. A **next** statement will cause a skip to the bottom of a **repeat-until** loop (i.e., to the **until** part).

8.11 THE "return" STATEMENT

The standard Fortran mechanism for returning a value from a routine uses the name of the routine as a variable. This variable can be assigned a value. The last value stored in it is the value returned by the function. For example, in a Fortran routine named *equal*, the statements

```
equal = 0
return
```

cause *equal* to return zero.

The Ratfor language provides a **return** statement similar to the C language **return** statement. In order to return a value from any routine, the **return** statement has the syntax

return (*expression*)

where *expression* is the value to be returned.

If there is no parenthesized expression after **return**, no value is returned.

8.12 THE "define" STATEMENT

The Ratfor language provides a **define** statement similar to the C language version. Any string of alphanumeric characters can be defined as a name. Whenever that name occurs in the input (delimited by nonalphanumerics), it is replaced by the rest of the definition line. (Comments and trailing white spaces are stripped off.) A defined name can be arbitrarily long and must begin with a letter.

Usually the **define** statement is used for symbolic parameters. The syntax of the **define** statement is

define *name value*

where *name* is a symbolic name that represents the quantity of *value*. For example:

```
define ROWS 100
define CLOS 50
dimension a(ROWS), b(ROWS, COLS)
    if (i > ROWS | j > COLS) ...
```

causes the preprocessor to replace the name *ROWS* with the value *100* and the name *COLS* with the value *50*. Alternately, definitions may be written as

```
define(ROWS, 100)
```

in which case the defining text is everything after the comma up to the right parenthesis. This allows multiple-line definitions.

8.13 THE "include" STATEMENT

The Ratfor language provides an **include** statement similar to the **#include** <...> statement in C language. The syntax for this statement is

include *file*

which inserts the contents of the named file into the Ratfor input file in place of the **include** statement. The standard usage is to place **COMMON** blocks on a file and use the **include** statement to include the common code whenever needed.

8.14 FREE-FORM INPUT

In Ratfor, statements can be placed anywhere on a line. Long statements are continued automatically as are long conditions in **if**, **for**, and **until** statements. Blank lines are ignored. Multiple statements may appear on one line if they are separated by semicolons. No semicolon is needed at the end of a line if Ratfor can make some reasonable guess about whether the statement ends there. Lines ending with any of the characters

= + - * , | & (_

are assumed to be continued on the next line. Underscores are discarded wherever they occur. All other characters remain as part of the statement.

Any statement that begins with an all-numeric field is assumed to be a Fortran label and placed in columns 1 through 5 upon output. Thus:

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

is converted into

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

8.15 TRANSLATIONS

When the **−h** option is chosen, text enclosed in matching single or double quotes is converted to *n*H... but is otherwise unaltered (except for formatting — it may get split across card boundaries during the reformatting process). Within quoted strings, the backslash (\) serves as an escape character; i.e., the next character is taken literally. This provides a way to get quotes and the backslash itself into quoted strings. For example:

```
"\'"
```

is a string containing a backslash and an apostrophe. (This is not the standard convention of doubled quotes, but it is easier to use and more general.)

Any line that begins with the character % is left absolutely unaltered except for stripping off the % and moving the line one position to the left. This is useful for inserting control cards and other things that should not be preprocessed (like an existing Fortran program). Use % only for ordinary statements not for the condition parts of **if**, **while**, etc., or the output may come out in an unexpected place.

The following character translations are made (except within single or double quotes or on a line beginning with a %):

= =	**.eq.**
! =	**.ne.**
>	**.gt.**
> =	**.ge.**
<	**.lt.**
< =	**.le.**
&	**.and.**

|	**.or.**
!	**.not.**

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

[	{
]	}
$(	{
$)	}

8.16 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 reserved. 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.

8.17 EXAMPLE OF RATFOR CONVERSION

As an example of how to use the Ratfor program, the following program **prog.r** (where the **.r** indicates a Ratfor source program), is written in the Ratfor language:

```
   ICNT = 0
10 WRITE(6,31)
31 FORMAT("INPUT FIRST NUMBER")
   READ(5,32) A
32 FORMAT(F10.2)
   WRITE(6,33)
33 FORMAT("INPUT SECOND NUMBER")
   READ(5,34) B
34 FORMAT(F10.2)
   IF(A < B)
     WRITE(6,36) A,B
   ELSE WRITE(6,37)A,B
36 FORMAT(F10.2," < ",F10.2)
37 FORMAT(F10.2," >= ",F10.2)
   ICNT = ICNT + 1
   IF(ICNT.EQ.5)
     GOTO 100
   GOTO 10
00 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.

Contents

Chapter 9

USING VU-PASCAL

9.1 THE PASCAL COMPILER

The Pascal compiler offers two alternatives for producing programs: "EM-1" code, executed interpretively, and true PDP-11 code which is executed directly. For simplicity's sake, these modes will be called "interpreted" and "compiled," respectively, even though both modes involve some compilation.

The interpreted mode is good for developing software, since it offers quick compilation and several useful debugging features. Once a program is fully debugged, it can be compiled to PDP-11 code for fast execution. Compiled code runs about seven times faster than interpreted code, but producing it takes somewhat longer.

The simplest way to compile a Pascal program is to type:

pc *prog*.**p**

The source file name must end with **.p**. This produces an EM-1 code module called **e.out** which can be interpreted by typing:

em1

To prevent this module from being overwritten by your next compile, you should rename it with **mv**(1) to a different name. Or you could have used the **−o** flag to call it something different right from the start, as in

pc −o *prog*.**em1** *prog*.**p**

and then run it with

em1 *prog*.**em1**

Since the object module is not called **e.out**, you have to specify its name. The extension ".em1" is not mandatory, but it may be useful to remind you that this file is EM-1 code, or to distinguish it from a compiled version (described below).

To produce a compiled version, you use the **−C** flag, as in

pc −C *prog*.**p**

(Note that the **C** is capitalized.) This produces an executable module called **a.out**. Since this module is directly executable, it can be run by simply typing

a.out

The **−o** flag can again be used to give this module a safer name.

When interpreting, debugging information can be collected during run-time. For example, the following program **square** produces a table of squares.

```
program square(output);
const start = 1; stop = 16;
var num : start..stop;
begin
        for num := start to stop do
        begin   write(num,num*num);
                if num mod 4 = 0 then
                        writeln
                else write('        ');
        end;
end.
```

The command

pc square.p

is done to produce the interpreted **e.out** module. It can be run by typing

em1

to produce

```
 1    1     2    4     3    9     4    16
 5   25     6   36     7   49     8    64
 9   81    10  100    11  121    12   144
13  169    14  196    15  225    16   256
```

To check the number of types each line was executed, we can turn on debugging by typing

em1 −d

This will give us the same output as before, but collect certain information in various files that we can examine later with **edebug**. For example, typing

edebug −c square.p

gives the output

```
1       0       program square(output);
2       0       const start = 1; stop = 16;
3       0       var num : start..stop;
4       0       begin
5       1               for num := start to stop do
6       16              begin   write(num,num*num);
7       16                      if num mod 4 = 0 then
8       4                               writeln
9       12                      else write('       ');
10      0               end;
11      0       end.
```

Each line is number, and a count of the number of times the line was executed is given. The main loop beginning at line 6, for example, was executed 16 times; the if statement on line 7 evaluated true four times (line 8) and false 12 times (line 9).

Other debugging flags are possible: see **edebug**(1).

It is possible to divide Pascal sources into several modules and compile them separately, and link them together at a later time. If a program is spread into two files *prog1* and *prog2*, the command

pc **−c** *prog1*.**p** *prog2*.**p**

will produce intermediate object files *prog1*.**k** and *prog2*.**k**. These files may be linked to produce an **e.out** with

pc *prog1*.**k** *prog2*.**k**

or to produce an **a.out** with

pc **−C** *prog1*.**k** *prog2*.**k**

Compiling with the **−c** and **−O** flag produces an optimized intermediate module with a **.m** instead of **.k** extension. These optimized modules may be combined just as the **.k** modules were.

The **−l***xxx* flag should be used to include external library *xxx* found in **/lib** or **/usr/lib**. The two standard libraries are "libmon," for making VENIX system calls, and "libpc," for other external routines. (See the discussion of these later in this tutorial.) This flag must be placed at the end of the command line, as in

pc *prog*.**p** **−lpc**

The **−L** (capital l) flag can be used to create library modules with **.l** extensions. These are similar to optimized modules, but in a slightly different format allowing for faster linking. Unlike **.k** or **.m** modules, these may be archived with the VENIX archiver **ar**(1). Archives may be given on the command line to **pc** as well. Only files containing Pascal procedures should be made in library modules; the main program should not.

9.2 EXTENSIONS TO ISO PASCAL

VU-Pascal contains several extensions to standard ISO Pascal.

The compiler is able to (separately) compile a collection of declarations, procedures and functions to form a library. The library may be linked with the main program, compiled later. The syntax of these modules is

module = [constant-definition-part]
[type-definition-part]
[var-declaration-part]
[procedure-and-function-declaration-part]

The compiler accepts a program or a module:

unit = program | module

All variables declared outside a module must be imported by parameters, even the files input and output. Access to a variable declared in a module is only possible using the procedures and functions declared in that same module. By giving the correct procedure/function heading followed by the directive **extern** you may use procedures and functions declared in other units.

9.2.1 Compiler Options

The compiler accepts a number of different options which are turned on or off with a statement of the form

{**$***opt* ...}

where *opt* is the option letter followed by a + to turn it on, or a − to turn it off. These options are described in the next chapter, **VU-PASCAL REFERENCE MANUAL**. Two useful options are:

c+ Allows the use of null terminated strings surrounded by double quotes. A new type identifier **string** is predefined for this purpose. This is useful when using VENIX system calls which require null-terminated strings for file names.

d+ Allows the use of variables of type "long."

These options must appear before the **program** symbol to be effective.

9.3 EXTERNAL ROUTINES

9.3.1 External Routine Library

A library of external routines exists called "libpc." When you use these, you must include the flag **−lpc** at the end of the **pc** command line. A complete description may be found in **libpc**(3). Among the routines there are those allowing the manipulation of command line arguments and character strings:

function	**argc***:integer*
function	**argv***(i:integer):string;*
function	**environ***(i:integer):string;*
procedure	**argshift;**
function	**strbuf***(var b:buf):string;*
function	**strtobuf***(s:string; var b:buf; len:br1):br2;*
function	**strlen***(s:string):integer;*
function	**strfetch***(s:string; i:integer):char;*
procedure	**strstore***(s:string; i:integer; c:char);*

argc Gives the number of arguments provided when the program is called.

argv Selects the specified argument from the argument list and returns a pointer to it. This pointer is nil if the index is out of bounds (<0 or >=argc).

environ Returns a pointer to the i-th environment string (i>=0). Returns null if i is beyond the end of the environment list.

argshift Effectively deletes the first argument from the argument list. Its function is equivalent to 'shift' in the VENIX shell: argv[2] becomes argv[1], argv[3] becomes argv[2], etc. It is a useful procedure to skip optional flag arguments. Note that the matching of arguments and files is done at the time a file is opened by a call to reset or rewrite.

strbuf Type conversion from character array to string. It is your own responsibility that the string is zero terminated.

strtobuf Copy string into buffer until the string terminating zero byte is found or until the buffer if full, whatever comes first. The zero byte is also copied. The number of copied characters, excluding the zero byte, is returned. So if the result is equal to the buffer length, then the end of buffer is reached before the end of string.

strlen Returns the string length excluding the terminating zero byte.

strfetch Fetches the i-th character from a string. There is no check against the string length.

strstore Stores a character in a string. There is no check against string length, so this is a dangerous procedure.

The following program shows how these may be used. It is equivalent to the VENIX command **cat**(1). Notice that the "string" option is enabled with the {**$c+**} statement at the beginning. The functions and procedures from **libpc** are all followed with the **extern** directive, to indicate that they are in a different module.

```
{$c+}
program cat(input,inp,output);
var inp:text;
        s:string;

function argc:integer; extern;
function argv(i:integer):string; extern;
procedure argshift; extern;
function strlen(s:string):integer; extern;
function strfetch(s:string; i:integer):char; extern;
```

```
procedure copy(var fi:text);
var c:char;
begin reset(fi);
        while not eof(fi) do
                begin
                        while not eoln(fi) do
                                begin
                                        read(fi,c);
                                        write(c)
                                end;
                        readln(fi);
                        writeln
                end
end;

begin {main}
        if argc = 1 then
                copy(input)
        else
                repeat
        s := argv(1);
        if (strlen(s) = 1) and (strfetch(s,1) = '-')
                then copy(input)
        else copy(inp);
        argshift;
        until argc <= 1;
end.
```

9.3.2 Trap Handling

For trap handling, the following procedures exist. These routines allows you to handle all the possible error situations. You may define your own trap handler, written in Pascal, instead of the default handler that produces an error message and quits. You may also generate traps yourself.

procedure **trap***(err:integer);*
procedure **encaps***(procedure p; procedure q(n:integer));*

trap Trap generates the trap passed as argument (0..255). The trap numbers 128..255 may be used freely. The others are reserved.

encaps Encapsulate the execution of 'p' with the trap handler 'q'. Encaps replaces the previous trap handler by 'q', calls 'p' and restores the previous handler when 'p' returns. If, during the execution of 'p', a trap occurs, then 'q' is called with the trap number as parameter. For the duration of 'q' the previous trap handler is restored, so that you may handle only some of the errors in 'q'. All the other errors must then be raised again by a call to 'trap'.

The following program is an example of how trap handling can be used:

```
program bigreal(output);
const EFOVFL = 10;
var trapped:boolean;

procedure encaps(procedure p;
                 procedure q(n:integer)); extern;
procedure trap(n:integer); extern;

procedure traphandler(n:integer);
begin if n = EFOVFL then trapped: = true else trap(n) end;

procedure work;
var i,j:real;
begin trapped: = false; i: = 1;
        while not trapped do
                begin j: = i; i: = i*2 end;
        writeln('bigreal = ',j);
end;

begin
        encaps(work,traphandler);
end.
```

9.3.3 Using VENIX System Calls

The "libmon" library contains interface routines for VENIX system calls. When using this library, the flag **−lmon** must be given at the end of the **pc** command line.

The interface to these calls is quite similar to the standard "C" language interface. See **libmon(3)** for a description of the differences. The calling sequence for some of these calls can be a bit tricky for Pascal programs. Some hints may be useful:

- The c-option {$c+} allows you to declare null-terminated string constants in Pascal like **/etc/passwd**. Moreover, the identifier **string** is then defined as type identifier for a pointer to these zero-terminated strings.

- The d-option {$d+} allows you to use double precision integers (longs). The **lseek** system call, for instance, needs a long argument and returns a long result.

- If the system call requires a pointer as argument use a 'var' parameter. For instance declare **times** as:

  procedure times*(var t:timesbuf); extern;*

  Note that a 'string' is already a pointer.

- When defining types, use packed records if two bytes must be allocated in a single word, as in

```
device = packed record
           minor,major:0..255;
           end;
```

- If a collection of bits is needed, then define an enumerated type and a set of this enumerated type. The create mode of a file, for example, can be declared as:

```
modebits = (XHIM,WHIM,RHIM,
            XYOU,WYOU,RYOU,
            XME, WME, RME,
            TEXT,SGID,SUID,... );
creatmode = set of XHIM..SUID;
```

- Always declare a routine as function if it returns a value, like for

  function close*(fd:integer):integer; extern;*

- There are special system call routines **uread** and **uwrite** in **libpc**(3), because the names 'read' and 'write' are blocked by similar functions in Pascal.

Contents

Chapter 10

VU-PASCAL REFERENCE MANUAL

10.1 INTRODUCTION

This chapter refers to the (March 1980) ISO standard proposal for Pascal [1]. VU-Pascal complies with the requirements of this proposal almost completely. The standard requires an accompanying document describing the implementation-defined and implementation-dependent features, the reaction on errors and the extensions to standard Pascal. These four items will be addressed in the rest of this chapter, each in a separate section. The other chapters describe the deviations from the standard and the list of options recognized by the compiler.

The VU-Pascal compiler produces code for an EM-1 virtual machine as defined in [2]. The following implementations exist under VENIX:

- an interpreter running on a PDP-11. The interpreter performs some tests to detect undefined integers, integer overflow, range errors, etc.
- a translator into PDP-11 instructions.

These implementations will be referred to as "interpreted" and "compiled", respectively (although both in fact require some compilation.)

10.2 IMPLEMENTATION-DEFINED FEATURES

For each implementation-defined feature mentioned in the ISO standard we give the section number, the quotation from that section and the definition. First we quote the definition of implementation-defined:

> **Those parts of the language which may differ between processors, but which will be defined for any particular processor.**

6.1.7 Each string-character shall denote an implementation-defined value of char-type.

All 7-bits ASCII characters except linefeed LF (10) are allowed. Note that an apostrophe ' must be doubled within a string.

6.4.2.2 The values of type real shall be an implementation-defined subset of the real numbers denoted as specified by 6.1.5.

The format of reals is not defined in VU-Pascal. It is only defined that a real number occupies 2 words (32 bits) of storage, but this might change to 4 words in the future. The compiler can be instructed, by the f-option, to use a different size for real values. The following constants must be defined:

epbase: the base for the exponent part
epprec: the precision of the fraction
epemin: the minimum exponent
epemax: the maximum exponent

These constants must be chosen so that zero and all numbers with exponent e in the range

$$\mathbf{epemin} <= \mathbf{e} <= \mathbf{epemax}$$

and fraction-parts of the form

$$\mathbf{f} = \pm \mathbf{f}_1 . \mathbf{b}^{-1} + \ldots + \mathbf{f}_{\mathbf{epprec}} . \mathbf{b}^{-\mathbf{epprec}}$$

where

$$f_i = 0,\ldots,epbase-1 \text{ and } f_1 \neq 0$$

are possible values for reals. All other values of type real are considered illegal. (See [3] for more information about these constants).
For VU-Pascal these constants are:

epbase = 2

epprec = 24

epemin = −127

epemax = +127

ditto

6.4.2.2 The type char shall be the enumeration of a set of implementation-defined characters, some possibly without graphic representations.

The 7-bits ASCII character set is used, where LF (10) denotes the end-of-line marker on text-files.

6.4.2.2 The ordinal numbers of the character values shall be values of integer-type, that are implementation-defined, and that are determined by mapping the character values on to consecutive non-negative integer values starting at zero.

The normal ASCII ordering is used: ord('0') = 48, ord('A') = 65, ord('a') = 97, etc.

6.4.3.4 The largest and smallest values of integer-type permitted as numbers of a value of a set-type shall be implementation-defined.

The smallest value is 0. The largest value is default 15, but can be changed by using the i-option of the compiler up to a maximum of 32767. The compiler allocates as many bits for set-type variables as are necessary to store all possible values of the host-type of the base-type of the set, rounded up to the nearest multiple of 16. If 8 bits are sufficient then only 8 bits are used if part of a packed

structure. Thus, the variable s, declared by

var s: set of '0'..'9';

will contain 128 bits, not 10 or 16. These 128 bits are stored in 16 bytes, both for packed and unpacked sets. If the host-type of the base-type is integer, then the number of bits depends on the i-option. The programmer may specify how many bits to allocate for these sets. The default is 16, the maximum is 32767. The effective number of bits is rounded up to the next multiple of 16, or up to 8 if the number of bits is less than or equal to 8. Note that the use of set-constructors for sets with more than 256 elements is far less efficient than for smaller sets.

6.7.2.2 The predefined constant maxint shall be of integer-type and shall denote an implementation-defined value, that satisfies the following conditions:

(a) All integral values in the closed interval from −maxint to +maxint shall be values in the integer-type.

(b) Any monadic operation performed on an integer value in this interval shall be correctly performed according to the mathematical rules for integer arithmetic.

(c) Any dyadic integer operation on two integer values in this same interval shall be correctly performed according to the mathematical rules for integer arithmetic, provided that the result is also in this interval.

(d) Any relational operation on two integer values in this same interval shall be correctly performed according to the mathematical rules for integer arithmetic.

The representation of integers in VU-Pascal is a 16-bit word using two's complement arithmetic. Thus always:

maxint = 32767

Because the number −32768 may be used to indicate 'undefined', the range of available integers depends on the VU-Pascal implementation:

1. −32767..+32767.
2. −32768..+32767.

6.9.4.2 The default TotalWidth values for integer, Boolean and real types shall be implementation-defined. The defaults are:

integer 6
Boolean 5
real 13

6.9.4.5.1 ExpDigits, the number of digits written in an exponent part of a real, shall be implementation-defined.

ExpDigits is defined as

ceil(log10(log10(2 ** epemax)))

For the current implementations this evaluates to 2.

6.9.4.5.1 The character written as part of the representation of a real to indicate the beginning of the exponent part shall be implementation-defined, either 'E' or 'e'.

The exponent part starts with 'e'.

6.9.4.6 The case of the characters written as representation of the Boolean values shall be implementation-defined.

The representations of true and false are 'true' and 'false'.

6.9.6 The effect caused by the standard procedure page on a text file shall be implementation-defined.

The ASCII character form feed FF (12) is written.

6.10 The binding of the variables denoted by the program-parameters to entities external to the program shall be implementation-defined if the variable is of a file-type.

The program parameters must be files and all, except input and output, must be declared as such in the program block.

The program parameters input and output, if specified, will correspond with the VENIX streams 'standard input' and 'standard output'.

The other program parameters will be mapped to the argument strings provided by the caller of this program. The argument strings are supposed to be path names of the files to be opened or created. The order of the program parameters determines the mapping: the first parameter is mapped onto the first argument string, etc. Note that input and output are ignored in this mapping.

The mapping is recalculated each time a program parameter is opened for reading or writing by a call to the standard procedures reset or rewrite. This gives the programmer the opportunity to manipulate the list of string arguments using the external procedures argc, argv and argshift available in libpc [7].

6.10 The effect of an explicit use of reset or rewrite on the standard textfiles input or output shall be implementation-defined.

The procedures reset and rewrite are no-ops if applied to input or output.

10.3 IMPLEMENTATION-DEPENDENT FEATURES

For each implementation-dependent feature mentioned in the ISO standard draft, we give the section number, the quotation from that section and the way this feature is treated by the VU-Pascal system. First we quote the definition of 'implementation-dependent':

Those parts of the language which may differ between processors, and for which there need not be a definition for a particular processor.

5.1.1 The method for reporting errors or warnings shall be implementation-dependent.

The error handling is treated in a following chapter.

6.1.4 Other implementation-dependent directives may be defined.

Except for the required directive 'forward' the VU-Pascal compiler recognizes only one directive: 'extern'. This directive tells the compiler that the procedure block of this procedure will not be present in the current program. The code for the body of this procedure must be included at a later stage of the compilation process.

This feature allows one to build libraries containing often used routines. These routines do not have to be included in all the programs using them. Maintenance is much simpler if there is only one library module to be changed instead of many Pascal programs.

The use of external routines, however, is dangerous. The compiler normally checks for the correct number and type of parameters when a procedure is called and for the result type of functions. If an external routine is called these checks are not sufficient, because the compiler can not check whether the procedure heading of the external routine as given in the Pascal program matches the actual routine implementation. It should be the loader's task to check this. However, the current loaders are not that smart. Another solution is to check at run time, at least the number of words for parameters. This is not currently done, except when the debugging option is specified.

6.7.2.1 The order of evaluation of the operands of a dyadic operator shall be implementation-dependent.

Operands are always evaluated, so the program part

```
if (p<>nil) and (p^.value<>0) then
```

is probably incorrect.

The left-hand operand of a dyadic operator is almost always evaluated before the right-hand side. Some peculiar evaluations exist for the following cases:

1. the modulo operation is performed by a library routine to check for negative values of the right operand.

2. the expression

   set1 < = set2

   where set1 and set2 are compatible set types is evaluated in the following steps:

   evaluate set2

   evaluate set1

   compute set2 + set1

   test set2 and set2 + set1 for equality

   This is the only case where the right-hand side is computed first.

3. the expression

   set1 > = set2

   where set1 and set2 are compatible set types is evaluated in the following steps:

   - evaluate set1
   - evaluate set2
   - compute set1 + set2
   - test set1 and set1 + set2 for equality

6.7.3 The order of evaluation and binding of the actual-parameters for functions shall be implementation-dependent.

The order of evaluation and binding is from left to right.

6.8.2.2 If access to the variable in an assignment-statement involves the indexing of an array and/or a reference to a field within a variant of a record and/or the de-referencing of a pointer-variable and/or a reference to a buffer-variable, the decision whether these actions precede or follow the evaluation of the expression shall be implementation-dependent.

The expression is evaluated first.

6.8.2.3 The order of evaluation and binding of the actual-parameters for procedures shall be implementation-dependent.

The same as for functions.

6.9.6 The effect of inspecting a text file to which the page procedure was applied during generation is implementation-dependent.

The formfeed character written by page is treated like a normal character, with ordinal value 12.

6.10 The binding of the variables denoted by the program-parameters to entities external to the program shall be implementation-dependent unless the variable is of a file-type.

Only variables of a file-type are allowed as program parameters.

10.4 ERROR HANDLING

There are three classes of errors to be distinguished. In the first class are the error messages generated by the compiler. The second class consists of the occasional errors generated by the other programs involved in the compilation process. Errors of the third class are the errors as defined in the standard by:

> **An error is a violation by a program of the requirements of this standard such that detection normally requires execution of the program.**

10.4.1 Compiler Errors

The error messages (and the listing) are not generated by the compiler itself. The compiler only detects errors and writes the errors in condensed form on an intermediate file. Each error in condensed form contains:

- an optional error message parameter (identifier or number)
- an error number
- a line number
- a column number

Every time the compiler detects an error that does not have influence on the code produced by the compiler or on the syntax decisions, a warning message is given. If only warnings are generated, compilation proceeds and probably results in a correctly compiled program.

The intermediate error file is read by the interface program pc [4], that produces the error messages. It uses another file, the error message file, indexed by the error number, to find an error script line. Whenever this error script line contains the character '%', the error messages parameter is substituted. For negative error numbers the message constructed is prepended with 'Warning: '.

Sometimes the compiler produces several errors for the same file position (line number, column number). Only the first of these messages is given, because the others are probably directly caused by the first one. If the first one is a warning while one of its successors for that position is a fatal message, then the warning is promoted to a fatal one. However, parameterized messages are always given.

The error messages and listing come in three flavors, selected by flags given to pc [4]:

default: no listing, one line per error giving the file name of the Pascal source file, the line number and the error messages.

−e: for each erroneous line a listing of the line and its predecessor. The next line contains one or more characters '^' pointing to the places where an error is detected. For each error on that line a message follows.

−E: same as for '**−e**', except that all source lines are listed, even if the program is perfect.

10.4.2 Other Errors Detected at Compilation Time

Two main categories: file system problems and table overflow. Problems with the file system may be caused by protection (you may not read or create files) or by space problems (no space left on device; out of inodes; too many processes). Table overflow problems are often caused by peculiar source programs: very long procedures or functions, a lot of strings. Table overflow problems can sometimes be cured by giving a flag (**−I** or **−sl**) to pc [4].

Extensive treatment of these errors is outside the scope of this manual.

10.4.3 Runtime Errors

Errors detected at run time cause an error message to be generated on the diagnostic output stream (VENIX file descriptor 2). The message consists of the name of the program followed by a message describing the error, possibly followed by the source line number. Unless the l-option is turned off, the compiler generates code to keep track of which source line causes which instructions to be generated. It depends on the VU-Pascal implementation whether these instructions are skipped or executed:

1. These instructions are always executed. The old line number is saved and restored whenever a procedure or function is called. All error messages contain this line number, except when the l-option was turned off.
2. same as above, but line numbers are not saved when procedures and functions are called.

For each error mentioned in the standard we give the section number, the quotation from that section and the way it is processed by the VU-Pascal system.

For detected errors the corresponding message and trap number are given. Trap numbers are useful for exception-handling routines. Normally, each error causes the program to terminate. By using exception-handling routines one can ignore errors or perform alternate actions. Only some of the errors can be ignored by restarting the failing instruction. These errors are marked as non-fatal, all others as fatal. A list of errors with trap number between 0 and 63 (VU-Pascal errors) can be found in the *Programmer Reference Manual*. Errors with trap number between 64 and 127 (Pascal errors) are also listed in there.

6.4.3.3 It shall be an error if any field-identifier defined within a variant is used in a field-designator unless the value of the tag-field is associated with that variant.

This error is not detected. Sometimes this feature is used to achieve easy type conversion. However, using record variants this way is dangerous, error prone and not portable.

6.4.6 It shall be an error if a value of type T2 must be assignment-compatible with type T1, while T1 and T2 are compatible ordinal-types and the value of type T2 is not in the closed interval specified by T1.

The compiler distinguishes between array-index expressions and the other places where assignment-compatibility is required.

Array subscripting is done using array instructions which have three arguments: the array base address, the index and the address of the array descriptor. An array descriptor describes one dimension by three values: the element size, the lower bound on the index and the number of elements minus one. It depends on the VU-Pascal implementation whether these bounds are checked:

1. checked (array bound error, trap 6, non-fatal).

2. not checked.

The other places where assignment-compatibility is required are:

- assignment
- value parameters
- procedures read and readln
- the final value of the for-statement

For these places the compiler generates a range check instruction, except when the r-option is turned off, or when the range of values of T2 is enclosed in the range of T1. If the expression consists of a single variable and if that variable is of a subrange type, then the subrange type itself is taken as T2, not its host-type. Therefore, a range instruction is only generated if T1 is a subrange type and if the expression is a constant, an expression with two or more operands, or a single variable with a type not enclosed in T1. If a constant is assigned, then the VU-Pascal optimizer removes the range check instruction, except when the value is out of bounds.

It depends on the VU-Pascal implementation whether the range check instruction is executed or skipped:

1. checked (range bound error, trap 7, non-fatal).
2. skipped

6.4.6 It shall be an error if a value of type T2 must be assignment-compatible with type T1, while T1 and T2 are compatible set-types and any member of the value of type T2 is not in the closed interval specified by the base-type of the type T1.

This error is not detected.

6.5.4 It shall be an error if the pointer-variable has a nil-value or is undefined at the time it is de-referenced.

The VU-Pascal definition does not specify the binary representation of pointer values, so that it is not possible to choose an otherwise illegal

binary representation for the pointer value NIL. Rather arbitrary the compiler uses the integer value zero to represent NIL. For all current implementations this does not cause problems.

The VU-Pascal definition does specify the size of pointer objects: 2 bytes. The compiler can be instructed, by the p-option, to use a different size for pointer objects. NIL is represented here by the appropriate number of zero words.

It depends on the VU-Pascal implementation whether de-referencing of a pointer with value NIL causes an error:

1. for every de-reference the pointer value is checked to be legal. The value NIL is always illegal. Objects addressed by a NIL pointer always cause an error, except when they are part of some extraordinary sized structure (bad pointer, trap 26, fatal).

2. de-referencing for a fetch operation will not cause an error. A store operation probably causes an error if the '**−n**' flag is specified to pc [4] or ld [5] while loading your program.

VU-Pascal may initialize all memory cells for newly created variables with a constant that probably causes an error if that variable is not initialized with a value of its own type before use. For each implementation, we must specify whether memory cells are initialized, with what value, and whether this value causes an error if de-referenced.

1. each memory word is initialized with the bit representation 1000000000000000, representing −32768 in 2's complement notation. For most small and medium sized programs this value will cause a segmentation violation (memory fault, trap 25, fatal).

2. no initialization. Whenever a pointer is de-referenced, without being properly initialized, a segmentation violation (memory fault, trap 25, fatal) or 'bus error' are possible.

6.5.5 It shall be an error if the value of a file-variable f is altered while the buffer-variable is an actual variable parameter, or an element of the record-variable-list of a with-statement, or both.

This error is not detected

6.5.5 It shall be an error if the value of a file-variable f is altered by an assignment-statement which contains the buffer-variable f^ in its left-hand side.

This error is not detected.

6.6.5.2 It shall be an error if the stated pre-assertion does not hold immediately prior to any use of the file handling procedures rewrite, put, reset and get.

For each of these four operations the pre-assertions can be reformulated as:

rewrite(f): *no pre-assertion.*
put(f): *f is opened for writing and f^ is not undefined.*
reset(f): *f exists.*
get(f): *f is opened for reading and eof(f) is false.*

The following errors are detected for these operations:

rewrite(f): more args expected, trap 64, fatal:
f is a program-parameter and the corresponding file name is not supplied by the caller of the program.

rewrite error, trap 101, fatal:
the caller of the program lacks the necessary access rights to create the file in the file system or operating system problems like table overflow prevent creation of the file.

put(f): file not yet open, trap 72, fatal:
reset or rewrite are never applied to the file. The checks performed by the run time system are not full-proof.

not writable, trap 96, fatal:
f is opened for reading.

write error, trap 104, fatal:
probably caused by file system problems. For instance, the file storage is exhausted. Because I/O is buffered to improve performance, it might happen that this error occurs if the file is closed. Files are closed whenever they are rewritten or reset, or on program termination.

reset(f): more args expected, trap 64, fatal:
same as for rewrite(f).

reset error, trap 100, fatal:
f does not exist, or the caller has insufficient access rights, or operating system tables are exhausted.

get(f): file not yet open, trap 72, fatal:
as for put(f).

not readable, trap 97, fatal:
f is opened for writing.

end of file, trap 98, fatal:
eof(f) is true just before the call to get(f).

read error, trap 103, fatal:
unlikely to happen. Probably caused by hardware problems or by errors elsewhere in your program that destroyed the file information maintained by the run time system.

truncated, trap 99, fatal:
the file is not properly formed by an integer number of

file elements. For instance, the size of a file of integer is odd.

non-ASCII char read, trap 106, non-fatal:
the character value of the next character-type file element is out of range (0..127). Only for text files.

6.6.5.3 It shall be an error to change any variant-part of a variable allocated by the form new(p,c1,...,cn) from the variant specified.

This error is not detected.

6.6.5.3 It shall be an error if a variable to be disposed had been allocated using the form new(p,c1,...,cn) with more variants specified than specified to dispose.

This error is not detected.

6.6.5.3 It shall be an error if the variants of a variable to be disposed are different from those specified by the case-constants to dispose.

This error is not detected.

6.6.5.3 It shall be an error if the value of the pointer parameter of dispose has nil-value or is undefined.

The same comments apply as for de-referencing NIL or undefined pointers.

6.6.5.3 It shall be an error if a variable that is identified by the pointer parameter of dispose (or a component thereof) is currently either an actual variable parameter, or an element of the record-variable-list of a with-statement, or both.

This error is not detected.

6.6.5.3 It shall be an error if a referenced-variable created using the second form of new is used in its entirety as an operand in an expression, or as the variable in an assignment-statement or as an actual-parameter.

This error is not detected.

6.6.6.2 It shall be an error if the mathematical defined result of an arithmetic function would fall outside the set of values of the indicated result.

Except for the errors for undefined arguments, the following errors may occur for the arithmetic functions:

abs(x): none.

sqr(x): real underflow, trap 11, non-fatal;
real overflow, trap 10, non-fatal

sin(x): real underflow, trap 11, non-fatal

cos(x): real underflow, trap 11, non-fatal

exp(x): error in exp, trap 65, non-fatal (if x>10000);
real underflow, trap 11, non-fatal;
real overflow, trap 10, non-fatal

ln(x): error in ln, trap 66, non-fatal (if x<=0)

sqrt(x): error in sqrt, trap 67, non-fatal (if x<0)

arctan(x): real underflow, trap 11, non-fatal;
real overflow, trap 10, non-fatal

6.6.6.2 It shall be an error if x in ln(x) is not greater than zero.

See above.

6.6.6.2 It shall be an error if x in sqrt(x) is negative.

See above.

6.6.6.2 It shall be an error if the integer value of trunc(x) does not exist.

This error is detected (real—>int error, trap 17, non-fatal).

6.6.6.2 It shall be an error if the integer value of round(x) does not exist.

This error is detected (real—>int error, trap 17, non-fatal).

6.6.6.2 It shall be an error if the integer value of ord(x) does not exist.

This error can not occur, because the compiler will not allow such ordinal types.

6.6.6.2 It shall be an error if the character value of chr(x) does not exist.

Except when the r-option is turned off, the compiler generates a range check instruction. The effect of this instruction depends on the VU-Pascal implementation as described before.

6.6.6.2 It shall be an error if the value of succ(x) does not exist.

Same comments as for chr(x).

6.6.6.2 It shall be an error if the value of pred(x) does not exist.

Same comments as for chr(x).

6.6.6.5 It shall be an error if f in eof(f) is undefined.

This error is detected (file not yet open, trap 72, fatal).

6.6.6.5 It shall be an error if f in eoln(f) is undefined, or if eof(f) is true at that time.

The following errors may occur:

file not yet open, trap 72, fatal;
not readable, trap 97, fatal;
end of file, trap 98, fatal.

6.7.1 It shall be an error if any variable or function used as an operand in an expression is undefined at the time of its use.

Detection of undefined operands is only possible if there is at least one bit representation that is not allowed as legal value. The set of legal values depends on the type of the operand. To detect undefined operands, all newly created variables must be assigned a value illegal for the type of the created variable. The compiler itself does not generate code to initialize newly created variables. Instead, the compiler generates code to allocate some new memory cells. It is up to the VU-Pascal implementation to initialize these memory cells. However, the EM-1 virtual machine does not know the types of the variables for which memory cells are allocated. Therefore, the best VU-Pascal can do is to initialize with a value that is illegal for the most common types of operands.

For all current VU-Pascal implementations we will describe whether memory cells are initialized, which value is used to initialize, for each operand type whether that value is illegal, and for all operations on all operand types whether that value is detected as undefined.

1. new memory words are initialized with −32768. Assignment of this value is always allowed. Errors may occur whenever undefined operands are used in operations.

integer: −32768 is illegal. All arithmetic operations (except unary +) cause an error (undefined integer, trap 14, non-fatal). Relational operations do not, except for IN when the left operand is undefined. Printing of −32768 using write is allowed.

real: the bit representation of a real, caused by initializing the constituent memory words with −32768, is illegal. All arithmetic and relational operations (except unary +) cause an error (real undefined, trap 16, non-fatal). Printing causes the same error.

char: the value −32768 is illegal. For objects of type 'packed array[] of char' half the characters will have the value chr(0), which is legal, and the others will have the value chr(128), outside the valid ASCII range. The relational operators, however, do not cause an error.

Boolean: the value −32768 is illegal. For objects of type 'packed array[] of boolean' half the booleans will have the value false, while the others have the value v, where ord(v) = 128, naturally illegal. However, the Boolean and relational operations do not cause an error.

set: undefined operands of type set can not be distinguished from properly initialized ones. The set and relational operations, therefore, can never cause an error. However, if one forgets to initialize a set of character, then spurious characters like '/', '?', 'O', '_' and 'o' appear.

2. Newly created memory cells are not initialized and therefore they have a random value.

6.7.1 It shall be an error if the value of any member denoted by any member-designator of the set-constructor is outside the implementation-defined limits.

This error is detected (set bound error, trap 5, non-fatal).

6.7.1 It shall be an error if the possible types of a set-constructor do not permit it to assume a suitable type.

The compiler allocates as many bits as are necessary to store all elements of the host-type of the base-type of the set, not the base-type itself. Therefore, all possible errors can be detected at compile time.

6.7.2.2 It shall be an error if j is zero in 'i div j'.

It depends on the VU-Pascal implementation whether this error is detected:

1. detected (divide by 0, trap 12, non-fatal).

2. not detected.

6.7.2.2 It shall be an error if j is zero or negative in i MOD j.

This error is detected (only positive j in 'i mod j', trap 71, non-fatal).

6.7.2.2 It shall be an error if the result of any operation on integer operands is not performed according to the mathematical rules for integer arithmetic.

The reaction depends on the VU-Pascal implementation:

1. error detected if

(result > = 32768) or (result < −32768).

(integer overflow, trap 8, non-fatal). Note that if the result is

−32768 the use of this value in further operations may cause an error.

2. not detected.

6.8.3.5 It shall be an error if none of the case-constants is equal to the value of the case-index upon entry to the case-statement.

This error is detected (case error, trap 4, fatal).

6.8.3.9 It shall be an error if the final-value of a for-statement is not assignment-compatible with the control-variable when the initial-value is assigned to the control-variable.

It is detected if the control variable leaves its allowed range of values while stepping from initial to final value. This is equivalent with the requirements if the for-statement is not terminated before the final value is reached.

6.9.2 It shall be an error if the sequence of characters read looking for an integer does not form a signed-integer as specified in 6.1.5.

This error is detected (digit expected, trap 105, non-fatal).

6.9.2 It shall be an error if the sequence of characters read looking for a real does not form a signed-number as specified in 6.1.5.

This error is detected (digit expected, trap 105, non-fatal).

6.9.2 It shall be an error if read is applied to f while f is undefined or not opened for reading.

This error is detected (see get(f)).

6.9.4 It shall be an error if write is applied to f while f is undefined or not opened for writing.

This error is detected (see put(f)).

6.9.4 It shall be an error if TotalWidth or FracDigits as specified in write or writeln procedure calls are less than one.

This error is not detected. Moreover, it is considered an extension to allow zero or negative values.

6.9.6 It shall be an error if page is applied to f while f is undefined or not opened for writing.

This error is detected (see put(f)).

10.5 EXTENSIONS TO THE STANDARD

1. Separate compilation.

The compiler is able to (separately) compile a collection of declarations, procedures and functions to form a library. The library may be linked with the main program, compiled later. The syntax of these modules is

module = [constant-definition-part]
[type-definition-part]
var-declaration-part]
rocedure-and-function-declaration-part]

The compiler accepts a program or a module:

unit = program | module

All variables declared outside a module must be imported by parameters, even the files input and output. Access to a variable declared in a module is only possible using the procedures and functions declared in that same module. By giving the correct procedure/function heading followed by the directive 'extern' you may use procedures and functions declared in other units.

2. Assertions.

The VU-Pascal compiler recognizes an additional statement, the assertion. Assertions can be used as an aid in debugging and documentation.

The syntax is:

assertion = 'assert' Boolean-expression

An assertion is a simple-statement, so

simple-statement = [assignment-statement |
procedure-statement |
goto-statement |
assertion

An assertion causes an error if the Boolean-expression is false. That is its only purpose. It does not change any of the variables, at least it should not. Therefore, do not use functions with side-effects in the Boolean-expression. If the a-option is turned off, then assertions are skipped by the compiler. 'assert' is not a word-symbol (keyword) and may be used as identifier. However, assignment to a variable and calling of a procedure with that name will be impossible.

3. Additional procedures.

Three additional standard procedures are available:

halt: a call of this procedure is equivalent to jumping to the end of your program. It is always the last statement executed. The exit status of the program may be supplied as optional argument.

release:

mark: for most applications it is sufficient to use the heap as second stack. Mark and release are suited for this type of use, more suited than dispose. mark(p), with p of type pointer, stores the current value of the heap pointer in p. release(p), with p initialized by a call of mark(p), restores the heap pointer to its old value. All the heap objects, created by calls of new between the call of mark and the call of release, are removed and the space they used can be reallocated. Never use mark and release together with dispose!

4. VENIX interfacing.

   If the c-option is turned on, then some special features are available to simplify an interface with the VENIX environment. First of all, the compiler allows you to use a different type of string constants. These string constants are delimited by double quotes (‘″’). To put a double quote into these strings, you must repeat the double quote, like the single quote in normal string constants. These special string constants are terminated by a zero byte (chr(0)). The type of these constants is a pointer to a packed array of characters, with lower bound 1 and unknown upper bound.
   Secondly, the compiler predefines a new type identifier ‘string’ denoting this just described string type.

   The only thing you can do with these features is declaration of constants and variables of type ‘string’. String objects may not be allocated on the heap and string pointers may not be de-referenced. Still these strings are very useful in combination with external routines. The procedure write is extended to print these zero-terminated strings correctly.

5. Double length (32 bit) integers.

   If the d-option is turned on, then the additional type ‘long’ is known to the compiler. Long variables have integer values in the range −2147483647..+2147483647. Long constants may be declared. It is not allowed to form subranges of type long. All operations allowed on integers are also allowed on longs and are indicated by the same operators: ‘+’, ‘−’, ‘*’, ‘/’, ‘div’, ‘mod’. The procedures read and write have been extended to handle long arguments correctly. The default width for longs is 11. The standard procedures ‘abs’ and ‘sqr’ have been extended to work on long arguments. Conversion from integer to long, long to real, real to long and long to integer are automatic, like the conversion from integer to real. Two of these conversions may cause errors to occur:

   - real—>longint error, trap 18, non-fatal
   - longint—>int error, trap 19, non-fatal

   This last error is only detected in implementation 1, with ‘test on’. Note that all current implementations use target machine floating point instructions to perform some of the long operations.

6. Underscore as letter.

   The character '_' may be used in forming identifiers, if the u-option is turned on.

7. Zero field width in write.

   Zero or negative TotalWidth arguments to write are allowed. No characters are written for character, string or Boolean type arguments then. A zero or negative FracDigits argument for fixed-point representation of reals causes the fraction and the character '.' to be suppressed.

8. Alternate symbol representation.

   The comment delimiters '(*' and '*)' are recognized and treated like '{' and '}'. The other alternate representations of symbols are not recognized.

10.6 DEVIATIONS FROM THE STANDARD

VU-Pascal deviates from the (March 1980) standard proposal in the following ways:

1. Only the first 8 characters of identifiers are significant, as requested by all standard proposals prior to March 1980. In the latest proposal, however, the sentence

   > **"A conforming program should not have its meaning altered by the truncation of its identifiers to eight characters or the truncation of its labels to four digits."**

   is missing.

2. The character sequences 'procedur', 'procedur8', 'functionXyZ' etc. are all erroneously classified as the word-symbols 'procedure' and 'function'.

3. Standard procedures and functions are not allowed as parameters in VU-Pascal, conforming to all previous standard proposals. You can obtain the

same result with negligible loss of performance by declaring some user routines like:

```
function sine(x:real):real;
begin
   sine: = sin(x)
end;
```

4. The scope of identifiers and labels should start at the beginning of the block in which these identifiers or labels are declared. The VU-Pascal compiler, as most other one pass compilers, deviates in this respect, because the scope of variables and labels starts at their defining-point.

10.7 COMPILER OPTIONS

Some options of the compiler may be controlled by using "{$....}." Each option consists of a lower case letter followed by +, − or an unsigned number. Options are separated by commas. The following options exist:

a +/− this option switches assertions on and off. If this option is on, then code is included to test these assertions at run time. Default +.

c +/− this option, if on, allows you to use C-type string constants surrounded by double quotes. Moreover, a new type identifier 'string' is predefined. Default −.

d +/− this option, if on, allows you to use variables of type 'long'. Default −.

f <**num**> the size of reals can be changed by this option. <num> should be specified in 16 bit words. The current default is 2, but might change to 4 in the future.

i <**num**> with this flag the setsize for a set of integers can be manipulated. The number must be the number of bits per set. The default value is 16, just fitting in one word on the PDP and many other minis.

l +/− if + then code is inserted to keep track of the source line number. When this flag is switched on and off, an incorrect line number may appear if the error occurs in a part of your program for which this flag is off. These same line numbers are used for the profile, flow and count options of the EM-1 interpreter em1 [6]. Default +.

p <**num**> the size of pointers can be changed by this option. <num> should be specified in 16 bit words. Default 1.

r +/− if + then code is inserted to check subrange variables against lower and upper subrange limits. Default +.

s +/− if + then the compiler will hunt for places in your program where non-standard features are used, and for each place found it will generate a warning. Default −.

t +/− if + then each time a procedure is entered, the routine 'procentry' is called. The compiler checks this flag just before the first symbol that follows the first 'begin' of the body of the procedure. Also, when the procedure exits, then the procedure 'procexit' is called if the t flag is on just before the last 'end' of the procedure body. Both 'procentry' and 'procexit' have a packed array of 8 characters as a parameter. Default procedures are present in the run time library. Default −.

u +/− if + then the character '_' is treated like a lower case letter, so that it may be used in identifiers. Procedure and function identifiers starting with an underscore may cause problems, because they may collide with library routine names. Default −.

Seven of these flags (c, d, f, i, p, s and u) are only effective when they appear before the 'program' symbol. The others may be switched on and off.

A second method of passing options to the compiler is available. This method uses the file on which the compact EM-1 code will be written. The compiler starts reading from this file scanning for options in the same format as used normally, except for the comment delimiters and the dollar sign. All options found on the file override the options set in your program. Note that the compact code file must always exist before the compiler is called.

The user interface program pc[4] takes care of creating this file normally and also writes one of its options onto this file. The user can specify, for instance, without changing any character in its Pascal program, that the compiler must include code for procedure/function tracing.

Another very powerful debugging tool is the knowledge that inaccessible statements and useless tests are removed by the VU-Pascal optimizer. For instance, a statement like:

```
if debug then
  writeln('initialization done');
```

is completely removed by the optimizer if debug is a constant with value false. The first line is removed if debug is a constant with value true. Of course, if debug is a variable nothing can be removed.

A disadvantage of Pascal, the lack of preinitialized data, can be diminished by making use of the possibilities of the VU-Pascal optimizer. For instance, initializing an array of reserved words is sometimes optimized into 3 EM-1 virtual machine instructions. To maximize this effect you must initialize variables as much as possible in order of declaration and array entries in order of increasing index.

10.8 REFERENCES

[1] ISO standard proposal ISO/TC97/SC5-N462, dated February 1979. The same proposal, in slightly modified form, can be found in: A.M.Addyman e.a., "A draft description of Pascal," Software, practice and experience, May 1979. An improved version, received March 1980, is followed as much as possible for the current VU-Pascal.

[2] A.S.Tanenbaum, J.W.Stevenson, Hans van Staveren, "Description of an experimental machine architecture for use of block structured languages," Informatica rapport IR-54.

[3] W.S.Brown, S.I.Feldman, "Environment parameters and basic functions for floating-point computation," Bell Laboratories CSTR #72.

[4] User Reference Manual pc(1).

[5] User Reference Manual ld(1).

[6] User Reference Manual em1(1).

[7] Programmer Reference Manual libpc(3).

[8] Programmer Reference Manual pc__emlib(3).

Contents

Chapter 11

SCREEN PACKAGE

11.1 Overview

In making available the generalized terminal descriptions in **/etc/termcap**, much information was made available to the programmer, but little work was taken out of one's hands. The purpose of this package is to allow the C programmer to do the most common type of terminal dependent functions, namely movement optimization and optimal screen updating, without doing any of the dirty work and (hopefully) with as much ease as is possible to simply print or read things.

The package is split into three parts: (1) Screen updating; (2) Screen updating with user input; and (3) Cursor motion optimization.

It is possible to use the motion optimization without using either of the other two, and screen updating and input can be done without any programmer knowledge of the motion optimization, or indeed the **/etc/termcap** data base itself.

11.1.1 Terminology

In this document, the following terminology is used:

window An internal representation containing an image of what a section of the terminal screen may look like at some point. This subsection can either encompass the entire terminal screen, or any smaller portion down to a single character within that screen.

terminal Sometimes called **terminal screen**. The package's idea of what the terminal's screen currently looks like, i.e., what the user sees now. This is a special *screen*.

screen This is a subset of windows that are as large as the terminal screen, i.e., they start at the upper left hand corner and encompass the lower right hand corner. One of these, *stdscr*, is automatically provided for the programmer.

11.1.2 Compiling

In order to use the library, it is necessary to have certain types and variables defined. Therefore, the programmer must have a line:

#include <curses.h>

at the top of the program source. The header file **<curses.h>** includes **<sgtty.h>***. Also, compilations should have the following form:

cc [*flags*] file ... −lcurses −ltermlib

11.1.3 Screen Updating

In order to update the screen optimally, it is necessary for the routines to know what the screen currently looks like and what the programmer wants it to look like next. For this purpose, a data type (structure) named *WINDOW* is defined which describes a window image to the routines, including its starting position on the screen (the (y, x) co-ordinates of the upper left hand corner) and its size. One of these (called *curscr* for *current screen*) is a screen image of what the terminal currently looks like. Another screen (called *stdscr*, for *standard screen*) is provided by default for making changes.

---

* The screen package also uses the Standard I/O library, so **<curses.h>** includes **<stdio.h>**. It is redundant (but harmless) for the programmer to do it, too.

A window is a purely internal representation. It is used to build and store a potential image of a portion of the terminal. It doesn't bear any necessary relation to what is really on the terminal screen. It is more like an array of characters on which to make changes.

When one has a window that describes what some part the terminal should look like, the routine *refresh*() (or *wrefresh*() if the window is not *stdscr*) is called. *refresh*() makes the terminal, in the area covered by the window, look like that window. Note, therefore, that changing something on a window *does* **not** *change the terminal*. Actual updates to the terminal screen are made only by calling *refresh*() or *wrefresh*(). This allows the programmer to maintain several different ideas of what a portion of the terminal screen should look like. Also, changes can be made to windows in any order, without regard to motion efficiency. Then, at will, the programmer can effectively say "make it look like this," and let the package worry about the best way to update the screen.

11.1.4 Naming Conventions

As hinted above, the routines can use several windows, but two are automatically given: *curscr*, which knows what the terminal looks like, and *stdscr*, which is what the programmer wants the terminal to look like next. The user should never really access *curscr* directly. Changes should be made to the appropriate screen, and then the routine *refresh*() (or *wrefresh*()) should be called.

Many functions are set up to deal with *stdscr* as a default screen. For example, to add a character to *stdscr*, one calls *addch*() with the desired character. If a different window is to be used, the routine *waddch()* (for window-specific *addch*()) is provided*. This convention of prepending function names with a **'w'** when they are to be applied to specific windows is consistent. The only routines which do *not* do this are those to which a window must always be specified.

---

* Actually, *addch*() is really a "#define" macro with arguments, as are most of the "functions" which deal with *stdscr* as a default.

In order to move the current (y, x) co-ordinates from one point to another, the routines *move*() and *wmove*() are provided. However, it is often desirable to first move and then perform some I/O operation. In order to avoid clumsiness, most I/O routines can be preceded by the prefix '**mv**' and the desired (y, x) co-ordinates then can be added to the arguments to the function. For example, the calls

```
move(y, x);
addch(ch);
```

can be replaced by

```
mvaddch(y, x, ch);
```

and

```
wmove(win, y, x);
waddch(win, ch);
```

can be replaced by

```
mvwaddch(win, y, x, ch);
```

Note that the window description pointer (*win*) comes before the added (y, x) co-ordinates. If such pointers are need, they are always the first parameters passed.

11.2 Variables

Many variables which are used to describe the terminal environment are available to the programmer. They are:

Type	Name	Description
WINDOW *	curscr	current version of the screen (terminal screen).
WINDOW *	stdscr	standard screen. Most updates are usually done here.
char *	Def_term	default terminal type if type cannot be determined
bool	My_term	use the terminal specification in *Def_term* as terminal, irrelevant of real terminal type
char *	ttytype	full name of the current terminal.
int	LINES	number of lines on the terminal

int	COLS	number of columns on the terminal
int	ERR	error flag returned by routines on a fail.
int	OK	error flag returned by routines when things go right.

There are also several "#define" constants and types which are of general usefulness:

reg	**storage class "register" (e.g., *reg int i;*)**
bool	**boolean type, actually a "char" (e.g., *bool doneit;*)**
TRUE	**boolean "true" flag (1).**
FALSE	**boolean "false" flag (0).**

11.3 Usage

This is a description of how to actually use the screen package. In it, we assume all updating, reading, etc. is applied to *stdscr*. All instructions will work on any window, with changing the function name and parameters as mentioned above.

11.3.1 Starting up

In order to use the screen package, the routines must know about terminal characteristics, and the space for *curscr* and *stdscr* must be allocated. These functions are performed by *initscr*(). Since it must allocate space for the windows, it can overflow core when attempting to do so. On this rather rare occasion, *initscr*() returns ERR. *initscr*() must **always** be called before any of the routines which affect windows are used. If it is not, the program will core dump as soon as either *curscr* or *stdscr* are referenced. However, it is usually best to wait to call it until after you are sure you will need it, like after checking for startup errors. Terminal status changing routines like *nl*() and *crmode*() should be called after *initscr*().

Now that the screen windows have been allocated, you can set them up for the run. If you want to, say, allow the window to scroll, use *scrollok*(). If you want the cursor to be left after the last change, use *leaveok*(). If this isn't done, *refresh*() will move the cursor to the window's current (y, x) co-ordinates after updating it. New windows of your own can be created, too, by using the

functions *newwin*() and *subwin*(). *delwin*() will allow you to get rid of old windows. If you wish to change the official size of the terminal by hand, just set the variables *LINES* and *COLS* to be what you want, and then call *initscr*(). This is best done before, but can be done either before or after, the first call to *initscr*(), as it will always delete any existing *stdscr*() and/or *curscr*() before creating new ones.

11.3.2 The Nitty-Gritty

11.3.2.1 Output

Now that we have set things up, we will want to actually update the terminal. The basic functions used to change what will go on a window are *addch*() and *move*(). *addch*() adds a character at the current (y, x) co-ordinates, returning ERR if it would cause the window to illegally scroll, i.e., printing a character in the lower right-hand corner of a terminal which automatically scrolls if scrolling is not allowed. *move*() changes the current (y, x) co-ordinates to whatever you want them to be. It returns ERR if you try to move off the window when scrolling is not allowed. As mentioned above, you can combine the two into *mvaddch*() to do both things in one fell swoop.

The other output functions, such as *addstr*() and *printw*(), all call *addch*() to add characters to the window.

After you have put on the window what you want there, when you want the portion of the terminal covered by the window to be made to look like it, you must call *refresh*(). In order to optimize finding changes, *refresh*() assumes that any part of the window not changed since the last *refresh*() of that window has not been changed on the terminal, i.e., that you have not refreshed a portion of the terminal with an overlapping window. If this is not the case, the routine *touchwin*() is provided to make it look like the entire window has been changed, thus making *refresh*() check the whole subsection of the terminal for changes.

If you call *wrefresh*() with *curscr*, it will make the screen look like *curscr* thinks it looks like. This is useful for implementing a command which would redraw the screen in case it get messed up.

11.3.2.2 Input

Input is essentially a mirror image of output. The complementary function to *addch*() is *getch*() which, if echo is set, will call *addch*() to echo the character. Since the screen package needs to know what is on the terminal at all times, if characters are to be echoed, the tty must be in raw or cbreak mode. If it is not, *getch*() sets it to be cbreak, and then reads in the character.

11.3.2.3 Miscellaneous

All sorts of fun functions exist for maintaining and changing information about the windows. For the most part, the descriptions in section 11.9 should suffice.

11.3.3 Finishing up

In order to do certain optimizations, and, on some terminals, to work at all, some things must be done before the screen routines start up. These functions are performed in *gettmode*() and *setterm*(), which are called by *initscr*(). In order to clean up after the routines, the routine *endwin*() is provided. It restores tty modes to what they were when *initscr*() was first called. Thus, anytime after the call to initscr, *endwin*() should be called before exiting.

11.4 Cursor Motion Optimization: Standing Alone

It is possible to use the cursor optimization functions of this screen package without the overhead and additional size of the screen updating functions. The screen updating functions are designed for uses where parts of the screen are changed, but the overall image remains the same. This includes such programs as **vi**. Certain other programs will find it difficult to use these functions in this manner without considerable unnecessary program overhead. For such applications, all that is needed is the motion optimizations. This, therefore, is a description of what goes on at the lower levels of this screen package. The descriptions assume a certain amount of familiarity with programming problems and some finer points of C. None of it is terribly difficult, but you should be forewarned.

11.4.1 Terminal Information

In order to use a terminal's features to the best of a program's abilities, it must first know what they are. The **/etc/termcap** data base describes these, but a certain amount of decoding is necessary, and there are, of course, both efficient and inefficient ways of reading them in. The algorithm that the screen package uses is taken from **vi** and is hideously efficient. It reads them in a tight loop into a set of variables whose names are two uppercase letters with some mnemonic value. For example, *HO* is a string which moves the cursor to the "home" position†. As there are two types of variables involving ttys, there are two routines. The first, *gettmode*(), sets some variables based upon the tty modes accessed by **gtty**(1) and **stty**(1). The second, *setterm*(), a larger task by reading in the descriptions from the **/etc/termcap** data base. This is the way these routines are used by *initscr*():

```
if (isatty(0)) {
        gettmode();
        if (sp = getenv("TERM"))
                setterm(sp);
}
else
        setterm(Def_term);
_puts(TI);
_puts(VS);
```

isatty() checks to see if file descriptor 0 is a terminal‡ If it is, *gettmode*() sets the terminal description modes from a **gtty**(1). *getenv*() is then called to get the name of the terminal, and that value (if there is one) is passed to *setterm*(), which reads in the variables from **/etc/termcap** associated with that terminal. *(getenv*() returns a pointer to a string containing the name of the terminal, which we save in the character pointer *sp*().) If *isatty*() returns false, the default terminal *Def_term*() is used. The *TI* and *VS* sequences initialize the

---

† These names are identical to those variables used in the **/etc/termcap** data base to describe each capability. See Appendix A for a complete list of those read, and **termcap**(5) for a full description.

‡ *isatty*() is defined in the default C library function routines. It does a **gtty**(1) on the descriptor and checks the return value.

terminal *__puts*() is a macro which uses *tputs*() (see **termcap**(3)) to put out a string). It is these things which *endwin*() undoes.

11.4.2 Movement Optimizations

Now that we have all this useful information, it would be nice to do something with it. The most difficult thing to do properly is motion optimization. When you consider how many different features various terminals have (tabs, backtabs, non-destructive space, home sequences, absolute tabs, ...) you can see that deciding how to get from here to there can be a decidedly non-trivial task. The editor **vi** uses many of these features, and the routines it uses to do this take up many pages of code.

After using *gettmode*() and *setterm*() to get the terminal descriptions, the function *mvcur*() deals with this task. Its usage is simple: you simply tell it where you are now and where you want to go. For example

mvcur(0, 0, LINES/2, COLS/2)

would move the cursor from the home position (0, 0) to the middle of the screen. If you wish to force absolute addressing, you can use the function *tgoto*() from the **termlib**(7) routines, or you can tell *mvcur*() that you are impossibly far away. For example, to absolutely address the lower left hand corner of the screen from anywhere just claim that you are in the upper right hand corner:

mvcur(0, COLS − 1, LINES − 1, 0)

11.5 The Functions

In the following definitions, "†" means that the "function" is really a "#define" macro with arguments. This means that it will not show up in stack traces in the debugger, or, in the case of such functions as *addch*(), it will show up as it's '**w**' counterpart. The arguments are given to show the order and type of each. Their names are not mandatory, just suggestive.

11.6 Output Functions

addch(ch)†
char ch;

waddch(win, ch)
WINDOW *win;
char ch;

Add the character *ch* on the window at the current (y, x) co-ordinates. If the character is a newline ('\n') the line will be cleared to the end, and the current (y, x) co-ordinates will be changed to the beginning off the next line if newline mapping is on, or to the next line at the same x co-ordinate if it is off. A return ('\r') will move to the beginning of the line on the window. Tabs ('\t') will be expanded into spaces in the normal tabstop positions of every eight characters. This returns ERR if it would cause the screen to scroll illegally.

addstr(str)†
char *str;

waddstr(win, str)
WINDOW *win;
char *str;

Add the string pointed to by *str* on the window at the current (y, x) co-ordinates. This returns ERR if it would cause the screen to scroll illegally. In this case, it will put on as much as it can.

box(win, vert, hor)
WINDOW *win;
char vert, hor;

Draws a box around the window using *vert* as the character for drawing the vertical sides, and *hor* for drawing the horizontal lines. If scrolling is not allowed, and the window encompasses the lower right-hand corner of the terminal, the corners are left blank to avoid a scroll.

clear()†

wclear(win)
WINDOW *win;

Resets the entire window to blanks. If *win* is a screen, this sets the clear flag, which will cause a clear-screen sequence to be sent on the next *refresh*() call. This also moves the current (y, x) co-ordinates to (0, 0).

clearok(scr, boolf)†
WINDOW *scr;
bool boolf;

Sets the clear flag for the screen *scr*. If *boolf* is TRUE, this will force a clear-screen to be printed on the next *refresh*(), or stop it from doing so if *boolf* is FALSE. This only works on screens, and, unlike *clear*(), does not alter the contents of the screen. If *scr* is *curscr*, the next *refresh*() call will cause a clear-screen, even if the window passed to *refresh*() is not a screen.

clrtobot()†

wclrtobot(win)
WINDOW *win;

Wipes the window clear from the current (y, x) co-ordinates to the bottom. This does not force a clear-screen sequence on the next refresh under any circumstances. This has no associated "**mv**" command.

clrtoeol()†

wclrtoeol(win)
WINDOW *win;

Wipes the window clear from the current (y, x) co-ordinates to the end of the line. This has no associated "**mv**" command.

delch()

wdelch(win)
WINDOW *win;

Delete the character at the current (y, x) co-ordinates. Each character after it on the line shifts to the left, and the last character becomes blank.

deleteln()

wdeleteln(win)
WINDOW *win;

Delete the current line. Every line below the current one will move up, and the bottom line will become blank. The current (y, x) co-ordinates will remain unchanged.

erase()†

werase(win)
WINDOW *win;

Erases the window to blanks without setting the clear flag. This is analogous to *clear*(), except that it never causes a clear-screen sequence to be generated on a *refresh*(). This has no associated "**mv**" command.

insch(c)
char c;

winsch(win, c)
WINDOW *win;
char c;

Insert *c* at the current (y, x) co-ordinates Each character after it shifts to the right, and the last character disappears. This returns ERR if it would cause the screen to scroll illegally.

insertln()

winsertln(win)
WINDOW *win;

Insert a line above the current one. Every line below the current line will be shifted down, and the bottom line will disappear. The current line will become blank, and the current (y, x) co-ordinates will remain unchanged. This returns ERR if it would cause the screen to scroll illegally.

move(y, x)†
int y, x;

wmove(win, y, x)
WINDOW *win;
int y, x;

Change the current (y, x) co-ordinates of the window to *(y,x)*. This returns ERR if it would cause the screen to scroll illegally.

overlay(win1, win2)
WINDOW *win1, *win2;

Overlay *win1* on *win2*. The contents of *win1*, insofar as they fit, are placed on *win2* at their starting (y, x) co-ordinates. This is done non-destructively, i.e., blanks on *win1* leave the contents of the space on *win2* untouched.

overwrite(win1, win2)
WINDOW *win1, *win2;

Overwrite *win1* on *win2*. The contents of *win1*, insofar as they fit, are placed on *win2* at their starting (y, x) co-ordinates. This is done destructively, i.e., blanks on *win1* become blank on *win2*.

printw(fmt, arg1, arg2, ...)
char *fmt;

wprintw(win, fmt, arg1, arg2, ...)
WINDOW *win;
char *fmt;

Performs a *printf*() on the window starting at the current (y, x) co-ordinates.

It uses *addstr*() to add the string on the window. It is often advisable to use the field width options of *printf*() to avoid leaving things on the window from earlier calls. This returns ERR if it would cause the screen to scroll illegally.

refresh()†

wrefresh(win)
WINDOW *win;

Synchronize the terminal screen with the desired window. If the window is not a screen, only that part covered by it is updated. This returns ERR if it would cause the screen to scroll illegally. In this case, it will update whatever it can without causing the scroll.

standout()†

wstandout(win)
WINDOW *win;

standend()†

wstandend(win)
WINDOW *win;

Start and stop putting characters onto *win* in standout mode. *standout*() causes any characters added to the window to be put in standout mode on the terminal (if it has that capability). *standend*() stops this. The sequences *SO* and *SE* (or *US* and *UE* if they are not defined) are used (see Appendix A).

11.7 Input Functions

crmode()†

nocrmode()†

Set or unset the terminal to/from cbreak mode.

echo()†

noecho()†

Sets the terminal to echo or not echo characters.

getch()†

wgetch(win)
WINDOW *win;

Gets a character from the terminal and (if necessary) echos it on the window. This returns ERR if it would cause the screen to scroll illegally. Otherwise, the character gotten is returned. If *noecho* has been set, then the window is left unaltered. In order to retain control of the terminal, it is necessary to have one of *noecho*, *cbreak*, or *rawmode* set. If you do not set one, whatever routine you call to read characters will set *cbreak* for you, and then reset to the original mode when finished.

getstr(str)†
char *str;

wgetstr(win, str)
WINDOW *win;
char *str;

Get a string through the window and put it in the location pointed to by *str*, which is assumed to be large enough to handle it. It sets tty modes if necessary, and then calls *getch*() (or *wgetch(win))* to get the characters needed to fill in the string until a newline or EOF is encountered. The newline stripped off the string. This returns ERR if it would cause the screen to scroll illegally.

raw()†

noraw()†

Set or unset the terminal to/from raw mode. On version 7 UNIX* this also turns of newline mapping (see *(nl))*.

---

* UNIX is a trademark of Bell Laboratories.

```
scanw(fmt, arg1, arg2, ...)
char    *fmt;

wscanw(win, fmt, arg1, arg2, ...)
WINDOW      *win;
char    *fmt;
```

Perform a *scanf*() through the window using *fmt*. It does this using consecutive *getch*()'s (or *wgetch(win)'s)* This returns ERR if it would cause the screen to scroll illegally.

11.8 Miscellaneous Functions

```
delwin(win)
WINDOW      *win;
```

Deletes the window from existence. All resources are freed for future use by **calloc**(3). If a window has a *subwin*() allocated window inside of it, deleting the outer window the subwindow is not affected, even though this does invalidate it. Therefore, subwindows should be deleted before their outer windows are.

```
endwin()
```

Finish up window routines before exit. This restores the terminal to the state it was before *initscr*() (or *gettmode*() and *setterm*()) was called. It should always be called before exiting. It does not exit. This is especially useful for resetting tty stats when trapping rubouts via **signal**(2).

```
getyx(win, y, x)†
WINDOW      *win;
int     y, x;
```

Puts the current (y, x) co-ordinates of *win* in the variables y and x. Since it is a macro, not a function, you do not pass the address of y and x.

inch()†

winch(win)†
WINDOW *win;

Returns the character at the current (y, x) co-ordinates on the given window. This does not make any changes to the window. This has no associated "**mv**" command.

initscr()

Initialize the screen routines. This must be called before any of the screen routines are used. It initializes the terminal-type data and such, and without it, none of the routines can operate. If standard input is not a tty, it sets the specifications to the terminal whose name is pointed to by *Def_term* (initially "dumb"). If the boolean *My_term* is true, *Def_term* is always used.

leaveok(win, boolf)†
WINDOW *win;
bool boolf;

Sets the boolean flag for leaving the cursor after the last change. If *boolf* is TRUE, the cursor will be left after the last update on the terminal, and the current (y, x) co-ordinates for *win* will be changed accordingly. If it is FALSE, it will be moved to the current (y, x) co-ordinates. This flag (initially FALSE) retains its value until changed by the user.

longname(termbuf, name)
char *termbuf, *name;

Fills in *name* with the long (full) name of the terminal described by the termcap entry in *termbuf*. It is generally of little use, but is nice for telling the user in a readable format what terminal we think he has. This is available in the global variable *ttytype*. *Termbuf* is usually set via the termlib routine *tgetent*().

mvwin(win, y, x)
WINDOW *win;
int y, x;

Move the home position of the window *win* from its current starting coordinates to *(y,* x). If that would put part or all of the window off the edge of the

terminal screen, *mvwin*() returns ERR and does not change anything.

WINDOW()†
newwin(lines, cols, begin_y, begin_x)
int lines, cols, begin_y, begin_x;

Create a new window with *lines* lines and *cols* columns starting at position *(begin_y, begin_x)*. If either *lines* or *cols* is 0 (zero), that dimension will be set to *(LINES − begin_y)* or *(COLS − begin_x)* respectively. Thus, to get a new window of dimensions *LINES* × *COLS*, use *newwin*(0, 0, 0, 0).

nl()†

nonl()†

Set or unset the terminal to/from nl mode, i.e., start/stop the system from mapping <**RETURN**> to <**LINE-FEED**>. If the mapping is not done, *refresh*() can do more optimization, so it is recommended, but not required, to turn it off.

scrollok(win, boolf)†
WINDOW *win;
bool boolf;

Set the scroll flag for the given window. If *boolf* is FALSE, scrolling is not allowed. This is its default setting.

touchwin(win)
WINDOW *win;

Make it appear that the every location on the window has been changed. This is usually only needed for refreshes with overlapping windows.

WINDOW()†
subwin(win, lines, cols, begin_y, begin_x)
WINDOW *win;
int lines, cols, begin_y, begin_x;

Create a new window with *lines* lines and *cols* columns starting at position *(begin_y,* begin_x) in the middle of the window *win*. This means that any change made to either window in the area covered by the subwindow will be made on both windows. *(begin_y,* begin_x) are specified relative to the overall

screen, not the relative (0, 0) of *win*. If either *lines* or *cols* is 0 (zero), that dimension will be set to *(LINES – begin_y)* or *(COLS – begin_x)* respectively.

unctrl(ch)†
char ch;

This is actually a debug function for the library, but it is of general usefulness. It returns a string which is a representation of *ch*. Control characters become their upper-case equivalents preceded by a "^". Other letters stay just as they are. To use *unctrl*(), you must have **#include <unctrl.h>** in your file.

11.9 Details

gettmode()

Get the tty stats. This is normally called by *initscr*().

mvcur(lasty, lastx, newy, newx)
int lasty, lastx, newy, newx;

Moves the terminal's cursor from *(lasty, lastx)* to *newy, newx)* in an approximation of optimal fashion. This routine uses the functions borrowed from *ex* version 2.6. It is possible to use this optimization without the benefit of the screen routines. With the screen routines, this should not be called by the user. *move*() and *refresh*() should be used to move the cursor position, so that the routines know what's going on.

scroll(win)
WINDOW *win;

Scroll the window upward one line. This is normally not used by the user.

savetty()†

resetty()†

savetty() saves the current tty characteristic flags. *resetty*() restores them to what *savetty*() stored. These functions are performed automatically by *initscr*() and *endwin*().

setterm(name)
char *name;

Set the terminal characteristics to be those of the terminal named *name*. This is normally called by *initscr*().

tstp()

If the new **tty**(4) driver is in use, this function will save the current tty state and then put the process to sleep. When the process gets restarted, it restores the tty state and then calls *wrefresh(curscr)* to redraw the screen. *initscr*() sets the signal SIGTSTP to trap to this routine.

Appendix A

11.10 Capabilities from termcap

11.10.1 Disclaimer

The description of terminals is a difficult business, and we only attempt to summarize the capabilities here: for a full description see the paper describing termcap.

11.10.2 Overview

Capabilities from termcap are of three kinds: string valued options, numeric valued options, and boolean options. The string valued options are the most complicated, since they may include padding information, which we describe now.

Intelligent terminals often require padding on intelligent operations at high (and sometimes even low) speed. This is specified by a number before the string in the capability, and has meaning for the capabilities which have a **P** at the front of their comment. This normally is a number of milliseconds to pad the operation. In the current system which has no true programmable delays, we do this by sending a sequence of pad characters (normally nulls, but can be changed (specified by *PC*)). In some cases, the pad is better computed as some number of milliseconds times the number of affected lines (to the bottom of the screen usually, except when terminals have insert modes which will shift several lines.) This is specified as, e.g., **12***. before the capability, to say 12 milliseconds per affected whatever (currently always line). Capabilities where this makes sense say **P***.

11.10.3 Variables Set By setterm()

Variables Set By *setterm()*

Type	Name	Pad	Description
char *	AL	P*	Add new blank Line
bool	AM		Automatic Margins
char *	BC		Back Cursor movement
bool	BS		BackSpace works
char *	BT	P	Back Tab
bool	CA		Cursor Addressable
char *	CD	P*	Clear to end of Display
char *	CE	P	Clear to End of line
char *	CL	P*	CLear screen
char *	CM	P	Cursor Motion
char *	DC	P*	Delete Character
char *	DL	P*	Delete Line sequence
char *	DM		Delete Mode (enter)
char *	DO		DOwn line sequence
char *	ED		End Delete mode
bool	EO		can Erase Overstrikes with ' '
char *	EI		End Insert mode
char *	HO		HOme cursor
bool	HZ		HaZeltine ~ braindamage
char *	IC	P	Insert Character
bool	IN		Insert – Null blessing
char *	IM		enter Insert Mode (IC usually set, too)
char *	IP	P*	Pad after char Inserted using IM + IE
char *	LL		quick to Last Line, column 0
char *	MA		ctrl character MAp for cmd mode
bool	MI		can Move in Insert mode
bool	NC		No Cr: \r sends \r\n then eats \n
char *	ND		Non – Destructive space
bool	OS		OverStrike works
char	PC		Pad Character
char *	SE		Standout End (may leave space)
char *	SF	P	Scroll Forwards

char *	SO		Stand Out begin (may leave space)
char *	SR	P	Scroll in Reverse
char *	TA	P	TAb (not ^I or with padding)
char *	TE		Terminal address enable Ending sequence
char *	TI		Terminal address enable Initialization
char *	UC		Underline a single Character
char *	UE		Underline Ending sequence
bool	UL		UnderLining works even though !OS
char *	UP		UPline
char *	US		Underline Starting sequence*
char *	VB		Visible Bell
char *	VE		Visual End sequence
char *	VS		Visual Start sequence
bool	XN		a Newline gets eaten after wrap

Names starting with **X** are reserved for severely nauseous glitches

11.10.4 Variables Set By gettmode()

Variables Set By *gettmode()*

Type	Name	Description
bool	NONL	Term can't hack linefeeds doing a CR
bool	GT	Gtty indicates Tabs
bool	UPPERCASE	Terminal generates only uppercase letters

---

* US and UE, if they do not exist in the termcap entry, are copied from SO and SE in *setterm*

Appendix B

11.11 The WINDOW structure

The WINDOW structure is defined as follows:

```
#define WINDOW          struct __win__st
struct __win__st {
        short   __cury, __curx;
        short   __maxy, __maxx;
        short   __begy, __begx;
        short   __flags;
        bool    __clear;
        bool    __leave;
        bool    __scroll;
        char    **__y;
        short   *__firstch;
        short   *__lastch;
};
#define __SUBWIN        01
#define __ENDLINE       02
#define __FULLWIN       04
#define __SCROLLWIN     010
#define __STANDOUT      0200
```

*__cury** and *__curx* are the current (y, x) co-ordinates for the window. New characters added to the screen are added at this point. *__maxy* and *__maxx* are the maximum values allowed for (*__cury, __curx*). *__begy* and *__begx* are the starting (y, x) co-ordinates on the terminal for the window, i.e., the window's home. *__cury*, *__curx*, *__maxy*, and *__maxx* are measured relative to (*__begy, __begx*), not the terminal's home.

---

* All variables not normally accessed directly by the user are named with an initial "__" to avoid conflicts with the user's variables.

__clear tells if a clear-screen sequence is to be generated on the next *refresh*() call. This is only meaningful for screens. The initial clear-screen for the first *refresh*() call is generated by initially setting clear to be TRUE for *curscr*, which always generates a clear-screen if set, irrelevant of the dimensions of the window involved. *__leave* is TRUE if the current (y, x) co-ordinates and the cursor are to be left after the last character changed on the terminal, or not moved if there is no change. *__scroll* is TRUE if scrolling is allowed.

__y is a pointer to an array of lines which describe the terminal. Thus:

__y[i] is a pointer to the *i*th line, and

__y[i][j] is the *j*th character on the *i*th line.

__flags can have one or more values or'd into it. **__SUBWIN** means that the window is a subwindow, which indicates to *delwin*() that the space for the lines is not to be freed. **__ENDLINE** says that the end of the line for this window is also the end of a screen. **__FULLWIN** says that this window is a screen. **__SCROLLWIN** indicates that the last character of this screen is at the lower right-hand corner of the terminal; *i.e.*, if a character was put there, the terminal would scroll. **__STANDOUT** says that all characters added to the screen are in standout mode.

Appendix C

11.12 Examples

Here we present a few examples of how to use the package. They attempt to be representative, though not comprehensive.

11.12.1 Screen Updating

The following examples are intended to demonstrate the basic structure of a program using the screen updating sections of the package. Several of the programs require calculational sections which are irrelevant to the example, and are therefore usually not included. It is hoped that the data structure definitions give enough of an idea to allow understanding of what the relevant portions do.

11.12.2 Twinkle

This is a moderately simple program which prints pretty patterns on the screen that might even hold your interest for 30 seconds or more. It switches between patterns of asterisks, putting them on one by one in random order, and then taking them off in the same fashion. It is more efficient to write this using only the motion optimization, as is demonstrated below.

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

/*
 * the idea for this program was a product of the imagination of
 * Kurt Schoens.  Not responsible for minds lost or stolen.
 */
```

```
#define NCOLS                   80
#define NLINES                  24
#define MAXPATTERNS             4

struct locs {
        char    y, x;
};

typedef struct locs         LOCS;

LOCS    Layout[NCOLS * NLINES];         /* current board layout */

int     Pattern, /* current pattern number */

        Numstars;       /* number of stars in pattern */

main() {

        char    *getenv();
        int     die();

        srand(getpid());  /* initialize random sequence */
```

```
        initscr();
        signal(SIGINT, die);
        noecho();
        nonl();
        leaveok(stdscr, TRUE);
        scrollok(stdscr, FALSE);

        for (;;) {
                makeboard();        /* make the board setup */
                puton('*');         /* put on '*'s */
                puton(' ');         /* cover up with ' 's */

        }
}
/*
 * On program exit, move the cursor to the lower left corner by
 * direct addressing, since current location is not guaranteed.
 * We lie and say we used to be at the upper right corner to guarantee
 * absolute addressing.
 */


die() {

        signal(SIGINT, SIG_IGN);
        mvcur(0, COLS-1, LINES-1, 0);
        endwin();
        exit(0);
}
/*
 * Make the current board setup.  It picks a random pattern and
 * calls ison() to determine if the character is on that pattern
 * or not.
 */
```

```
makeboard() {

        reg int   y, x;
        reg LOCS        *lp;

        Pattern = rand() % MAXPATTERNS;
        lp = Layout;
        for (y = 0; y < NLINES; y++)
        for (x = 0; x < NCOLS; x++)
        if (ison(y, x)) {
                lp->y = y;
                lp++->x = x;
        }
        Numstars = lp - Layout;
}
/*
 * Return TRUE if (y, x) is on the current pattern.
 */
```

```
ison(y, x)
reg int   y, x; {

        switch (Pattern) {
          case 0: /* alternating lines */

        return !(y & 01);
          case 1: /* box */

        if (x >= LINES && y >= NCOLS)
                return FALSE;
        if (y < 3 || y >= NLINES - 3)
                return TRUE;
        return (x < 3 || x >= NCOLS - 3);
          case 2: /* holy pattern! */

        return ((x + y) & 01);
          case 3: /* bar across center */

        return (y >= 9 && y <= 15);
        }
        /* NOTREACHED */

}


puton(ch)
reg char ch; {

        reg LOCS        *lp;
        reg int   r;
        reg LOCS        *end;
        LOCS    temp;
```

```
        end = &Layout[Numstars];
        for (lp = Layout; lp < end; lp++) {
                r = rand() % Numstars;
                temp = *lp;
                *lp = Layout[r];
                Layout[r] = temp;
        }

        for (lp = Layout; lp < end; lp++) {
                mvaddch(lp->y, lp->x, ch);
                refresh();
        }
}
```

11.12.3 Life

This program plays the famous computer pattern game of life (Scientific American, May, 1974). The calculational routines create a linked list of structures defining where each piece is. Nothing here claims to be optimal, merely demonstrative. This program, however, is a very good place to use the screen updating routines, as it allows them to worry about what the last position looked like, so you don't have to. It also demonstrates some of the input routines.

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

/*
 * Run a life game.  This is a demonstration program for
 * the Screen Updating section of the -lcurses cursor package.
 */
```

```
struct lst__st {    /* linked list element */

        int     y, x;     /* (y, x) position of piece */

        struct lst__st     *next, *last;        /* doubly linked */

};

typedef struct lst__st     LIST;

LIST    *Head;   /* head of linked list */


main(ac, av)
int     ac;
char    *av[]; {

        int     die();

        evalargs(ac, av);  /* evaluate arguments */

        initscr();         /* initialize screen package */
        signal(SIGINT, die);       /* set to restore tty stats */

        crmode();          /* set for char-by-char */

        noecho();          /* input */

        nonl();            /* for optimization */

        getstart();        /* get starting position */

        for (;;) { prboard();      /* print out current board */
        update();                  /* update board position */
        }
}
```

```
/*
 * This is the routine which is called when rubout is hit.
 * It resets the tty stats to their original values.  This
 * is the normal way of leaving the program.
 */

die() {

        signal(SIGINT, SIG_IGN);        /* ignore rubouts */

        mvcur(0, COLS-1, LINES-1, 0); /* go to bottom of screen */

        endwin();       /* set terminal to initial state */

        exit(0);
}

/*
 * Get the starting position from the user.  They keys u, i, o, j, l,
 * m, ,, and . are used for moving their relative directions from the
 * k key. Thus, u moves diagonally up to the left, , moves directly down,
 * etc.  x places a piece at the current position, " " takes it away.
 * The input can also be from a file.  The list is built after the
 * board setup is ready.
 */
```

```
getstart() {

        reg char c;
        reg int  x, y;

        box(stdscr, '|', '_');      /* box in the screen */

        move(1, 1);                 /* move to upper left corner */

        do {
                refresh();          /* print current position */

                if ((c=getch()) == 'q')
                break;
                switch (c) {
                  case 'u':
                  case 'i':
                  case 'o':
                  case 'j':
                  case 'l':
                  case 'm':
                  case ',':
                  case '.':
                adjustyx(c);
                break;
                  case 'f':
                mvaddstr(0, 0, ``File name: ``);
                getstr(buf);
                readfile(buf);
                break;
                  case 'x':
                addch('X');
                break;
                  case ' ':
                addch(' ');
                break;
                }
        }
```

```
        if (Head != NULL)     /* start new list */

        dellist(Head);
        Head = malloc(sizeof (LIST));

/*
 * loop through the screen looking for 'x's, and add a list
 * element for each one
 */

        for (y = 1; y < LINES - 1; y++)
        for (x = 1; x < COLS - 1; x++) {
                move(y, x);
                if (inch() == 'x')
                        addlist(y, x);

}
/*
 * Print out the current board position from the linked list
        */
```

```
prboard() {

        reg LIST*hp;

        erase();  /* clear out last position */

        box(stdscr, '|', '_');        /* box in the screen */

/*
 * go through the list adding each piece to the newly
 * blank board
 */

        for (hp = Head; hp; hp = hp->next)
        mvaddch(hp->y, hp->x, 'X');

        refresh();
}
```

11.12.4 Motion optimization

The following example shows how motion optimization is written on its own. Programs which flit from one place to another without regard for what is already there usually do not need the overhead of both space and time associated with screen updating. They should instead use motion optimization.

11.12.5 Twinkle

The **twinkle** program is a good candidate for simple motion optimization. Here is how it could be written (only the routines that have been changed are shown):

```
main() {

        reg char *sp;
        char     *getenv();
        int      __putchar(), die();

        srand(getpid());  /* initialize random sequence */

        if (isatty(0)) {
              gettmode();
              if (sp = getenv("TERM"))
              setterm(sp);
        signal(SIGINT, die);
        }
        else {
                printf("Need a terminal on %d\n", __tty__ch);
                exit(1);
        }
        __puts(TI);
        __puts(VS);

        noecho();
        nonl();
        tputs(CL, NLINES, __putchar);
        for (;;) {
                makeboard();      /* make the board setup */
                puton('*');        /* put on '*'s */
                puton(' ');       /* cover up with ' 's */

        }
}

/*
 * __putchar defined for tputs() (and __puts())
 */
```

```
__putchar(c)
reg char c; {

        putchar(c);
}
puton(ch)
char    ch; {

        static int lasty, lastx;
        reg LOCS        *lp;
        reg int   r;
        reg LOCS        *end;
        LOCS    temp;

        end = &Layout[Numstars];
        for (lp = Layout; lp < end; lp++) {
                r = rand() % Numstars;
                temp = *lp;
                *lp = Layout[r];
                Layout[r] = temp;
        }

        for (lp = Layout; lp < end; lp++) /* prevent scrolling */

        if (!AM || (lp->y < NLINES - 1 || lp->x < NCOLS - 1)) {
                mvcur(lasty, lastx, lp->y, lp->x);
                putchar(ch);
                lasty = lp->y;
                if ((lastx = lp->x + 1) >= NCOLS)
                if (AM) {
                        lastx = 0;
                        lasty++;
                }
        else
                lastx = NCOLS - 1;
        }
}
```

Contents

12

Chapter 12

VENIX ASSEMBLER REFERENCE MANUAL

12.1 INTRODUCTION

This document describes the usage and input syntax of the VENIX PDP-11 assembler **as**. The details of the PDP-11 are not described.

The input syntax of the VENIX assembler is generally similar to that of the DEC assembler PAL-11R, although its internal workings and output format are unrelated. It may be useful to read the publication DEC-11-ASDB-D, which describes PAL-11R, although naturally one must use care in assuming that its rules apply to **as**.

as is a rather ordinary assembler without macro capabilities. It produces an output file that contains relocation information and a complete symbol table; thus the output is acceptable to the VENIX link-editor **ld**, which may be used to combine the outputs of several assembler runs and to obtain object programs from libraries. The output format has been designed so that if a program contains no unresolved references to external symbols, it is executable without further processing.

12.2 USAGE

as is used as follows:

as [**−u**] [**−o** *output*] *file1* ...

If the optional **−u** argument is given, all undefined symbols in the current assembly will be made undefined-external. See the **.globl** directive below.

The other arguments name files which are concatenated and assembled. Thus programs may be written in several pieces and assembled together.

The output of the assembler is by default placed on the file **a.out** in the current directory; the **−o** flag causes the output to be placed on the named file. If there were no unresolved external references, and no errors detected, the output file is marked executable; otherwise, if it is produced at all, it is made non-executable.

12.3 LEXICAL CONVENTIONS

Assembler tokens include identifiers (alternatively, "symbols" or "names"), temporary symbols, constants, and operators.

12.3.1 Identifiers

An identifier consists of a sequence of alphanumeric characters (including period ".", underscore "_", and tilde "~" as alphanumeric) of which the first may not be numeric. Only the first eight characters are significant. When a name begins with a tilde, the tilde is discarded and that occurrence of the identifier generates a unique entry in the symbol table which can match no other occurrence of the identifier. This feature is used by the C compiler to place names of local variables in the output symbol table without having to worry about making them unique.

12.3.2 Temporary Symbols

A temporary symbol consists of a digit followed by "f" or "b". Temporary symbols are discussed fully in section 5.1.

12.3.3 Constants

An octal constant consists of a sequence of digits; "8" and "9" are taken to have octal value 10 and 11. The constant is truncated to 16 bits and interpreted in two's complement notation.

A decimal constant consists of a sequence of digits terminated by a decimal point ".". The magnitude of the constant should be representable in 15 bits; i.e., be less than 32,768.

A single-character constant consists of a single quote ' followed by an ASCII1 character not a new-line. Certain dual-character escape sequences are acceptable in place of the ASCII character to represent new-line and other non-graphics (see the section "String Statements"). The constant's value has the code for the given character in the least significant byte of the word and is null-padded on the left.

A double-character constant consists of a double quote " followed by a pair of ASCII characters not including new-line. Certain dual-character escape sequences are acceptable in place of either of the ASCII characters to represent new-line and other non-graphics (see "String Statements"). The constant's value has the code for the first given character in the least significant byte and that for the second character in the most significant byte.

12.3.4 Operators

There are several single- and double-character operators; see section 6.

12.3.5 Blanks

Blank and tab characters may be interspersed freely between tokens, but may not be used within tokens (except character constants). A blank or tab is required to separate adjacent identifiers or constants not otherwise separated.

12.3.6 Comments

The character "/" introduces a comment, which extends through the end of the line on which it appears. Comments are ignored by the assembler.

12.4 SEGMENTS

Assembled code and data fall into three segments: the text segment, the data segment, and the bss segment. The text segment is the one in which the assembler begins, and it is the one into which instructions are typically placed. The VENIX system will, if desired, enforce the purity of the text segment of programs by trapping write operations into it. Object programs produced by the assembler must be processed by the link-editor **ld** (using its **−n** flag) if the text segment is to be write-protected. A single copy of the text segment is shared among all processes executing such a program.

The data segment is available for placing data or instructions which will be modified during execution. Anything which may go in the text segment may be put into the data segment. In programs with write-protected, sharable text segments, data segment contains the initialized but variable parts of a program. If the text segment is not pure, the data segment begins immediately after the text segment; if the text segment is pure, the data segment begins at the lowest 8K byte boundary after the text segment.

The bss segment may not contain any explicitly initialized code or data. The length of the bss segment (like that of text or data) is determined by the high-water mark of the location counter within it. The bss segment is actually an extension of the data segment and begins immediately after it. At the start of execution of a program, the bss segment is set to 0. Typically the bss segment is set up by statements exemplified by

```
lab: . = .+10
```

The advantage in using the bss segment for storage that starts off empty is that the initialization information need not be stored in the output file. See also "Location Counter" and "Assignment Statements" below.

12.5 THE LOCATION COUNTER

One special symbol, ".", is the location counter. Its value at any time is the offset within the appropriate segment of the start of the statement in which it appears. Assignments may be made to the location counter, with the restriction that the current segment may not change; furthermore, the value of "." may not decrease. If the effect of the assignment is to increase the value of ".", the required number of null bytes are generated (but see "Segments" above).

12.6 STATEMENTS

A source program is composed of a sequence of *statements.* Statements are separated either by new-lines or by semicolons. There are five kinds of statements: *null* statements, *expression* statements, *assignment* statements, *string* statements, and *keyword* statements.

Any kind of statement may be preceded by one or more labels.

12.6.1 Labels

There are two kinds of labels: *name* labels and *numeric* labels. A name label consists of a name followed by a colon (:). The effect of a name label is to assign the current value and type of the location counter "." to the name. An error is indicated in pass 1 if the name is already defined; an error is indicated in pass 2 if the "." value assigned changes the definition of the label.

A numeric label consists of a digit **0** to **9** followed by a colon (:). Such a label serves to define temporary symbols of the form *n***b** and *n***f**, where *n* is the digit of the label. As in the case of name labels, a numeric label assigns the current value and type of "." to the temporary symbol. However, several numeric labels with the same digit may be used within the same assembly. References of the form *n***f** refer to the first numeric label *n*: *f*orward from the reference; *n***b** symbols refer to the first *n* : label *b*ackward from the reference. This sort of temporary label was introduced by Knuth [*The Art of Computer Programming, Vol I: Fundamental Algorithms*]. Such labels tend to conserve both the symbol table space of the assembler and the inventive powers of the programmer.

12.6.2 Null Statements

A null statement is an empty statement (which may, however, have labels). A null statement is ignored by the assembler. Common examples of null statements are empty lines or lines containing only a label.

12.6.3 Expression Statements

An expression statement consists of an arithmetic expression not beginning with a keyword. The assembler computes its (16-bit) value and places it in the output stream, together with the appropriate relocation bits.

12.6.4 Assignment Statements

An assignment statement consists of an *identifier*, an equals sign (=), and an *expression*. The value and type of the expression are assigned to the identifier.n It is not required that the type or value be the same in pass 2 as in pass 1, nor is it an error to redefine any symbol by assignment.

Any external attribute of the expression is lost across an assignment. This means that it is not possible to declare a global symbol by assigning to it, and that it is impossible to define a symbol to be offset from a non-locally defined global symbol.

As mentioned, it is permissible to make assignments to the location counter ".". It is required, however, that the type of the expression assigned be of the same type as ".", and it is forbidden to decrease the value of ".". In practice, the most common assignment to "." has the form ". = . + *n*" for some number *n;* this has the effect of generating *n* null bytes.

12.6.5 String Statements

A string statement generates a sequence of bytes containing ASCII characters. A string statement consists of a left string quote "<" followed by a sequence of ASCII characters not including newline, followed by a right string quote ">". Any of the ASCII characters may be replaced by a two-character escape sequence to represent certain non-graphic characters, as follows:

\n	NL	(012)
\s	SP	(040)
\t	HT	(011)
\e	EOT	(004)
\0	NUL	(000)
\r	CR	(015)
\a	ACK	(006)
\p	PFX	(033)
\\	\	
\>	>	

The last two are included so that the escape character and the right string quote may be represented. The same escape sequences may also be used within single- and double-character constants (see the section "Constants" above).

12.6.6 Keyword Statements

Keyword statements are numerically the most common type, since most machine instructions are of this sort. A keyword statement begins with one of the many predefined keywords of the assembler; the syntax of the remainder depends on the keyword. All the keywords are listed below with the syntax they require.

12.7 EXPRESSIONS

An expression is a sequence of symbols representing a value. Its constituents are identifiers, constants, temporary symbols, operators, and brackets. Each expression has a type.

All operators in expressions are fundamentally binary in nature; if an operand is missing on the left, a 0 of absolute type is assumed. Arithmetic is two's complement and has 16 bits of precision. All operators have equal precedence, and expressions are evaluated strictly left to right except for the effect of brackets.

12.7.1 Expression Operators

The operators are:

(blank)	when there is no operator between operands, the effect is exactly the same as if a "+" had appeared.
+	addition
−	subtraction
*	multiplication
\/	division (note that plain "/" starts a comment)
&	bitwise **and**
|	bitwise **or**
\>	logical right shift
\<	logical left shift

%	modulo
!	*a* ! *b* is *a* **or** (**not** *b*); i.e., the **or** of the first operand and the one's complement of the second; most common use is as a unary.
^	result has the value of first operand and the type of the second; most often used to define new machine instructions with syntax identical to existing instructions.

Expressions may be grouped by use of square brackets "[ ]". (Round parentheses are reserved for address modes.)

12.7.2 Types

The assembler deals with a number of types of expressions. Most types are attached to keywords and used to select the routine which treats that keyword. The types likely to be met explicitly are:

undefined
Upon first encounter, each symbol is undefined. It may become undefined if it is assigned an undefined expression. It is an error to attempt to assemble an undefined expression in pass 2; in pass 1, it is not (except that certain keywords require operands which are not undefined).

undefined external
A symbol which is declared **.globl** but not defined in the current assembly is an undefined external. If such a symbol is declared, the link editor **ld** must be used to load the assembler's output with another routine that defines the undefined reference.

absolute
An absolute symbol is defined ultimately from a constant. Its value is unaffected by any possible future applications of the link-editor to the output file.

text The value of a text symbol is measured with respect to the beginning of the text segment of the program. If the assembler output is link-edited, its text symbols may change in value since the program need not be the first in the link editor's output. Most text symbols are defined by appearing as labels. At the start of an assembly, the value of "." is text 0.

data The value of a data symbol is measured with respect to the origin of the data segment of a program. Like text symbols, the value of a data symbol may change during a subsequent link-editor run since previously loaded programs may have data segments. After the first *.data* statement, the value of "." is data 0.

bss The value of a bss symbol is measured from the beginning of the bss segment of a program. Like text and data symbols, the value of a bss symbol may change during a subsequent link-editor run, since previously loaded programs may have bss segments. After the first *.bss* statement, the value of "." is bss 0.

external absolute, text, data, or bss
symbols declared **.globl** but defined within an assembly as absolute, text, data, or bss symbols may be used exactly as if they were not declared **.globl**; however, their value and type are available to the link editor so that the program may be loaded with others that reference these symbols.

register
The symbols

r0 ... r5
fr0 ... fr5
sp
pc

are predefined as register symbols. Either they or symbols defined from them must be used to refer to the six general-purpose, six floating-point, and the 2 special-purpose machine registers. The behavior of the floating register names is identical to that of the corresponding general register names; the former are provided as a mnemonic aid.

other types
Each keyword known to the assembler has a type which is used to select the routine which processes the associated keyword statement. The behavior of such symbols when not used as keywords is the same as if they were absolute.

12.7.3 Type Propagation In Expressions

When operands are combined by expression operators, the result has a type which depends on the types of the operands and on the operator. The rules involved are complex to state but were intended to be sensible and predictable. For purposes of expression evaluation the important types are

undefined
absolute
text
data
bss
undefined external
other

The combination rules are then:

If one of the operands is undefined, the result is undefined. If both operands are absolute, the result is absolute. If an absolute is combined with one of the "other types" mentioned above, or with a register expression, the result has the register or other type. As a consequence, one can refer to r3 as "r0+3". If two operands of "other type" are combined, the result has the numerically larger type An "other type" combined with an explicitly discussed type other than absolute acts like an absolute.

Further rules applying to particular operators are:

+ If one operand is text-, data-, or bss-segment relocatable, or is an undefined external, the result has the postulated type and the other operand must be absolute.

− If the first operand is a relocatable text-, data-, or bss-segment symbol, the second operand may be absolute (in which case the result has the type of the first operand); or the second operand may have the same type as the first (in which case the result is absolute). If the first operand is external undefined, the second must be absolute. All other combinations are illegal.

This operator follows no other rule than that the result has the value of the first operand and the type of the second.

others It is illegal to apply these operators to any but absolute symbols.

12.8 PSEUDO-OPERATIONS

The keywords listed below introduce statements that generate data in unusual forms or influence the later operations of the assembler. The metanotation

[*stuff*] ...

means that 0 or more instances of the given stuff may appear. Also, boldface tokens are literals, italic words are substitutable.

12.8.1 .byte expression [, expression] ...

The *expression*s in the comma-separated list are truncated to 8 bits and assembled in successive bytes. The expressions must be absolute. This statement and the string statement above are the only ones that assemble data one byte at at time.

12.8.2 .even

If the location counter "." is odd, it is advanced by one so the next statement will be assembled at a word boundary.

12.8.3 .if expression

The *expression* must be absolute and defined in pass 1. If its value is nonzero, the **.if** is ignored; if zero, the statements between the **.if** and the matching **.endif** (below) are ignored. **.if** may be nested. The effect of **.if** cannot extend beyond the end of the input file in which it appears. (The statements are not totally ignored, in the following sense: **.if**s and **.endif**s are scanned for, and moreover all names are entered in the symbol table. Thus names occurring only inside an **.if** will show up as undefined if the symbol table is listed.)

12.8.4 .endif

This statement marks the end of a conditionally-assembled section of code. See **.if** above.

12.8.5 .globl name [, name] ...

This statement makes the *names* external. If they are otherwise defined (by assignment or appearance as a label) they act within the assembly exactly as if the **.globl** statement were not given; however, the link editor **ld** may be used to combine this routine with other routines that refer these symbols.

Conversely, if the given symbols are not defined within the current assembly, the link editor can combine the output of this assembly with that of others which define the symbols. As discussed in section 1, it is possible to force the assembler to make all otherwise undefined symbols external.

12.8.6 .text, .data, and .bss

These three pseudo-operations cause the assembler to begin assembling into the text, data, or bss segment respectively. Assembly starts in the text segment. It is forbidden to assemble any code or data into the bss segment, but symbols may be defined and "." moved about by assignment.

12.8.7 .comm name , expression

Provided the *name* is not defined elsewhere, this statement is equivalent to

.globl *name*
name = *expression* ^ *name*

That is, the type of *name* is "undefined external", and its value is *expression*. In fact the *name* behaves in the current assembly just like an undefined external. However, the link-editor **ld** has been special-cased so that all external symbols which are not otherwise defined, and which have a non-zero value, are defined to lie in the bss segment, and enough space is left after the symbol to hold *expression* bytes. All symbols which become defined in this way are located before all the explicitly defined bss-segment locations.

12.9 MACHINE INSTRUCTIONS

Because of the rather complicated instruction and addressing structure of the PDP-11, the syntax of machine instruction statements is varied. Although the following sections give the syntax in detail, the machine handbooks should be consulted on the semantics.

12.9.1 Sources and Destinations

The syntax of general source and destination addresses is the same. Each must have one of the following forms, where *reg* is a register symbol, and *expr* is any sort of expression:

syntax	words	mode
reg	0	00 + *reg*
(*reg*) +	0	20 + *reg*
− (*reg*)	0	40 + *reg*
expr(*reg*)	1	60 + *reg*
(*reg*)	0	10 + *reg*
* *reg*	0	10 + *reg*
* (*reg*) +	0	30 + *reg*
* − (*reg*)	0	50 + *reg*
* (*reg*)	1	70 + *reg*
* *expr*(*reg*)	1	70 + *reg*
expr	1	67
$ *expr*	1	27
* *expr*	1	77
* $ *expr*	1	37

The *words* column gives the number of address words generated; the *mode* column gives the octal address-mode number. The syntax of the address forms is identical to that in DEC assemblers, except that "*" has been substituted for "@" and "$" for "#"; the VENIX typing conventions make "@" and "#" rather inconvenient.

Notice that mode "*reg" is identical to "(reg)"; that "*(reg)" generates an index word (namely, 0); and that addresses consisting of an unadorned expression are assembled as pc-relative references independent of the type of the expression. To force a non-relative reference, the form "*$expr" can be used, but notice that further indirection is impossible.

12.9.2 Simple Machine Instructions

The following instructions are defined as absolute symbols:

clc
clv
clz
cln
sec
sev
sez
sen

They therefore require no special syntax. The PDP-11 hardware allows more than one of the "clear" class, or alternatively more than one of the "set" class to be **or**-ed together; this may be expressed as follows:

clc | clv

12.9.3 Branch

The following instructions take an expression as operand. The expression must lie in the same segment as the reference, cannot be undefined-external, and its value cannot differ from the current location of "." by more than 254 bytes:

br	**blos**	
bne	**bvc**	
beq	**bvs**	
bge	**bhis**	
blt	**bec**	**(= bcc)**
bgt	**bcc**	
ble	**blo**	
bpl	**bcs**	
bmi	**bes**	**(= bcs)**
bhi		

bes ("branch on error set") and **bec** ("branch on error clear") are intended to test the error bit returned by system calls (which is the c-bit).

12.9.4 Extended Branch Instructions

The following symbols are followed by an expression representing an address in the same segment as ".". If the target address is close enough, a branch-type instruction is generated; if the address is too far away, a **jmp** will be used.

jbr	**jlos**
jne	**jvc**
jeq	**jvs**
jge	**jhis**
jlt	**jec**
jgt	**jcc**
jle	**jlo**
jpl	**jcs**
jmi	**jes**
jhi	

jbr turns into a plain **jmp** if its target is too remote; the others (whose names are constructed by replacing the "b" in the branch instruction's name by "j") turn into the converse branch over a **jmp** to the target address.

12.9.5 Single Operand Instructions

The following symbols are names of single-operand machine instructions. The form of address expected is discussed in the section "Sources and Destinations" above.

clr	**sbcb**
clrb	**ror**
com	**rorb**
comb	**rol**
inc	**rolb**
incb	**asr**
dec	**asrb**
decb	**asl**
neg	**aslb**
negb	**jmp**
adc	**swab**
adcb	**tst**
sbc	**tstb**

12.9.6 Double Operand Instructions

The following instructions take a general source and destination, separated by a comma, as operands.

mov
movb
cmp
cmpb
bit
bitb
bic
bicb
bis
bisb
add
sub

12.9.7 Miscellaneous Instructions

The following instructions have more specialized syntax. Here *reg* is a register name, *src* and *dst* a general source or destination, and *expr* is an expression:

jsr	*reg,dst*	
rts	*reg*	
sys	*expr*	
ash	*src, reg*	**(or, als)**
ashc	*src, reg*	**(or, alsc)**
mul	*src, reg*	**(or, mpy)**
div	*src, reg*	**(or, dvd)**
xor	*reg, dst*	
sxt	*dst*	
mark	*expr*	
sob	*reg, expr*	

sys is another name for the **trap** instruction. It is used to code system calls. Its operand is required to be expressible in 6 bits. The expression in **mark** must be expressible in six bits, and the expression in **sob** must be in the same segment as

"**.**", must not be external-undefined, must be less than "**.**", and must be within 510 bytes of "**.**".

12.9.8 Floating-point Unit Instructions

The following floating-point operations are defined, with syntax as indicated:

cfcc		
setf		
setd		
seti		
setl		
clrf	*fdst*	
negf	*fdst*	
absf	*fdst*	
tstf	*fsrc*	
movf	*fsrc, freg*	**(= ldf)**
movf	*freg, fdst*	**(= stf)**
movif	*src, freg*	**(= ldcif)**
movfi	*freg, dst*	**(= stcfi)**
movof	*fsrc, freg*	**(= ldcdf)**
movfo	*freg, fdst*	**(= stcfd)**
movie	*src, freg*	**(= ldexp)**
movei	*freg, dst*	**(= stexp)**
addf	*fsrc, freg*	
subf	*fsrc, freg*	
mulf	*fsrc, freg*	
divf	*fsrc, freg*	
cmpf	*fsrc, freg*	
modf	*fsrc, freg*	
ldfps	*src*	
stfps	*dst*	
stst	*dst*	

fsrc, *fdst*, and *freg* mean floating-point source, destination, and register respectively. Their syntax is identical to that for their non-floating counterparts, but note that only floating registers 0-3 can be a *freg*.

The names of several of the operations have been changed to bring out an analogy with certain fixed-point instructions. The only strange case is **movf**, which turns into either **stf** or **ldf** depending respectively on whether its first operand is or is not a register. Warning: **ldf** sets the floating condition codes, **stf** does not.

12.10 Other Symbols

12.10.1 ..

The symbol "." is the *relocation counter.* Just before each assembled word is placed in the output stream, the current value of this symbol is added to the word if the word refers to a text, data or bss segment location. If the output word is a pc-relative address word that refers to an absolute location, the value of "." is subtracted.

Thus the value of "." can be taken to mean the starting memory location of the program. The initial value of "." is 0.

The value of "." may be changed by assignment. Such a course of action is sometimes necessary, but the consequences should be carefully thought out. It is particularly ticklish to change "." midway in an assembly or to do so in a program which will be treated by the loader, which has its own notions of ".".

12.10.2 System Calls

System call names are not predefined. They may be found in the file **/usr/include/sys.s**

12.11 DIAGNOSTICS

When an input file cannot be read, its name followed by a question mark is typed and assembly ceases. When syntactic or semantic errors occur, a single-character diagnostic is typed out together with the line number and the file name in which it occurred. Errors in pass 1 cause cancellation of pass 2. The possible errors are:

)	parentheses error
]	parentheses error
>	string not terminated properly
*	indirection (*) used illegally
.	illegal assignment to "."
A	error in address
B	branch address is odd or too remote
E	error in expression
F	error in local ("f" or "b") type symbol
G	garbage (unknown) character
I	end of file in side an .if
M	multiply defined symbol as label
O	word quantity assembled at odd address
P	phase error — "." different in pass1 and pass2
R	relocation error
U	undefined symbol
X	syntax error

Printed in U.S.A.

AA-BM35A-TH