ROS Programmer's Guide

RIDGE

9050

ROS Programmer's Guide

Ridge Computers
2451 Mission College Blvd.
Santa Clara, CA 95054

First Edition: (9050) (AUG 84)

PUBLICATION HISTORY

Manual Title: ROS Programmer's Guide

First Edition: (9050) (AUG 84)

NOTICE

ACKNOWLEDGEMENT

This software and documentation is based in part on the fourth Berkeley Software Distribution, under license from the regents of the University of California. We acknowledge the following individuals for their part in its development: Ken Arnold, Jim Kleckner, Bill Joy, Mark Horton, Rick Blau, Eric Shienbrood, John Foderaro, Geoffrey Peck, Robert P. Corbett, and Randy King.

These tutorial guides are based on documents created at Bell Laboratories to describe UNIX System software, and at the University of California, Berkeley, to describe the Berkeley Software Distribution. Credits are given on the first page of each document contained in this volume. Some text has been changed to more accurately describe Ridge Computers' implementation of the software. Inappropriate material has been deleted.

PREFACE

The ROS Programmer's Guide (manual 9050) is a collection of tutorial documents related to programming languages and language preprocessors. Except for the PROG section, which explains programming in general, each section contains the detailed information that is omitted from the page of the same name in the ROS Reference Manual (manual 9010).

The topics in the Table of Contents are not the only ones related to programming. The ROS Reference Manual has many entries that are fully explained within. Typically, the user will see a program in the ROS Reference Manual, and if one of these tutorials is mentioned under the **SEE ALSO** heading, he/she will turn to this Programmer's Guide for help. After the reader is familiar with a topic, he/she might refer to the ROS Reference Manual only.

TABLE OF CONTENTS

ROS Programming

This document is based on a paper by Brian W. Kernighan and Dennis M. Ritchie of Bell Laboratories.

1. INTRODUCTION

This paper describes how to write programs that interface with the ROS operating system in a non-trivial way. 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.

This part of the Programmer's Guide summaraizes other material in the ROS Programmer's Guide and the ROS Reference Manual. The reader should have a basic understanding of C (see B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., 1978.)

2. BASICS

2.1. Program Arguments

When a C program is run like a shell command, the shell arguments are passed to the main function in an argument count *argc* and an array *argv* of pointers to character strings that contain the arguments themselves. **Argv[0]** points to the command name itself.

The following program **main** demonstrates this mechanism. It declares the type of the count variable and pointer array and echos all arguments to the output. (This is similar to the code of echo(1)).

```
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) ? '\0' : '\n');
}
```

Argv is an array of pointers to character arrays. Each of those character arrays are terminated by the null character \0 so they can be treated as strings.

If you want to maintain copies of *argc* and *argv* for use by other routines, copy them to external variables.

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 the > character; 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** does, 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 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. Part 6 discusses status returns in more detail.

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 documented in detail in section (3S) *of the ROS Reference Manual.*

The standard I/O library is available on each system that supports C, so programs that confine their system interactions to its facilities can be transported from one system to another essentially without change.

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

3.1. File Access

The programs written so far have all read the standard input and written the standard output, which we have assumed are magically pre-defined. 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(3S)**. **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.

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. See the fopen(3S) page of the ROS reference manual for a listing of the allowable file access modes, such as 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 what 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 to files or pipes as described in Part 2.2. **stdin**, **stdout** and **stderr** are pre-defined 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**, save 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.)

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 Part 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.

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.

4. LOW-LEVEL I/O

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

4.1. File Descriptors

All input and output is done by reading or writing files, because all peripheral devices, even the user's terminal, are treated as 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.

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 transferred per READ or WRITE operation is arbitrary. It can be 1 (reads and writes are performed one character at a time "unbuffered"), 512, 1024, or 4096 (used by Ridge for efficiency) often corresponding to the physical blocksize on peripheral devices.

With these facts, we can write a program to copy its input to its output file. This program copies any file device to any other because its input or output files can be redirected in any way.

```
#define   BUFSIZE   4096 /* best size for Ridge 32 ROS */

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. The character being returned must be masked with **0377** to ensure that it is positive; otherwise sign extension on some machines may make it negative. (The constant **0377** is appropriate for the PDP-11 and VAX, but not necessarily for other machines.) The Ridge 32 does not need to mask characters because they are treated as unsigned characters. For portability, however, masking is a good idea for Ridge programs.

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   4096

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);
}
```

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. The open call can take an optional parameter that offers more sophisticated control of the open process; see open(2).

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 ROS 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 *cp(1)* command, 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 4096
#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("0);
     exit(1);
}
```

With ROS, up to 64 files can be open at one time. Any program which intends to process more than 64 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.

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. If necessary, however, a file can be read or written in any order. The system call **lseek** provides a way to move around in a file without reading or writing:

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

forces the current position in the file **fd** to move to position **offset**, which is 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. (Longs and ints are the same size on ROS.) **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 move to the beginning of (or "rewind") the file,

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

The **0L** argument could also be written as **(long) 0**. With ROS, the file contents are long by default, but it is needed for portability to 16-bit machines. ???

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;
char *buf;
{
     lseek(fd, pos, 0);  /* get to pos */
     return(read(fd, buf, n));
}
```

Older versions of the UNIX System used a called named seek; lseek is so named to avoid confusion with the other.

4.5. 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 intro(2) pages of the ROS Reference Manual, so 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.

Frequently, you may want to print 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.

Use of perror or sys_errno requires the program to contain the code: #include error.h. ???

5. PROCESSES

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

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.

Remember than **getc** and **putc** normally buffer their input; terminal I/O will not be properly synchronized unless this buffering is defeated. For output, use **fflush**; for input, see **setbuf** in the appendix.

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; 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");
```

Execv; is a variation of **execl** that it is used when you don't know how many arguments there will be:

```
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.)

For other variations of exec, see exec(2).

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 say

```
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**.

5.3. Control of Processes — Fork and Wait

So far what we've talked about isn't really all that useful by itself. Now we will show how to regain control after running a program with **execl** or **execv**. Since these routines simply overlay the new program on the old one, to save 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 **fork** nor the **exec** calls 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.

5.4. 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 most 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 somewhat. 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 in the child 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 to from the parent instead of reading. You may find it a useful exercise to step through that case.

The job is not quite done, for we 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.

6. SIGNALS and INTERRUPTS

This section covers the graceful handling of signals from the outside world: *interrupt*, which is sent when the DEL character is typed; *quit*, which is generated by the FS character; *hangup*, which is caused by hanging up the phone; and *terminate*, which is 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 case of *quit*, the debugger is then invoked automatically.

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 is either a function, or a somewhat strange code that requests that the signal either be ignored, or that it 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; he 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 ROS on the Ridge 32; the definitions are sufficiently ugly and unportable to encourage use of the include file.

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

Appendix — The Standard I/O Library

1. 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. 4096) 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. 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 the printf(3S) pages of the ROS 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**.

Lint, a C Program Checker

This document is based on a paper by S. C. Johnson of Bell Laboratories.

Lint examines C source programs and detects some bugs and obscurities that the C compiler does not. It enforces the type rules of C 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 constructions which are legal but wasteful or error-prone.

Lint accepts multiple input files and library specifications, and checks them for consistency.

Lint is separate from the C compiler for practical reasons. The C compiler works fast and efficient, partly because it does not do sophisticated type checking, especially between separately-compiled programs. *Lint* examines compatibilities more carefully.

This document discusses the use of *lint*, gives an overview of the implementation, and gives some hints on the writing of machine-independent C code.

Introduction and Usage

Suppose there are two C source files, *file1*.c and *file2.c*, which are ordinarily compiled and loaded together. Then the command

```
lint file1.c file2.c
```

produces messages describing inconsistencies and inefficiencies in the programs. The program enforces the typing rules of C more strictly than the C compilers (for both historical and practical reasons) enforce them. The command

```
lint -p file1.c file2.c
```

will produce, in addition to the above messages, additional messages which relate to the portability of the programs to other operating systems and machines. Replacing the **-p** by **-h** will produce messages about various error-prone or wasteful constructions which, strictly speaking, are not bugs. Saying **-hp** gets the whole works.

The next several sections describe the major messages; the document closes with sections discussing the implementation and giving suggestions for writing portable C. See **lint(1)** in the **ROS Reference Manual (9010)** for a list of *lint* options.

Philosophy of Lint

Lint cannot ascertain every fact about a program, so it makes some assumptions. For example, the input data may determine whether a given function is ever called, or whether a specific *exit* condition will ever be reached.

Therefore, *lint* algorithms are a compromise. For example, if a function is defined but not mentioned, *lint* flags an error, but if it is mentioned, *lint* assumes the program logic could lead to it.

Lint reports messages in three categories: unused variables and functions, set/used information, and flow of control.

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 "errors of commission" rarely cause working programs to fail, but they are a source of inefficiency, and make programs harder to understand and change. Moreover, information about such unused variables and functions can occasionally serve to discover bugs; if a function does a necessary job, and is never called, something is wrong!

Lint complains 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 float 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; they can be discovered by adding the **– x** flag to the *lint* invocation.

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 complaints 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.

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, and, conversely, many functions and variables defined elsewhere may be used. The **– u** flag may be used to suppress the spurious messages which might otherwise appear.

Set/Used Information

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

The restriction to the physical appearance of variables in the file makes the algorithm simple because the true flow of control need not be discovered. For this reason, *lint* may complain about a program that is legal but in bad style (e.g. might contain at least two **goto**'s).

Because static and external variables are initialized to 0, no meaningful information can be discovered about their uses. The algorithm deals correctly, however, with initialized automatic variables, and variables which are used in the expression which first sets them.

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

Flow of Control

Lint attempts to detect unreachable portions of the programs which it processes. It will complain 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, detecting the special cases **while(1)** and **for(;;)** as infinite loops. *Lint* also complains about loops which cannot be entered at the top; some valid programs may have such loops, but at best they are bad style, at worst bugs.

Lint has an important area of blindness in the flow of control algorithm: it has no way of detecting functions which are called and never return. Thus, a call to *exit* may cause unreachable code which *lint* does not detect; the most serious effects of this are in the determination of returned function values (see the next section).

One form of unreachable statement is not usually complained about by *lint;* a **break** statement that cannot be reached causes no message. Programs generated by *yacc,* and especially *lex,* may have literally hundreds of unreachable **break** statements. The **– O** flag in the C compiler will often eliminate the resulting object code inefficiency. Thus, these unreached statements are of little importance, there is typically nothing the user can do about them, and the resulting messages would clutter up the *lint* output. If these messages are desired, *lint* can be invoked with the **– b** option.

Function Values

Sometimes functions return values which are never used; sometimes programs incorrectly use function "values" which have never been returned. *Lint* addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
return( expr );
```

and

```
return ;
```

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

```
function name 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 complaint 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; it also accounts for a substantial fraction of the "noise" messages produced by *lint.*

On a global scale, *lint* detects cases where a function returns a value, but this value is sometimes, or always, unused. When the value is always unused, 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. Amazingly, this bug has been observed on a couple of occasions in "working" programs; the desired function value just happened to have been computed in the function return register!

Type Checking

Lint enforces the type checking rules of C 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, and 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 also 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.

Type Casts

The type cast feature in C was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where *p* is a character pointer. *Lint* will quite rightly complain. Now, consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this, and has clearly signaled his intentions. It seems harsh for *lint* to continue to complain 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 complaint; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

Nonportable Character Use

On the PDP-11, characters are signed quantities, with a range from – 128 to 127. On most other C implementations, including that on the Ridge 32, characters take on only positive values. Thus, *lint* will flag certain comparisons and assignments as being illegal or nonportable. For example, the fragment

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

works on the PDP-11, but will fail on machines where characters always take on positive values. The real solution is to declare *c* an integer, since *getchar* is actually returning integer values. In any case, *lint* will say "nonportable character comparison".

A similar issue arises with bitfields; when assignments of constant values are made to bitfields, the field may be too small to hold the value. On the Ridge 32, bitfields are unsigned. On some machines, bitfields are considered as signed quantities. While it may seem unintuitive to consider that a two-bit field declared of type **int** cannot hold the value 3, the problem disappears if the bitfield is declared to have type **unsigned**.

Assignments of longs to ints

Bugs may arise from the assignment of **long** to an **int**, which loses accuracy. (On the Ridge 32, **long** and **int** values are the same size, so this problem does not exist.) 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**, losing accuracy. Since there are a number of legitimate reasons for assigning **longs** to **ints**, the detection of these assignments is enabled by the **– a** flag.

Strange Constructions

The **– h** flag enables checking for some legal but poor constructions. For example, in the statement

```
*p++ ;
```

the * does nothing; this provokes the message "null effect" from *lint*. The program fragment

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

is clearly somewhat strange; the test will never succeed. Similarly, the test

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

is equivalent to

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

which may not be the intended action. *Lint* will say "degenerate unsigned comparison" in these cases. If one says

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

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

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

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

or

```
x<<2 + 40
```

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

The **-h** flag causes *lint* to complain about variables which are redeclared in inner blocks different from their use in outer blocks. This is legal, but is bad style.

Ancient History

There are several forms of older syntax which are now considered errors by the C compiler. 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 (+=, -=, etc.) have no such ambiguities. To spur the abandonment of the older forms, *lint* complains about these old fashioned operators.

A similar issue arises with initialization. The older language allowed

```
int x 1;
```

to initialize *x* to 1. This also caused syntactic difficulties: for example,

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

looks somewhat like the beginning of a function declaration:

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

and the compiler must read a fair ways past *x* in order to sure what the declaration really is.. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

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

This is free of any possible syntactic ambiguity.

Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due entirely to alignment restrictions. For example, on the PDP-11, it is reasonable to assign integer pointers to double pointers, since double precision values may begin on any integer boundary. On the Ridge 32, double precision values must begin on even double-word boundaries; thus, not all such assignments make sense. *Lint* tries to detect cases where pointers are assigned to other pointers, and such alignment problems might arise. The message "possible pointer alignment problem" results from this situation whenever either the **-p** or **-h** flags are in effect.

Multiple Uses and Side Effects

For complicated expressions, the best evaluation order for sub-expressions is machine-dependent. On the Ridge 32, evaluation is left-to-right. Other machines use right-to-left. Function calls embedded as arguments of other functions may or may not be treated like ordinary arguments. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators. if any variable is changed by a side effect, and also used elsewhere in the same expression, the result is undefined.

Lint specifically checks for the special case where a simple scalar variable is affected. For example, the statement

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

will draw the complaint:

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

Implementation

Lint consists of two programs and a driver. The first program is a version of the Portable C Compiler which is the basis of the IBM 370, Honeywell 6000, and Interdata 8/32 C compilers. This compiler does lexical and syntax analysis on the input text, constructs and maintains symbol tables, and builds trees for expressions. Instead of writing an intermediate file which is passed to a code generator, as the other compilers do, *lint* produces an intermediate file which consists of lines of ascii text. Each line contains an external variable name, an encoding

of the context in which it was seen (use, definition, declaration, etc.), a type specifier, and a source file name and line number. The information about variables local to a function or file is collected by accessing the symbol table, and examining the expression trees.

Comments about local problems are produced as detected. The information about external names is collected onto an intermediate file. After all the source files and library descriptions have been collected, the intermediate file is sorted to bring all information collected about a given external name together. The second, rather small, program then reads the lines from the intermediate file and compares all of the definitions, declarations, and uses for consistency.

The driver controls this process, and is also responsible for making the options available to both passes of *lint*.

Portability

This section describes some of the differences between various implementations, and discusses the *lint* features which encourage portability.

A difficulty arises from the amount of information retained about external names during the loading process. On the ROS system, externally known names have **unlimited** characters, with distinction between upper- and lowercase letters. On other systems, the number of significant characters in a name may be less, or case distinction may be lost. This leads to situations where programs run on the one system, but encounter loader problems on others. *Lint* – **p** causes all external symbols to be mapped to one case and truncated to six characters, providing a worst-case analysis.

Many differences arise in the area of character handling. Characters in most UNIX systems are eight-bit ascii, but they are eight-bit ebcdic on the IBM, and nine bit ascii on Honeywell GCOS. Also, character strings go from high to low bit positions ("left-to-right") on many systems (Ridge, IBM, 68000, 3B20) and low to high ("right-to-left") on others (VAX, PDP-11, Z8000, NS16000). This means that code attempting to construct strings out of character constants, or attempting to use characters as indexes into arrays, must be looked at with great suspicion. *Lint* is of little help here, except to flag multi-character character constants.

Word sizes are different, but this causes less trouble than might be expected when moving from 16-bit words to 32- or 36-bit words. The main problems are likely to arise in shifting or masking. C now supports a bit-field facility, which can be used to write much of this code in a reasonably portable way. Frequently, portability of such code can be enhanced by slight rearrangements in coding style. Many of the incompatibilities seem to have the flavor of writing

```
x &= 0177700 ;
```

to clear the low order six bits of *x*. This suffices on a 16-bit machine, but fails on 32- or 36-bit machines. If the bit field feature cannot be used, the same effect can be obtained by writing

```
x &= ~ 077 ;
```

which will work on all these machines.

The right shift operator is arithmetic shift on the VAX 11/780 and PDP-11, and logical shift on the Ridge 32 and most other machines. To obtain a logical shift on all machines, the left operand can be typed **unsigned**. Characters are considered signed integers on the VAX PDP-11, and unsigned on the other machines. This persistence of the sign bit may be reasonably considered a bug in the PDP-11 hardware which has infiltrated itself into the C language. If there were a good way to discover the programs which would be affected, C could be changed; in any case, *lint* is no help here.

The above discussion may have made the problem of portability seem bigger than it in fact is. The issues involved here are rarely subtle or mysterious, at least to the implementor of the program, although they can involve some work to straighten out. The most serious bar to the portability of UNIX† system utilities has been the inability to mimic essential UNIX system

†UNIX is a Trademark of Bell Laboratories.

functions on the other systems. On many systems **other than Ridge**, the inability to seek to a random character position in a text file, or to establish a pipe between processes, has involved more rewriting and debugging than any of the differences in C compilers. On the other hand, *lint* has been very helpful in moving the UNIX operating system and associated utility programs to other machines.

Shutting Lint Up

There are occasions when the programmer is smarter than *lint*. There may be valid reasons for "illegal" type casts, functions with a variable number of arguments, etc. Moreover, as specified above, the flow of control information produced by *lint* often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Thus, some way of communicating with *lint*, typically to shut it up, is desirable.

The form which this mechanism should take is not at all clear. New keywords would require current and old compilers to recognize these keywords, if only to ignore them. This has both philosophical and practical problems. New preprocessor syntax suffers from similar problems.

What was finally done was to cause a number of words to be recognized by *lint* when they were embedded in comments. This required minimal preprocessor changes; the preprocessor just had to agree to pass comments through to its output, instead of deleting them as had been previously done. Thus, *lint* directives are invisible to the compilers, and the effect on systems with the older preprocessors is merely that the *lint* directives don't work.

The first directive is concerned with flow of control information; if a particular place in the program cannot be reached, but this is not apparent to *lint*, this can be asserted by the directive

```
/* NOTREACHED */
```

at the appropriate spot in the program. Similarly, if it is desired to turn off strict type checking for the next expression, the directive

```
/* NOSTRICT */
```

can be used; the situation reverts to the previous default after the next expression. The **-v** flag can be turned on for one function by the directive

```
/* ARGSUSED */
```

Complaints about variable number of arguments in calls to a function can be turned off by the directive

```
/* VARARGS */
```

preceding 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 by following the VARARGS keyword immediately with a digit giving the number of arguments which should be checked; thus,

```
/* VARARGS2 */
```

will cause the first two arguments to be checked, the others unchecked. Finally, the directive

```
/* LINTLIBRARY */
```

at the head of a file identifies this file as a library declaration file; this topic is worth a section by itself.

Library Declaration Files

Lint accepts certain library directives, such as

```
-lm
```

and tests the source files for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library directives. These files all begin with the directive

```
/* 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 directives can be used to specify features of the library functions.

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, draw no complaints. *Lint* does not simulate a full library search algorithm, and complains if the source files contain a redefinition of a library routine (this is a feature!).

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 program is run. When the **-p** flag is in effect, another file is checked containing descriptions of the standard I/O library routines which are expected to be portable across various machines. The **-n** flag can be used to suppress all library checking.

Bugs, etc.

Lint was a difficult program to write, partially because it is closely connected with matters of programming style, and partially because users usually don't notice bugs which cause *lint* to miss errors which it should have caught. (By contrast, if *lint* incorrectly complains about something that is correct, the programmer reports that immediately!)

A number of areas remain to be further developed. The checking of structures and arrays is rather inadequate; size incompatibilities go unchecked, and no attempt is made to match up structure and union declarations across files. Some stricter checking of the use of the **typedef** is clearly desirable, but what checking is appropriate, and how to carry it out, is still to be determined.

Lint shares the preprocessor with the C compiler. At some point it may be appropriate for a special version of the preprocessor to be constructed which checks for things such as unused macro definitions, macro arguments which have side effects which are not expanded at all, or are expanded more than once, etc.

The central problem with *lint* is the packaging of the information which it collects. There are many options which serve only to turn off, or slightly modify, certain features. There are pressures to add even more of these options.

In conclusion, the use of two programs is efficient: the compiler turns the program source into executable form, and *lint* concentrates on issues of portability, style, and efficiency. Incorrectness and over-conservatism are only annoying, not fatal, so *Lint* can afford to be wrong. The compiler can be fast since it knows that *lint* will correct its deficiencies. Finally, the programmer can concentrate at one stage of the programming process solely on the algorithms, data structures, and correctness of the program, and then later retrofit, with the aid of *lint*, the desirable properties of universality and portability.

See the lint(1) page of the **ROS Reference Manual (9010)** for a summary of the lint options.

Make — for Maintaining Computer Programs

This document is based on a paper by S.I. Feldman of Bell Laboratories, August, 1978.

Introduction

Make mechanizes many activities of program development and maintenance. It is a mechanism for maintaining an up-to-date version of all the component files of a small to medium-size program.

Many files may exist as parts of a larger program. Some may require a macro procesor, others may need compiling with special options by different language compilers, and others may require processing by **yacc** or **lex**. The output code from some steps may have to be loaded with special libraries and tested by certain test scripts. **Make** records the interdependence of files, mechanizes the procedure of figuring out which object modules need recompilation, and memorizes the exact sequence of operations needed to make or exercise a new version of the program.

Once the appropriate information has been established in a file, the simple command

```
make
```

is frequently sufficient to update the involved program files, regardless of the number that have been edited since the last "make". The description file is easy to write and it changes infrequently, making use of the **make** command easier than issuing one of the component commands by hand. The typical cycle of program development is:

think — edit — *make* — test ...

Make does not solve the problems of maintaining multiple source versions or of describing huge programs.

Introductory Examples

make updates a target file by ensuring that all of the files on which it depends exist and are up to date, then creates the target if it has not been modified since its dependents were. *Make* does a depth-first search of the graph of dependences. The date and time of file modification is the key for **make** to determine if it needs updating.

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

```
#include "defs"
```

The following text describes the relationships and operations:

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

x.o y.o :  defs
```

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

```
make
```

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

Make operates using three sources of information: a user-supplied description file (as above), file names and "last-modified" times from the file system, and built-in rules to bridge some of the gaps. In our example, the first line says that *prog* depends on three "*.o*" files. Once these object files are current, the second line describes how to load them to create *prog*. The third line says that *x.o* and *y.o* depend on the file *defs*. From the file system, *make* discovers that there are three "*.c*" files corresponding to the needed "*.o*" files, and uses built-in information on how to generate an object from a source file (*i.e.*, issue a "cc – c" command).

The following long-winded description file is equivalent to the one above, but takes no advantage of *make*'s innate knowledge:

```
prog : x.o y.o z.o
       cc x.o y.o z.o – lS – o prog
x.o : x.c defs
       cc – c x.c
y.o : y.c defs
       cc – c y.c
z.o : z.c
       cc – c z.c
```

If none of the source or object files had changed since the last time *prog* was made, all of the files would be current, and the command

```
make
```

would just announce this fact and stop. If, however, the *defs* file had been edited, *x.c* and *y.c* (but not *z.c*) would be recompiled, and then *prog* would be created from the new "*.o*" files. If only the file *y.c* had changed, only it would be recompiled, but it would still be necessary to reload *prog*.

If no target name is given on the *make* command line, the first target mentioned in the description is created; otherwise the specified targets are made. The command

```
make x.o
```

would recompile *x.o* if *x.c* or *defs* had changed.

If the file exists after the commands are executed, its time of last modification is used in further decisions; otherwise the current time is used. It is often quite useful to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of *make*'s ability to generate files and substitute macros. Thus, an entry "save" might be included to copy a certain set of files, or an entry "cleanup" might be used to throw away unneeded intermediate files. In other cases one may maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

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

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

The last two invocations are identical. $$ is a dollar sign. All of these macros are assigned values during input, as shown below. Four special macros change values during the execution of the command: $*, $@, $?, and $<. They will be discussed later. The following fragment shows the use:

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

The command

```
make
```

loads the three object files with the *lc* library, which is automatically included by the cc(1) command. The command

```
make "LIBES= -ll -lm"
```

loads them with both the Lex ("-ll") and the Standard ("-lc") libraries, since macro definitions on the command line override definitions in the description.

The following sections detail the form of description files and the command line, and discuss options and built-in rules in more detail.

Description Files and Substitutions

A description file contains three types of information: macro definitions, dependency information, and executable commands. There is also a comment convention: all characters after a sharp (#) are ignored, as is the sharp itself. Blank lines and lines beginning with a sharp are totally ignored. If a non-comment line is too long, it can be continued using a backslash. If the last character of a line is a backslash, the backslash, newline, and following blanks and tabs are replaced by a single blank.

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

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

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as value. Macro definitions may also appear on the *make* command line (see below).

Other lines give information about target files. The general form of an entry is:

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

Items inside brackets may be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. (Shell metacharacters "*" and "?" are expanded.) A command is any string of characters not including a sharp (except in quotes) or newline. Commands may

appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line.

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type.

1. For the usual single-colon case, at most one of these dependency lines may have a command sequence associated with it. If the target is out of date with any of the dependents on any of the lines, and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise a default creation rule may be invoked.
2. In the double-colon case, a command sequence may be associated with each dependency line; if the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. This detailed form is of particular value in updating archive-type files.

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

Before issuing any command, certain macros are set. $@ is set to the name of the file to be "made". $? is set to the string of names that were found to be younger than the target. If the command was generated by an implicit rule (see below), $< is the name of the related file that caused the action, and $* is the prefix shared by the current and the dependent file names.

If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name ".DEFAULT" are used. If there is no such name, *make* prints a message and stops.

Command Usage

See the make(1) pages of the ROS Reference Manual for command syntax and options.

Implicit Rules

The *make* program uses a table of interesting suffixes and a set of transformation rules to supply default dependency information and implied commands. (The Appendix describes these tables and means of overriding them.) The default suffix list is:

.o	Object file
.c	C source file
.e	Efl source file
.r	Ratfor source file
.f	Fortran source file
.s	Assembler source file
.y	Yacc-C source grammar
.yr	Yacc-Ratfor source grammar
.ye	Yacc-Efl source grammar
.l	Lex source grammar

The following diagram summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is

named in the description.

.o

.c *.r* *.e* *.f* *.s* *.y* *.yr* *.ye* *.l* *.d*

.y *.l* *.yr* *.ye*

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

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

make CC=newcc

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

make "CFLAGS=-O"

causes the optimizing C compiler to be used.

Example

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

```
# Description file for the Make command
FILES = Makefile version.c defs main.c doname.c misc.c files.c dosys.cgram.y lex.c gcos.c
OBJECTS = version.o main.o doname.o misc.o files.o dosys.o gram.o
LIBES=
LINT = lint - p
CFLAGS = - O
make: $(OBJECTS)
        cc $(CFLAGS) $(OBJECTS) $(LIBES) - o make
        size make
$(OBJECTS): defs
gram.o: lex.c
cleanup:
        -rm *.o gram.c
        -du
install:
        @ size make /usr/bin/make
        cp make /usr/bin/make ; rm make
print: $(FILES)     # print recently changed files
        pr $? |$P
        touch print
test:
        make - dp |grep - v TIME >1zap
        /usr/bin/make - dp |grep - v TIME >2zap
        diff 1zap 2zap
        rm 1zap 2zap
lint : dosys.c doname.c files.c main.c misc.c version.c gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c version.c gram.c
        rm gram.c
arch:
        ar uv /sys/source/s2/make.a $(FILES)
```

Make usually prints out each command before issuing it. The following output results from typing the simple command

```
make
```

in a directory containing only the source and description file:

```
cc - c version.c
cc - c main.c
cc - c doname.c
cc - c misc.c
cc - c files.c
cc - c dosys.c
yacc gram.y
mv y.tab.c gram.c
cc - c gram.c
cc version.o main.o doname.o misc.o files.o dosys.o gram.o - o make
13188+ 3348+ 3044 = 19580b = 046174b
```

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

of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The "print" entry prints only the files that have been changed since the last "make print" command. A zero-length file *print* is maintained to keep track of the time of the printing; the $? macro in the command line then picks up only the names of the files changed since *print*

Suggestions and Warnings

The most common difficulties arise from *make*'s specific meaning of dependency. If file *x.c* has a "#include "defs"" line, then the object file *x.o* depends on *defs*; the source file *x.c* does not. (If *defs* is changed, it is not necessary to do anything to the file *x.c*, while it is necessary to recreate *x.o*.)

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

```
make – n
```

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

```
make – ts
```

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

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

Appendix. Suffixes and Transformation Rules

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

The list of suffixes is actually the dependency list for the name ".SUFFIXES"; *make* looks for a file with any of the suffixes on the list. If such a file exists, and if there is a transformation rule for that combination, *make* acts as described earlier. The transformation rule names are the concatenation of the two suffixes. The name of the rule to transform a "*.r*" file to a "*.o*" file is thus "*.r.o*". If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule ".r.o" is used. If a command is generated by using one of these suffixing rules, the macro $* is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro $< is the name of the dependent that caused the action.

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

The following is an excerpt from the default rules file:

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

LEX — A Lexical Analyzer Generator

This document is based on a paper by M.E. LEsk and E. Schmidt of Bell Laboratories. It supplements the lex(1) pages of the **ROS Reference Manual (9010)**.

Table of Contents

1. Introduction.

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages." Just as general purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C.

Lex turns the user's expressions and actions (called *source)* into the host general-purpose language; the generated program is named *yylex.* The *yylex* program recognizes expressions in a stream (called *input)* and performs the specified actions for each expression as it is detected:

Source -> *lex* -> yylex

Input -> *yylex* -> output

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

```
%%
[ \t]+$    ;
```

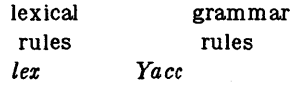
is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates "one or more ..."; and the $ indicates "end of line," as in QED. No action is specified, so the program generated by Lex (yylex) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[ \t]+$    ;
[ \t]+     printf(" ");
```

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs

whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface Lex and Yacc [3]. Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context-free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of Lex and Yacc is often appropriate. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is:



Input -> *yylex* -> *yyparse* -> parsed input

Additional programs, written by other generators or by hand, can be added easily to programs written by Lex.

Yacc users will realize that the name *yylex* is what Yacc expects its lexical analyzer to be named, so that the use of this name by Lex simplifies interfacing.

Lex generates a deterministic finite automaton from the regular expressions in the source [4]. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a Lex program to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of re-scanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by Lex.

In the program written by Lex, the user's fragments (representing the *actions* to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

Lex is not limited to source which can be interpreted on the basis of one character look-ahead. For example, if there are two rules, one looking for *ab* and another for *abcdefg*, and the input stream is *abcdefh*, Lex will recognize *ab* and leave the input pointer just before *cd*. . . Such backup is more costly than the processing of simpler languages.

2. Lex Source.

The general format of Lex source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the *rules* represent the user's control decisions; they are a table, in which the left column contains *regular expressions* (see section 3) and the right column contains *actions*, program fragments to be executed when the expressions are recognized. Thus an individual rule might appear

```
integer    printf("found keyword INT");
```

to look for the string *integer* in the input stream and print the message "found keyword INT" whenever it appears. In this example, the host procedural language is C, and the C library function *printf* prints the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. Lex rules such as

```
colour      printf("color");
mechanise   printf("mechanize");
petrol      printf("gas");
```

would be a start. These rules are not quite enough, since the word *petroleum* would become

gaseum; a way of dealing with this will be described later.

3. Lex Regular Expressions.

The definitions of regular expressions are very similar to those in QED [5]. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

integer

matches the string *integer* wherever it appears and the expression

a57D

looks for the string *a57D*.

Operators. The operator characters are

" \ [ ] ^ - ? . * + |( ) $ / { } % < >

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

xyz"+ +"

matches the string *xyz+ +* when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

"xyz+ +"

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to Lex lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

xyz\+ \+

which is another, less readable, equivalent of the above expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

Character classes. Classes of characters can be specified using the operator pair [ ]. The construction *[abc]* matches a single character, which may be *a*, *b*, or *c*. Within square brackets, most operator meanings are ignored. Only three characters are special: these are \ - and ^. The - character indicates ranges. For example,

[a- z0- 9< >_]

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using - between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. (E.g., [0- z] in ASCII is many more characters than it is in EBCDIC). If it is desired to include the character - in a character class, it should be first or last; thus

[- + 0- 9]

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus

[^abc]

matches all characters except a, b, or c, including all special or control characters; or

[^a- zA- Z]

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

Arbitrary character. To match almost any character, the operator character

.

is the class of all characters except newline. Escaping into octal is possible although nonportable:

[\40- \176]

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

Optional expressions. The operator ? indicates an optional element of an expression. Thus

ab?c

matches either *ac* or *abc*.

Repeated expressions. Repetitions of classes are indicated by the operators * and + .

```
a*
```

is any number of consecutive *a* characters, including zero; while

```
a+
```

is one or more instances of *a*. For example,

```
[a- z]+
```

is all strings of lower case letters. And

```
[A- Za- z][A- Za- z0- 9]*
```

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping. The operator | indicates alternation:

```
(ab|cd)
```

matches either *ab* or *cd*. Note that parentheses are used for grouping, although they are not necessary on the outside level;

```
ab|cd
```

would have sufficed. Parentheses can be used for more complex expressions:

```
(ab|cd+)?(ef)*
```

matches such strings as *abefef, efefef, cdef,* or *cddd*; but not *abc, abcd,* or *abcdef.*

Context sensitivity. Lex will recognize a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of ^, complementation of character classes, since that only applies within the [ ] operators. If the very last character is $, the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the / operator character, which indicates trailing context. The expression

```
ab/cd
```

matches the string *ab*, but only if followed by *cd*. Thus

```
ab$
```

is the same as

```
ab/\n
```

Left context is handled in Lex by *start conditions* as explained in section 10. If a rule is only to be executed when the Lex automaton interpreter is in start condition *x*, the rule should be prefixed by

```
<x>
```

using the angle bracket operator characters. If we considered "being at the beginning of a line" to be start condition *ONE*, then the ^ operator would be equivalent to

```
<ONE>
```

Start conditions are explained more fully later.

Repetitions and Definitions. The operators {} specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

```
{digit}
```

looks for a predefined string named *digit* and inserts it at that point in the expression. The definitions are given in the first part of the Lex input, before the rules. In contrast,

```
a{1,5}
```

looks for 1 to 5 occurrences of *a*.

Finally, initial %% is special, being the separator for Lex source segments.

4. Lex Actions.

When an expression written as above is matched, Lex executes the corresponding action. This section describes some features of Lex which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the Lex user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When Lex is being used with Yacc, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, ; as an action causes this result. A frequent rule is

```
[ \t\n]    ;
```

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is the action character | which indicates that the action for this rule is the action for the next rule.

The previous example could also have been written

```
" "
"\t"
"\n"
```

with the same result, although in different style. The quotes around \n and \t are not required.

In more complex actions, the user will often want to know the actual text that matched some expression like *[a- z]+*. Lex leaves this text in an external character array named *yytext*. Thus, to print the name found, a rule like

```
[a- z]+     printf("%s", yytext);
```

will print the string in *yytext*. The C function *printf* accepts a format argument and data to be printed; in this case, the format is "print string" (% indicating data conversion, and *s* indicating string type), and the data are the characters in *yytext*. So this just places the matched string on the output. This action is so common that it may be written as ECHO:

```
[a- z]+     ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches *read* it will normally match the instances of *read* contained in *bread* or *readjust*; to avoid this, a rule of the form *[a- z]+* is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence Lex also provides a count *yyleng* of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might write

```
[a- zA- Z]+     {words++; chars += yyleng;}
```

which accumulates in *chars* the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng- 1]
```

Occasionally, a Lex action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore()* can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in *yytext*. Second, *yyless (n)* may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument *n* indicates the number of characters in *yytext* to be retained. Further characters previously matched are returned to the input. This provides the same sort of look-ahead offered by the / operator, but in a different form.

Example: Consider a language which defines a string as a set of characters between quotation (") marks, and provides that to include a " in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write

```
\"[^"]*    {
           if (yytext[yyleng- 1] == '\\')
               yymore();
           else
               ... normal user processing
           }
```

which will, when faced with a string such as *"abc\"def"* first match the five characters *"abc*; then the call to *yymore()* will cause the next part of the string, *"def*, to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labeled "normal processing".

The function *yyless()* might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of "=- a". Suppose it is desired to treat this as "=- a" but print a message. A rule might be

```
=- [a- zA- Z]    {
                 printf("Operator (=-) ambiguous\n");
                 yyless(yyleng- 1);
                 ... action for =- ...
                 }
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as "=-". Alternatively it might be desired to treat this as "= -a". To do this, just return the minus sign as well as the letter to the input:

```
=- [a- zA- Z]    {
                 printf("Operator (=-) ambiguous\n");
                 yyless(yyleng- 2);
                 ... action for = ...
                 }
```

will perform the other interpretation. Note that the expressions for the two cases might more easily be written

```
=- /[A- Za- z]
```

in the first case and

```
=/- [A- Za- z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of "=-3", however, makes

```
=- /[^ \t\n]
```

a still better rule.

In addition to these routines, Lex also permits access to the I/O routines it uses. They are:

1) *input()* which returns the next input character;
2) *output(c)* which writes the character *c* on the output; and
3) *unput(c)* pushes the character *c* back onto the input stream to be read later by *input()*.

By default these routines are provided as macro definitions, but the user can override them and supply private versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by *input* must mean end of file; and the relationship between *unput* and *input* must be retained or the Lex look~ahead will not work. Lex does not look ahead at all if it does not have to, but every rule ending in + * ? or $ or containing / implies look~ahead. Look~ahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion of the character set used by Lex. The standard Lex library imposes a 100 character limit on backup.

Another Lex library routine that the user will sometimes want to redefine is *yywrap()* which is called whenever Lex reaches an end-of-file. If *yywrap* returns a 1, Lex continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a *yywrap* which arranges for new input and returns 0. This instructs Lex to continue processing. The default *yywrap* always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule which recognizes end-of-file; the only access to this condition is through *yywrap*. In fact, unless a private version of *input()* is supplied a file containing nulls cannot be handled, since a value of 0 returned by *input* is taken to be end-of-file.

5. Ambiguous Source Rules.

Lex can handle ambiguous specifications. When more than one expression can match the current input, Lex chooses as follows:

1) The longest match is preferred.
2) Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer     keyword action ...;
[a- z]+     identifier action ...;
```

to be given in that order. If the input is *integers*, it is taken as an identifier, because *[a- z]+* matches 8 characters while *integer* matches only 7. If the input is *integer*, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g. *int*) will not match the expression *integer* and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like .* dangerous. For example,

```
'.*'
```

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

```
'first' quoted string here, 'second' here
```

the above expression will match

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule is of the form

```
'[^'\n]*'
```

which, on the above input, will stop after *'first'*. The consequences of errors like this are mitigated by the fact that the . operator will not match newline. Thus expressions like .* stop on the current line. Don't try to defeat this with expressions like *[.\n]+* or equivalents; the Lex generated program will try to read the entire input file, causing internal buffer overflows.

Note that Lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both *she* and *he* in an input text. Some Lex rules to do this might be

```
she     s++;
he      h++;
\n      |
.       ;
```

where the last two rules ignore everything besides *he* and *she*. Remember that . does not include newline. Since *she* includes *he*, Lex will normally *not* recognize the instances of *he* included in *she*, since once it has passed a *she* those characters are gone.

Sometimes the user would like to override this choice. The action REJECT means "go do the next alternative." It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of *he*:

```
she     {s++; REJECT;}
he      {h++; REJECT;}
\n      |
        ;
```

these rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that *she* includes *he* but not vice versa, and omit the REJECT action on *he*; in other cases, however, it would not be possible a priori to tell which input characters were in both classes.

Consider the two rules

```
a[bc]+      { ... ; REJECT;}
a[cd]+      { ... ; REJECT;}
```

If the input is *ab*, only the first rule matches, and on *ad* only the second matches. The input string *accb* matches the first rule for four characters and then the second rule for three characters. In contrast, the input *accd* agrees with the second rule for four characters and then the first rule for three.

In general, REJECT is useful whenever the purpose of Lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word *the* is considered to contain both *th* and *he*. Assuming a two-dimensional array named *digram* to be incremented, the appropriate source is

```
%%
[a-z][a-z]    {digram[yytext[0]][yytext[1]]++; REJECT;}
\n            ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

6. Lex Source Definitions.

Remember the format of the Lex source:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far only the rules have been described. The user needs additional options, though, to define variables for use in his program and for use by Lex. These can go either in the definitions section or in the rules section.

Remember that Lex is turning the rules into a program. Any source not intercepted by Lex is copied into the generated program. There are three classes of such things.

1) Any line which is not part of a Lex rule or action which begins with a blank or tab is copied into the Lex generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by Lex which contains the actions. This material must look like program fragments, and should precede the first Lex rule.

   As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the Lex source or the generated code. The comments should follow the host language convention.

2) Anything included between lines containing only %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3) Anything after the third %% delimiter, regardless of formats, etc., is copied out after the Lex output.

Definitions intended for Lex are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and begining in column 1, is assumed to define Lex substitution strings. The format of such lines is

name translation

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one

blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D                       [0-9]
E                       [DEde][-+]?{D}+
%%
{D}+                    printf("integer");
{D}+"."{D}*({E})?       |
{D}*"."{D}+({E})?       |
{D}+{E}
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a Fortran expression such as *35.EQ.I*, which does not contain a real number, a context-sensitive rule such as

```
[0-9]+/"."EQ    printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within Lex itself for larger source programs. These possibilities are discussed below under "Summary of Source Format," section 12.

7. Usage.

There are two steps in compiling a Lex source program. First, the Lex source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of Lex subroutines. The generated program is on a file named *lex.yy.c*. The I/O library is defined in terms of the C standard library [6].

The C programs generated by Lex are slightly different on OS/370, because the OS compiler is less powerful than the UNIX or GCOS compilers, and does less at compile time. C programs generated on GCOS and UNIX are the same.

UNIX. The library is accessed by the loader flag *-ll*. So an appropriate set of commands is

```
lex source cc lex.yy.c -ll
```

The resulting program is placed on the usual file *a.out* for later execution. To use Lex with Yacc see below. Although the default Lex I/O routines use the C standard library, the Lex automata themselves do not do so; if private versions of *input, output* and *unput* are given, the library can be avoided.

8. Lex and Yacc.

If you want to use Lex with Yacc, note that what Lex writes is a program named *yylex()*, the name required by Yacc for its analyzer. Normally, the default main program on the Lex library calls this routine, but if Yacc is loaded, and its main program is used, Yacc will call *yylex()*. In this case each Lex rule should end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to Yacc's names for tokens is to compile the Lex output file as part of the Yacc output file by placing the line

```
# include "lex.yy.c"
```

in the last section of Yacc input. If the grammar were named "good" and the lexical rules were named "better", the system command sequence can be:

```
yacc good
lex better
cc y.tab.c -ly -ll
```

The Yacc library (-ly) should be loaded before the Lex library, to obtain a main program which invokes the Yacc parser. The generations of Lex and Yacc programs can be done in either order.

9. Examples.

As a trivial problem, consider copying an input file while adding 3 to every positive number divisible by 7. Here is a suitable Lex source program

```
%%
                int k;
[0-9]+          {
                k = atoi(yytext);
                if (k%7 == 0)
                    printf("%d", k+3);
                else
                    printf("%d",k);
                }
```

to do just that. The rule [0-9]+ recognizes strings of digits; *atoi* converts the digits to binary and stores the result in *k*. The operator % (remainder) checks whether *k* is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objectionable that this program will alter such input items as *49.63* or *X7*. Furthermore, it

increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

```
%%
                        int k;
-?[0-9]+                {
                        k = atoi(yytext);
                        printf("%d", k%7 == 0 ? k+3 : k);
                        }
-?[0-9.]+               ECHO;
[A-Za-z][A-Za-z0-9]+    ECHO;
```

Numerical strings containing a "." or preceded by a letter will be picked up by one of the last two rules, and not changed. The *if-else* has been replaced by a C conditional expression to save space; the form *a?b:c* means "if *a* then *b* else *c*".

For an example of statistics gathering, here is a program which histograms the lengths of words, where a word is defined as a string of letters.

```
        int lengs[100];
%%
[a-z]+  lengs[yyleng]++;
.       |
\n      ;
%%
yywrap()
{
int i;
printf("Length No. words\n");
for(i=0; i<100; i++)
    if (lengs[i] > 0)
        printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement *return(1);* indicates that Lex is to perform wrapup. If *yywrap* returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a *yywrap* that never returns true causes an infinite loop.

As a larger example, here are some parts of a program written by N. L. Schryer to convert double precision Fortran to single precision Fortran. Because Fortran does not distinguish upper and lower case letters, this routine begins by defining a set of classes including both cases of each letter:

```
a       [aA]
b       [bB]
c       [cC]
...
z       [zZ]
```

An additional class recognizes white space:

```
W       [ \t]*
```

The first rule changes "double precision" to "real", or "DOUBLE PRECISION" to "REAL".

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
printf(yytext[0]=='d'? "real" : "REAL");
}
```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
^"     "[^0]    ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as "beginning of line, then five blanks, then anything but blank or zero." Note the two different meanings of ^. There follow some rules to change double precision constants to ordinary floating constants.

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+     |
[0-9]+{W}"."{W}{d}{W}[+-]?{W}[0-9]+     |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+     {
    /* convert constants */
    for(p=yytext; *p != 0; p++)
        {
        if (*p == 'd' | *p == 'D')
            *p=+ 'e'- 'd';
        ECHO;
        }
```

After the floating point constant is recognized, it is scanned by the *for* loop to find the letter *d* or *D*. The program than adds *'e'- 'd'*, which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. There follow a series of names which must be respelled to remove their initial *d*. By using the array *yytext* the same action suffices for all the names (only a sample of a rather long list is given here).

```
    {d}{s}{i}{n}        |
    {d}{c}{o}{s}        |
    {d}{s}{q}{r}{t}     |
    {d}{a}{t}{a}{n}     |
    ...
    {d}{f}{l}{o}{a}{t}  printf("%s",yytext+1);
```

Another list of names must have initial *d* changed to initial *a*:

```
    {d}{l}{o}{g}        |
    {d}{l}{o}{g}10      |
    {d}{m}{i}{n}1       |
    {d}{m}{a}{x}1       {
                        yytext[0] =+ 'a' - 'd';
```

```
        ECHO;
        }
```

And one routine must have initial *d* changed to initial *r*:

```
{d}1{m}{a}{c}{h}    {yytext[0] =+ 'r' - 'd';
```

To avoid such names as *dsinx* being detected as instances of *dsin*, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A-Za-z][A-Za-z0-9]*    |
[0-9]+                  |
\n                      |
.                       ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in Fortran or with the use of keywords as identifiers.

10. Left Context Sensitivity.

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The ^ operator, for example, is a prior context operator, recognizing immediately preceding left context just as $ recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another, the use of *start conditions* on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules which recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since Lex is not involved at all. It may be more convenient, however, to have Lex remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when Lex is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word *magic* to *first* on every line which began with the letter *a*, changing *magic* to *second* on every line which began with the letter *b*, and changing *magic* to *third* on every line which began with the letter *c*. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
        int flag;
%%
^a      {flag = 'a'; ECHO;}
^b      {flag = 'b'; ECHO;}
^c      {flag = 'c'; ECHO;}
\n      {flag = 0 ; ECHO;}
magic   {
        switch (flag)
        {
        case 'a': printf("first"); break;
        case 'b': printf("second"); break;
        case 'c': printf("third"); break;
        default: ECHO; break;
        }
        }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to Lex in the definitions section with a line reading

```
%Start    name1 name2 ...
```

where the conditions may be named in any order. The word *Start* may be abbreviated to *s* or *S*. The conditions may be referenced at the head of a rule with the <> brackets:

```
<name1>expression
```

is a rule which is only recognized when Lex is in the start condition *name1*. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to *name1*. To resume the normal state,

```
BEGIN 0;
```

resets the initial condition of the Lex automaton interpreter. A rule may be active in several start conditions:

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the

<> prefix operator is always active.

The same example as before can be written:

```
%START AA BB CC
%%
^a                  {ECHO; BEGIN AA;}
^b                  {ECHO; BEGIN BB;}
^c                  {ECHO; BEGIN CC;}
\n                  {ECHO; BEGIN 0;}
<AA>magic           printf("first");
<BB>magic           printf("second");
<CC>magic           printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but Lex does the work rather than the user's code.

11. Character Set.

The programs generated by Lex handle character I/O only through the routines *input, output,* and *unput.* Thus the character representation provided in these routines is accepted by Lex and employed to return values in *yytext.* For internal use a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter *a* is represented as the same form as the character constant *'a'*. If this interpretation is changed, by providing I/O routines which translate the characters, Lex must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only "%T". The table contains lines of the form

{integer} {character string}

which indicate the value associated with each character. Thus the next example

```
%T
 1      Aa
 2      Bb
...
26      Zz
27      \n
28      +
29      -
30      0
31      1
...
39      9
%T
```

Sample character table.

maps the lower and upper case letters together into the integers 1 through 26, newline into 27, + and - into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear either in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

12. Summary of Source Format.

The general form of a Lex source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of

1) Definitions, in the form "name space translation".
2) Included code, in the form "space code".
3) Included code, in the form

```
%{
code
%}
```

4) Start conditions, given in the form

```
%S name1 name2 ...
```

5) Character set tables, in the form

```
%T
number space character-string
...
%T
```

6) Changes to internal array sizes, in the form

%x nnn

where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

Letter	Parameter
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form "expression action" where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in Lex use the following operators:

x the character "x"

"x"	an "x", even if x is an operator.
\x	an "x", even if x is an operator.
[xy]	the character x or y.
[x- z]	the characters x, y or z.
[^x]	any character but x.
.	any character but newline.
^x	an x at the beginning of a line.
<y>x	an x when Lex is in start condition y.
x$	an x at the end of a line.
x?	an optional x.
x*	0,1,2, ... instances of x.
x+	1,2,3, ... instances of x.
x|y	an x or a y.
(x)	an x.
x/y	an x but only if followed by y.
{xx}	the translation of xx from the definitions section.
x{m,n}	*m* through *n* occurrences of x

13. Caveats and Bugs.

There are pathological expressions which produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT executed, the user must not have used ***unput*** to change the characters forthcoming from the input stream. This is the only restriction on the user's ability to manipulate the not-yet-processed input.

Yacc: Yet Another Compiler-Compiler

This document is based on a paper by Stephen C. Johnson of Bell Laboratories.

0: Introduction

Yacc is a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a *parser*, calls the user-supplied low-level input routine (the *lexical analyzer*) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called *grammar rules*; when one of these rules has been recognized, then user code supplied for this rule, an *action*, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in C, and many of its syntax conventions follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ',' year ;
```

Here, *date*, *month_name*, *day*, and *year* represent structures of interest in the input process; presumably, *month_name*, *day*, and *year* are defined elsewhere. The comma "," is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

```
July 4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a *terminal symbol*, while the structure recognized by the parser is called a *nonterminal symbol*. To avoid confusion, terminal symbols are usually called *tokens*.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;

    . . .

month_name : 'D' 'e' 'c' ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and *month_name* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a *month_name* was seen; in this case, *month_name* would be a token.

Literal characters such as "," must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is easy to add to the above example the rule

```
date : month '/' day '/' year ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be "slipped in" to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data reduced, but the bad data can be found more quickly. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While Yacc cannot handle all possible specifications, its power compares favorably with similar systems; the constructions which are difficult for Yacc are often difficult for humans. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program development.

The theory underlying Yacc has been described elsewhere. [Aho Johnson Surveys LR Parsing] [Aho Johnson Ullman Ambiguous Grammars] [Aho Ullman Principles Compiler Design] Yacc has been extensively used in numerous practical applications, including *lint* [Johnson Lint], the Portable C Compiler [Johnson Portable Compiler Theory] and a system for typesetting mathematics [Kernighan Cherry typesetting system CACM].

The next several sections describe the basic process of preparing a Yacc specification: **Section 1** preparation of grammar rules, **Section 2** preparation of the user-supplied actions associated with these rules, and **Section 3** preparation of lexical analyzers. **Section 4** operation of the parser. **Section 5** reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. **Section 6** a simple mechanism for handling operator precedences in arithmetic expressions. **Section 7** error detection and recovery. **Section 8** the operating environment and special features of the parsers Yacc produces. **Section 9** suggestions to improve the style and efficiency of specifications. **Section 10** advanced topics, and **Section 11** acknowledgements. **Appendix A** a brief example, **Appendix B** a summary of the Yacc input syntax. **Appendix C** an example using advanced features of Yacc. **Appendix D** mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of Yacc.

1: Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations, (grammar) rules,* and *programs.* The sections are separated by double percent "*%%*" marks. (The percent "*%*" is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ".", underscore "_", and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes "'". As in C, the backslash "\" is an escape character within literals, and all the C escapes are recognized. Thus

'\n'	newline
'\r'	return
'\''	single quote "'"
'\\'	backslash "\"
'\t'	tab
'\b'	backspace
'\f'	form feed
'\xxx'	"xxx" in octal

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar "|" can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A    :    B C D  ;
A    :    E F  ;
A    :    G  ;
```

can be given to Yacc as

```
A    :    B C D
     |    E F
     |    G
     ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty : ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token  name1 name2 . . .
```

in the declarations section. (See Sections 3 , 5, and 6 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the *start symbol*, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start  symbol
```

The end of the input to the parser is signaled by a special token, called the *endmarker*. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it *accepts* the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as "end-of-file" or "end-of-record".

2: Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces "{" and "}". For example,

```
A       :       '(' B ')'
                        {       hello( 1, "abc" ); }
```

and

```
XXX     :       YYY ZZZ
                        {       printf("a message\n");
                                flag = 25;  }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol "dollar sign" "$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable "$$" to some value. For example, an action that does nothing but return the value 1 is

```
{ $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A     :     B C D  ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr  :     '(' expr ')' ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by

```
expr  :     '(' expr ')'        { $$ = $2 ; }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A     :     B  ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A     :     B
                  { $$ = 1; }
            C
                  { x = $2;  y = $3; }
      ;
```

the effect is to set *x* to 1, and *y* to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
$ACT  :     /* empty */
                  { $$ = 1; }
      ;

A     :     B $ACT C
                  { x = $2;  y = $3; }
      ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node*, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

```
expr  :     expr '+' expr
                  { $$ = node( '+', $1, $3 ); }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks "%{" and "%}". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{  int variable = 0;  %}
```

could be placed in the declarations section, making *variable* accessible to all of the actions. The Yacc parser uses only names beginning in "yy"; the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.

3: Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the "# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
        ...
        c = getchar();
        ...
        switch( c ) {
                ...
        case '0':
        case '1':
          ...
        case '9':
                yylval = c- '0';
                return( DIGIT );
                ...
                }
        ...
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names *if* or *while* will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name *error* is reserved for error handling, and should not be used naively (see Section 7).

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are

assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal *in the declarations section* can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the *Lex* program developed by Mike Lesk. Lesk Lex These lexical analyzers are designed to work in close harmony with Yacc parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. Lex can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

4: How the Parser Works

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *lookahead* token). The *current state* is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called *shift, reduce, accept,* and *error.* A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yylex* to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF      shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The *reduce* action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ".") is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
reduce 18
```

refers to *grammar rule* 18, while the action

```
IF      shift 34
```

refers to *state* 34.

Suppose the rule being reduced is

```
A       :       x y z   ;
```

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing *x*, *y*, and *z*, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
A       goto 20
```

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action "turns back the clock" in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable *yylval* is copied onto the value stack. After the return from the user code, the reduction is carried out. When the *goto* action is done, the external variable *yyval* is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 7.

It is time for an example! Consider the specification

```
%token DING DONG DELL
%%
rhyme :     sound place
      ;
sound :     DING DONG
      ;
place :     DELL
      ;
```

When Yacc is invoked with the – v option, a file called *y.output* is produced, with a human-readable description of the parser. The *y.output* file corresponding to the above

grammar (with some statistics stripped off the end) is:

```
state 0
        $accept : _rhyme $end

        DING shift 3
        . error

        rhyme  goto 1
        sound  goto 2

state 1
        $accept :  rhyme_$end

        $end accept
        . error

state 2
        rhyme :  sound_place

        DELL shift 5
        . error

        place  goto 4

state 3
        sound  :  DING_DONG

        DONG shift 6
        . error

state 4
        rhyme :  sound place_   (1)

        .  reduce 1

state 5
        place :  DELL_   (3)

        .  reduce 3

state 6
        sound  :  DING DONG_   (2)

        .  reduce 2
```

In addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character indicates what has been seen, and what is yet to come, in each rule. Suppose the input is

```
DING DONG DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read, becoming the lookahead token. The action in state 0 on *DING* is is "shift 3", so state 3 is pushed onto the

stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read, becoming the lookahead token. The action in state 3 on the token *DONG* is "shift 6", so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
sound : DING DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on *sound*,

```
sound  goto 2
```

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, *DELL*, must be read. The action is "shift 5", so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by "$end" in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as *DING DONG DONG, DING DONG, DING DONG DELL DELL*, etc. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

5: Ambiguity and Conflicts

A set of grammar rules is *ambiguous* if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
expr  :  expr '-' expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

```
expr - expr - expr
```

the rule allows this input to be structured as either

```
( expr - expr ) - expr
```

or as

```
expr - ( expr - expr )
```

(The first is called *left association*, the second *right association*).

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

```
expr - expr - expr
```

When the parser has read the second expr, the input that it has seen:

```
expr - expr
```

matches the right side of the grammar rule above. The parser could *reduce* the input by

applying this rule; after applying the rule; the input is reduced to *expr*(the left side of the rule). The parser would then read the final part of the input:

```
- expr
```

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

```
expr - expr
```

it could defer the immediate application of the rule, and continue reading the input until it had seen

```
expr - expr - expr
```

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving

```
expr - expr
```

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

```
expr - expr
```

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a *shift / reduce conflict*. It may also happen that the parser has a choice of two legal reductions; this is called a *reduce / reduce conflict*. Note that there are never any "Shift/shift" conflicts.

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a "*disambiguating*rule"

Yacc invokes two "disambiguating" rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than Yacc can construct. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of "disambiguating" rules is inappropriate, and leads to an incorrect parser. For this reason, Yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply "disambiguating" rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, Yacc will produce parsers even in the presence of conflicts.

As an example of the power of "disambiguating" rules, consider a fragment from a programming language involving an "if-then-else" construction:

```
stat    :       IF '(' cond ')' stat
        |       IF '(' cond ')' stat ELSE stat
        ;
```

In these rules, *IF* and *ELSE* are tokens, *cond* is a nonterminal symbol describing conditional

(logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the *simple-if* rule, and the second the *if-else* rule.

These two rules form an ambiguous construction, since input of the form

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be structured according to these rules in two ways:

```
IF ( C1 ) {
        IF ( C2 ) S1
        }
ELSE S2
```

or

```
IF ( C1 ) {
        IF ( C2 ) S1
        ELSE S2
        }
```

The second interpretation is the one given in most programming languages having this construct. Each *ELSE* is associated with the last preceding "un-*ELSE*'d" *IF*. In this example, consider the situation where the parser has seen

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the *ELSE*. It can immediately reduce by the simple-if rule to get

```
IF ( C1 ) stat
```

and then read the remaining input,

```
ELSE S2
```

and reduce

```
IF ( C1 ) stat ELSE S2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the *ELSE* may be shifted, *S2* read, and then the right hand portion of

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be reduced by the if-else rule to get

```
IF ( C1 ) stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things – there is a shift/reduce conflict. The application of "disambiguating" rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, *ELSE*, and particular inputs already seen, such as

```
IF ( C1 ) IF ( C2 ) S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of Yacc are best understood by examining the verbose (– **v**) option output file. For example, the output corresponding to the above conflict state might be:

23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

```
stat : IF ( cond ) stat_        (18)
stat : IF ( cond ) stat_ELSE stat

ELSE    shift 45
.       reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF ( cond ) stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is *ELSE*, it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat : IF ( cond ) stat ELSE_stat
```

since the *ELSE* will have been shifted in this state. Back in state 23, the alternative action, described by ".", is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not *ELSE*, the parser reduces by grammar rule 18:

```
stat : IF '(' cond ')' stat
```

Once again, notice that the numbers following "shift" commands refer to other states, while the numbers following "reduce" commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most one states, there will be at most reduce action possible in the state, and this will be the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references Aho Johnson Surveys Parsing Aho Johnson Ullman Deterministic Ambiguous Aho Ullman Principles Design might be consulted; the services of a local guru might also be appropriate.

6: Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate "disambiguating" rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As "disambiguating" rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is

sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. The '+', '-', '*'. and '/' are all left associative, but '+' and '-' have lower precedence than '*' and '/'. The keyword %right describes right associative operators, and the keyword %nonassoc describes operators, like the operator .LT. in Fortran, that may not associate with themselves; thus,

```
A .LT. B .LT. C
```

is illegal in Fortran, and such an operator would be described with the keyword %nonassoc in Yacc. As an example of the behavior of these declarations, the description

```
%right '='
%left '+' '-'
%left '*' '/'

%%

expr    :       expr '=' expr
        |       expr '+' expr
        |       expr '-' expr
        |       expr '*' expr
        |       expr '/' expr
        |       NAME
        ;
```

might be used to structure the input

```
a = b = c*d - e - f*g
```

as follows:

```
a = ( b = ( ((c*d)- e) - (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary '-'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left '+' '-'
%left '*' '/'

%%

expr    :       expr '+' expr
        |       expr '-' expr
        |       expr '*' expr
        |       expr '/' expr
        |       '-' expr        %prec '*'
        |       NAME
        ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

The precedences and associativities are used by Yacc to resolve parsing conflicts; they give rise to "disambiguating" rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two "disambiguating" rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by Yacc. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially "cookbook" fashion, until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

7: Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, Yacc provides a simple, but reasonably general, feature. The token name "error" is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token "error" is

legal. It then behaves as if the token "error" were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat    :       error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat    :       error   ';'
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any "cleanup" action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input   :       error   '\n'  { printf( "Reenter last line: " ); } input
                        {       $$ = $4; }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
input   :       error   '\n'
                        {       yyerrok;
                                printf( "Reenter last line: " );  }
                input
                        {       $$ = $4; }
        ;
```

As mentioned above, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some

sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat    :       error
                        {       resynch();
                                yyerrok ;
                                yyclearin ;     }
        ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

8: The Yacc Environment

When the user inputs a specification to Yacc, the output is a file of C programs, called *y.tab.c* on most systems (due to local file system conventions, the names may differ from installation to installation). The function produced by Yacc is called *yyparse*; it is an integer valued function. When it is called, it in turn repeatedly calls *yylex*, the lexical analyzer supplied by the user (see Section 3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse* returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, *yyparse* returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called *main* must be defined, that eventually calls *yyparse*. In addition, a routine called *yyerror* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using Yacc, a library has been provided with default versions of *main* and *yyerror*. The name of this library is system dependent; on many systems the library is accessed by a –ly argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
        return( yyparse() );
        }
```

and

```
# include <stdio.h>

yyerror(s) char *s; {
        fprintf( stderr, "%s\n", s );
        }
```

The argument to *yyerror* is a string containing an error message, usually the string "syntax error". The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the *main* program is probably supplied by the user (to read arguments, etc.) the Yacc library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

9: Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

a. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of "knowing who to blame when things go wrong."

b. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

c. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.

d. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.

e. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The example in Appendix A is written following this style, as are the examples in the text of this paper (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the Yacc parser encourages so called "left recursive" grammar rules: rules of the form

```
name    :       name  rest_of_rule  ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list    :       item
        |       list  ','  item
        ;
```

and

```
seq     :       item
        |       seq  item
        ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
seq     :       item
        |       item  seq
        ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq     :       /* empty */
        |       seq  item
        ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if Yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
        int dflag;
%}
  ... other declarations ...

%%

prog    :       decls  stats
        ;

decls   :       /* empty */
                        {       dflag = 1; }
        |       decls  declaration
        ;

stats   :       /* empty */
                        {       dflag = 0; }
        |       stats  statement
        ;

    ... other rules ...
```

The flag *dflag* is now 0 when reading statements, and 1 when reading declarations, *except for the first token in the first statement.* This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of "backdoor" approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit the user to use words like "if", which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of Yacc; it is difficult to pass information to the lexical analyzer telling it "this instance of 'if' is a keyword, and that instance is a variable". The user can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved*; that is, be forbidden for use as variable names. There are

powerful stylistic reasons for preferring this, anyway.

Appendix A: A Simple Example

This example gives the complete Yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled "a" through "z", and accepts arithmetic expressions made up of the operators +, -, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a Yacc specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; This job is probably better done by the lexical analyzer.

```
%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS    /* supplies precedence for unary minus */

%%    /* beginning of rules section */

list    :       /* empty */
        |       list stat '\n'
        |       list error '\n'
                        {       yyerrok; }
        ;

stat    :       expr
                        {       printf( "%d\n", $1 ); }
        |       LETTER '=' expr
                        {       regs[$1] = $3; }
        ;

expr    :       '(' expr ')'
                        {       $$ = $2; }
        |       expr '+' expr
                        {       $$ = $1 + $3; }
```

```
        |       expr '-' expr
                        {       $$ = $1 - $3; }
        |       expr '*' expr
                        {       $$ = $1 * $3; }
        |       expr '/' expr
                        {       $$ = $1 / $3; }
        |       expr '%' expr
                        {       $$ = $1 % $3; }
        |       expr '&' expr
                        {       $$ = $1 & $3; }
        |       expr '|' expr
                        {       $$ = $1 | $3; }
        |       '-' expr        %prec UMINUS
                        {       $$ = - $2; }
        |       LETTER
                        {       $$ = regs[$1]; }
        |       number
        ;

number:         DIGIT
                        {       $$ = $1;   base = ($1==0) ? 8 : 10; }
        |       number DIGIT
                        {       $$ = base * $1 + $2; }
        ;

%%      /* start of programs */

yylex() {               /* lexical analysis routine */
          /* returns LETTER for a lower case letter, yylval = 0 through 25 */
          /* return DIGIT for a digit, yylval = 0 through 9 */
          /* all other characters are returned immediately */

        int c;

        while( (c=getchar()) == ' ' ) {/* skip blanks */ }

        /* c is now nonblank */

        if( islower( c ) ) {
                yylval = c - 'a';
                return ( LETTER );
                }
        if( isdigit( c ) ) {
                yylval = c - '0';
                return( DIGIT );
                }
        return( c );
        }
```

Appendix B: Yacc Input Syntax

This Appendix has a description of the Yacc input syntax, as a Yacc specification. Context dependencies, etc., are not considered. Ironically, the Yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS, but never as part of C_IDENTIFIERs.

```
        /* grammar for the input to Yacc */

        /* basic entities */
%token  IDENTIFIER      /*  includes identifiers  and literals */
%token  C_IDENTIFIER    /*   identifier (but not literal) followed by colon   */
%token  NUMBER                  /*   [0-9]+    */

        /* reserved words:  %type ==> TYPE, %left ==> LEFT, etc. */

%token  LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token  MARK  /* the %% mark */
%token  LCURL /* the %{ mark */
%token  RCURL /* the %} mark */

        /* ascii character literals stand for themselves */

%start  spec

%%

spec    :       defs MARK rules tail
        ;

tail    :       MARK { In this action, eat up the rest of the file }
        |       /* empty: the second MARK is optional */
        ;

defs    :       /* empty */
        |       defs def
        ;

def     :       START IDENTIFIER
        |       UNION { Copy union definition to output }
        |       LCURL { Copy C code to output file } RCURL
        |       ndefs rword tag nlist
        ;

rword   :       TOKEN
        |       LEFT
        |       RIGHT
```

```
        |       NONASSOC
        |       TYPE
        ;

tag     :       /* empty: union tag is optional */
        |       '<' IDENTIFIER '>'
        ;

nlist   :       nmno
        |       nlist nmno
        |       nlist ',' nmno
        ;

nmno    :       IDENTIFIER              /* NOTE: literal illegal with %type */
        |       IDENTIFIER NUMBER       /* NOTE: illegal with %type */
        ;

        /* rules section */

rules   :       C_IDENTIFIER rbody prec
        |       rules rule
        ;

rule    :       C_IDENTIFIER rbody prec
        |       '|' rbody prec
        ;

rbody   :       /* empty */
        |       rbody IDENTIFIER
        |       rbody act
        ;

act     :       '{' { Copy action, translate $$, etc. } '}'
        ;

prec    :       /* empty */
        |       PREC IDENTIFIER
        |       PREC IDENTIFIER act
        |       prec ';'
        ;
```

Appendix C: An Advanced Example

This Appendix gives an example of a grammar using some of the advanced features discussed in Section 10. The desk calculator example in Appendix A is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, -, *, /, unary -, and = (assignment), and has 26 floating point variables, "a" through "z". Moreover, it also understands *intervals*, written

(x , y)

where x is less than or equal to y. There are 26 interval valued variables "A" through "Z" that may also be used. The usage is similar to that in Appendix A; assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of Yacc and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as *double*'s. This structure is given a type name, INTERVAL, by using *typedef*. The Yacc value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of Yacc is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g. scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through Yacc: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

```
2.5 + ( 3.5 - 4. )
```

and

```
2.5 + ( 3.5 , 4. )
```

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the "," is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine *atof* performs the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
%{

# include <stdio.h>
# include <ctype.h>

typedef struct interval {
        double lo, hi;
        } INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[ 26 ];
INTERVAL vreg[ 26 ];

%}

%start  lines

%union  {
        int ival;
        double dval;
        INTERVAL vval;
        }

%token <ival> DREG VREG         /* indices into dreg, vreg arrays */

%token <dval> CONST             /* floating point constant */

%type <dval> dexp               /* expression */

%type <vval> vexp               /* interval expression */

        /* precedence information about the operators */

%left   '+' '-'
%left   '*' '/'
%left   UMINUS      /* precedence for unary minus */

%%

lines   :       /* empty */
        |       lines line
        ;

line    :       dexp '\n'
                        {       printf( "%15.8f\n", $1 ); }
        |       vexp '\n'
                        {       printf( "(%15.8f , %15.8f )\n", $1.lo, $1.hi ); }
        |       DREG '=' dexp '\n'
                        {       dreg[$1] = $3; }
        |       VREG '=' vexp '\n'
```

```
                { vreg[$1] = $3; }
        |   error '\n'
                { yyerrok; }
        ;

dexp    :   CONST
        |   DREG
                { $$ = dreg[$1]; }
        |   dexp '+' dexp
                { $$ = $1 + $3; }
        |   dexp '-' dexp
                { $$ = $1 - $3; }
        |   dexp '*' dexp
                { $$ = $1 * $3; }
        |   dexp '/' dexp
                { $$ = $1 / $3; }
        |   '-' dexp    %prec UMINUS
                { $$ = - $2; }
        |   '(' dexp ')'
                { $$ = $2; }
        ;

vexp    :   dexp
                { $$.hi = $$.lo = $1; }
        |   '(' dexp ',' dexp ')'
                {
                $$.lo = $2;
                $$.hi = $4;
                if( $$.lo > $$.hi ){
                        printf( "interval out of order\n" );
                        YYERROR;
                        }
                }
        |   VREG
                { $$ = vreg[$1]; }
        |   vexp '+' vexp
                { $$.hi = $1.hi + $3.hi;
                  $$.lo = $1.lo + $3.lo; }
        |   dexp '+' vexp
                { $$.hi = $1 + $3.hi;
                  $$.lo = $1 + $3.lo; }
        |   vexp '-' vexp
                { $$.hi = $1.hi - $3.lo;
                  $$.lo = $1.lo - $3.hi; }
        |   dexp '-' vexp
                { $$.hi = $1 - $3.lo;
                  $$.lo = $1 - $3.hi; }
        |   vexp '*' vexp
                { $$ = vmul( $1.lo, $1.hi, $3 ); }
        |   dexp '*' vexp
                { $$ = vmul( $1, $1, $3 ); }
        |   vexp '/' vexp
                { if( dcheck( $3 ) ) YYERROR;
                  $$ = vdiv( $1.lo, $1.hi, $3 ); }
```

```
        |       dexp '/' vexp
                        {       if( dcheck( $3 ) ) YYERROR;
                                $$ = vdiv( $1, $1, $3 ); }
        |       '-' vexp        %prec UMINUS
                        {       $$.hi = - $2.lo;   $$.lo = - $2.hi;   }
        |       '(' vexp ')'
                        {       $$ = $2; }
        ;

%%

# define BSZ 50      /* buffer size for floating point numbers */

        /* lexical analysis */

yylex(){
        register c;

        while( (c=getchar()) == ' ' ){ /* skip over blanks */ }

        if( isupper( c ) ){
                yylval.ival = c - 'A';
                return( VREG );
                }
        if( islower( c ) ){
                yylval.ival = c - 'a';
                return( DREG );
                }

        if( isdigit( c ) || c=='.' ){
                /* gobble up digits, points, exponents */

                char buf[BSZ+1], *cp = buf;
                int dot = 0, exp = 0;

                for( ; (cp-buf)<BSZ ; ++cp,c=getchar() ){

                        *cp = c;
                        if( isdigit( c ) ) continue;
                        if( c == '.' ){
                                if( dot++ || exp ) return( '.' );   /* will cause syntax error */
                                continue;
                                }

                        if( c == 'e' ){
                                if( exp++ ) return( 'e' );   /* will cause syntax error */
                                continue;
                                }

                        /* end of number */
                        break;
                        }
                *cp = '\0';
                if( (cp-buf) >= BSZ ) printf( "constant too long: truncated\n" );
```

```
            else ungetc( c, stdin );   /* push back last char read */
            yylval.dval = atof( buf );
            return( CONST );
            }
        return( c );
        }

INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
        /* returns the smallest interval containing a, b, c, and d */
        /* used by *, / routines */
        INTERVAL v;

        if( a>b ) { v.hi = a;   v.lo = b; }
        else { v.hi = b;   v.lo = a; }

        if( c>d ) {
            if( c>v.hi ) v.hi = c;
            if( d<v.lo ) v.lo = d;
            }
        else {
            if( d>v.hi ) v.hi = d;
            if( c<v.lo ) v.lo = c;
            }
        return( v );
        }

INTERVAL vmul( a, b, v ) double a, b;  INTERVAL v; {
        return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
        }

dcheck( v ) INTERVAL v; {
        if( v.hi >= 0. && v.lo <= 0. ){
            printf( "divisor interval contains 0.\n" );
            return( 1 );
            }
        return( 0 );
        }

INTERVAL vdiv( a, b, v ) double a, b;  INTERVAL v; {
        return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
        }
```

Appendix D: Old Features Supported but not Encouraged

This Appendix mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes ''"''.

2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

   The use of multi-character literals is likely to mislead those unfamiliar with Yacc, since it suggests that Yacc is doing a job which must be actually done by the lexical analyzer.

3. Most places where % is legal, backslash "\" may be used. In particular, \\ is the same as %%, \left the same as %left, etc.

4. There are a number of other synonyms:

   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec

5. Actions may also have the form

   ={ . . . }

   and the curly braces can be dropped if the action is a single C statement.

6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

f77 on the Ridge

This document is based on a paper by S.I. Feldman and P.J. Weinberger of Bell Laboratories, and a paper by David L. Wasley of the University of California, Berkeley.

TABLE OF CONTENTS

RUNNING THE f77 COMPILER

f77(1) is a general purpose command for compiling and loading FORTRAN and FORTRAN-related files into an executable module. Based on the suffix of the input files, f77(1) will translate EFL compiler or Ratfor preprocessor source files into FORTRAN, or invoke the C compiler to translate C source files, or the AS(1) assembler to translate assembler source files. Object files will be link-edited.

USAGE

To run the compiler:
f 77 *flags file . . .*
The following file name suffixes are understood:

.f	FORTRAN source file
.F	FORTRAN source file
.e	EFL source file
.r	Ratfor source file
.c	C source file
.s	Assembler source file
.o	Object file

Arguments whose names end with **.f** are taken to be FORTRAN 77 source programs; they are compiled, and each object program is left on the file in the current directory whose name is that of the source with **.o** substituted for **.f**.

Arguments whose names end with **.F** are also taken to be FORTRAN 77 source programs; these are first processed by the C preprocessor before being compiled by f77.

Arguments whose names end with **.r** or **.e** are taken to be Ratfor or EFL source programs, respectively; these are first transformed by the appropriate preprocessor, then compiled by f77.

Arguments whose names end with **.r** or **.e** are taken to be Ratfor or EFL source programs, respectively; these are first transformed by the appropriate preprocessor, then compiled by f77.

In the same way, arguments whose names end with **.c** or **.s** are taken to be C or assembly source programs and are compiled or assembled, producing a **.o** file.

For a description of the f77 compiler options, see the f77(1) page in the ROS Reference Manual.

LANGUAGE EXTENSIONS

This compiler has several more features than the FORTRAN 77 American National Standard. Some enhancements are to the language, and others allow easier communication with C procedures or permit compilation of old (1966 Standard) programs.

Double Complex Data Type

The new type **double complex** is added. each datum is represented by a pair of double precision real variables. A double complex version of every **complex** built-in function is provided.

Internal Files

The FORTRAN 77 American National Standard introduces "internal files" (memory arrays), but restricts their use to formatted sequential I/O statements. This I/O system also permits internal files to be used in formatted direct reads and writes.

Implicit Undefined Statement

FORTRAN 66 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**, and **real** otherwise. FORTRAN 77 has an **implicit** statement for overriding this rule. An additional type, **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 flag is equivalent to beginning each procedure with this statement.

Recursion

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

Automatic Storage

Two new keywords 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. Generally, recursive subroutines should use automatic local variables.

Variable Length Input Lines

The Standard expects input to the compiler to be in 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 seventy-second are ignored.)

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 Standard, there are only 26 letters — FORTRAN is a one-case language. Consistent with ordinary UNIX system usage, this compiler expects lower case input. By default, the compiler converts all upper case characters to lower case except those inside character constants. However, if the **-U** compiler flag is specified, upper case letters are not transformed. In this mode, it is possible to specify external names with upper case letters in them, and to have distinct variables differing only in case. Regardless of the setting of the flag, keywords will only be recognized in lower case.

Include Statement

The statement

```
include 'stuff'
```

is replaced by the contents of the file **stuff**. **include** statements may be nested to a reasonable depth, currently ten.

Binary Initialization Constants

A **logical**, **real** or **integer** variable may be initialized in a **data** statement by a binary constant, denoted by a letter followed by a quoted string. If the letter is **b**, the string is binary, and only zeroes and ones are permitted. If the letter is **o**, the string is octal, with digits **0-7**. If the letter is **z** or **x**, the string is hexadecimal, with digits **0-9**, **a-f**. Thus, the statements

```
integer a(3)
data a / b'1010', o'12', z'a' /
```

initialize all three elements of **a** to ten.

Character Strings

For compatibility with C usage, the following backslash escapes are recognized:

\n	newline
\t	tab
\b	backspace
\f	form feed
\0	null
\'	apostrophe (does not terminate a string)
\"	quotation mark (does not terminate a string)
****	\
*****x*	*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 routines.

Hollerith

FORTRAN 77 does not have the old Hollerith "*n*h" notation, though the new Standard recommends implementing the old Hollerith feature in order to improve compatibility with old programs. In this compiler, Hollerith data may be used in place of character string constants, and may also be used to initialize non-character variables in **data** statements.

Equivalence Statements

This compiler permits single subscripts in **equivalence** statements, under the interpretation that all missing subscripts are equal to 1. A warning message is printed for each such incomplete subscript.

One-Trip DO Loops

The FORTRAN 77 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 flag causes nonstandard loops to be generated.

Commas in Formatted Input

The I/O system attempts to be more lenient than the Standard when it seems worthwhile. When doing a formatted read of non-character variables, commas may be used as value separators in the input record, overriding the field lengths given in the format statement. Thus, the format

```
(i10, f20.10, i4)
```

will read the record

```
-345,.05e-3,12
```

correctly.

Short Integers

integer*2 is a declaration of a 2-byte (1 halfword) integer. **integer*4** (which is the default) is a declaration of a 4-byte (word) integer. (Ordinary integers follow the FORTRAN rules about occupying the same space as a REAL variable; they are assumed to be of C type "**long int**" ; halfword integers are of C type "**short int**" .) An expression involving only objects of type **integer*2** is of that type. Generic functions return short or long integers depending on the actual types of their arguments. If a procedure is compiled using the **-i2** flag, all small integer constants will be of type **integer*2.** If the precision of an integer-valued intrinsic function is not determined by the generic function rules, one will be chosen that returns the prevailing length (**integer*2** when the **-i2** command flag is in effect). When the **-i2** option is in effect, all quantities of type **logical** will be short. Note that these short integer and logical quantities do not obey the standard rules for storage association.

Additional Intrinsic Functions

This compiler supports all of the intrinsic functions specified in the FORTRAN 77 Standard.

All FORTRAN functions are are documented behind the F77 Functions tab of this volume..

F77 COMPILER EXCEPTIONS TO THE STANDARD

Double Precision Alignment

The FORTRAN Standards (both 1966 and 1977) permit **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)
```

The Ridge 32 requires 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 the Ridge, it would be necessary to use separate operations to move the upper and lower halves into the halves of an aligned temporary, then to load that double precision temporary; the reverse would be needed to store a result. We have chosen to require that all double precision real and complex quantities fall on even word boundaries, and to issue a diagnostic if the source code demands a violation of the rule.

Dummy Procedure Arguments

If any argument of a procedure is of type **character**, all dummy procedure arguments of that procedure must be declared in an **external** statement. A warning is printed if a dummy procedure is not declared **external**. Code is correct if there are no **character** arguments.

T and TL Formats

The implementation of the **t** (absolute tab) and **tl** (leftward tab) format codes is defective. These codes allow rereading or rewriting part of the record which has already been processed. The implementation uses seeks, so if the unit is not one which allows seeks, such as a terminal, the program is in error. A benefit of the implementation chosen is that there is no upper limit on the length of a record, nor is it necessary to pre-declare any record lengths except where specifically required by FORTRAN or the operating system.

INTER-PROCEDURE INTERFACE

To be able to write C procedures that call or are called by FORTRAN procedures, it is necessary to know the conventions for procedure names, data representation, return values, and argument lists that the compiled code obeys.

Procedure Names

On this system, the name of a common block or a FORTRAN procedure has an underscore prefix and suffix added to it by the compiler to distinguish it from a C procedure or external variable with the same user-assigned name (example: _name_). FORTRAN library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

Data Representations

The following is a table of corresponding FORTRAN and C 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, except with the **-i2** option which makes integers and logicals into short (2-byte) types.

Return Values

A function of type **integer, logical, real,** or **double precision** declared as a C function returns the corresponding type. A **complex** or **double complex** function is equivalent to a C routine with an additional initial argument that points to the place where the return value is to be stored. Thus,

```
      complex function f( . . . )
```

is equivalent to

```
      f_(temp, . . .)
      struct { float r, i; } *temp;
```

A character-valued function is equivalent to a C routine with two extra initial arguments: a data address and a length. Thus,

```
      character*15 function g( . . . )
```

is equivalent to

```
      g_(result, length, . . .)
      char *result;
      int length;
      . . .
```

and could be invoked in C by

```
      char chars[15];
      . . .
      g_(chars, 15, . . . );
```

Alternate Returns and Computed Gotos

The following program illustrates alternate return labels.

```
      call nret(A, *10, *20, *30)
      code
10    code
20    code
30    code
      stop
      end

      subroutine nret(A, *, *, *)
      code
      return 1
      code
      return 2
      code
      return 3
      return
      end
```

In the subroutine, a numbered **return** is executed. Its number is mapped to the dummy with its ordinal value in the parameter list, and that parameter is used as a **goto** label in the calling program. If **return 2** were executed in **nret**, processing would continue at label **20** in the calling program.

A computed **goto** causes branching to the label in the ordinal position that is equal to an integer value:

```
goto (12, 16, 19), K
```

goes to 12, 16, or 19 if K equals 1, 2, or 3, respectively.

Actually, 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.) In other words, the statement

```
call nret (A, *10, *20, *30)
```

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

```
goto (10, 20, 30), nret(A)
```

(Both statements cause a **computed goto** based on the integer value returned by the function.)

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 **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 s
integer b(3)
. . .
call sam(f, b(2), s)
```

is equivalent to that in

```
int f();
char s[7];
long int b[3];
```

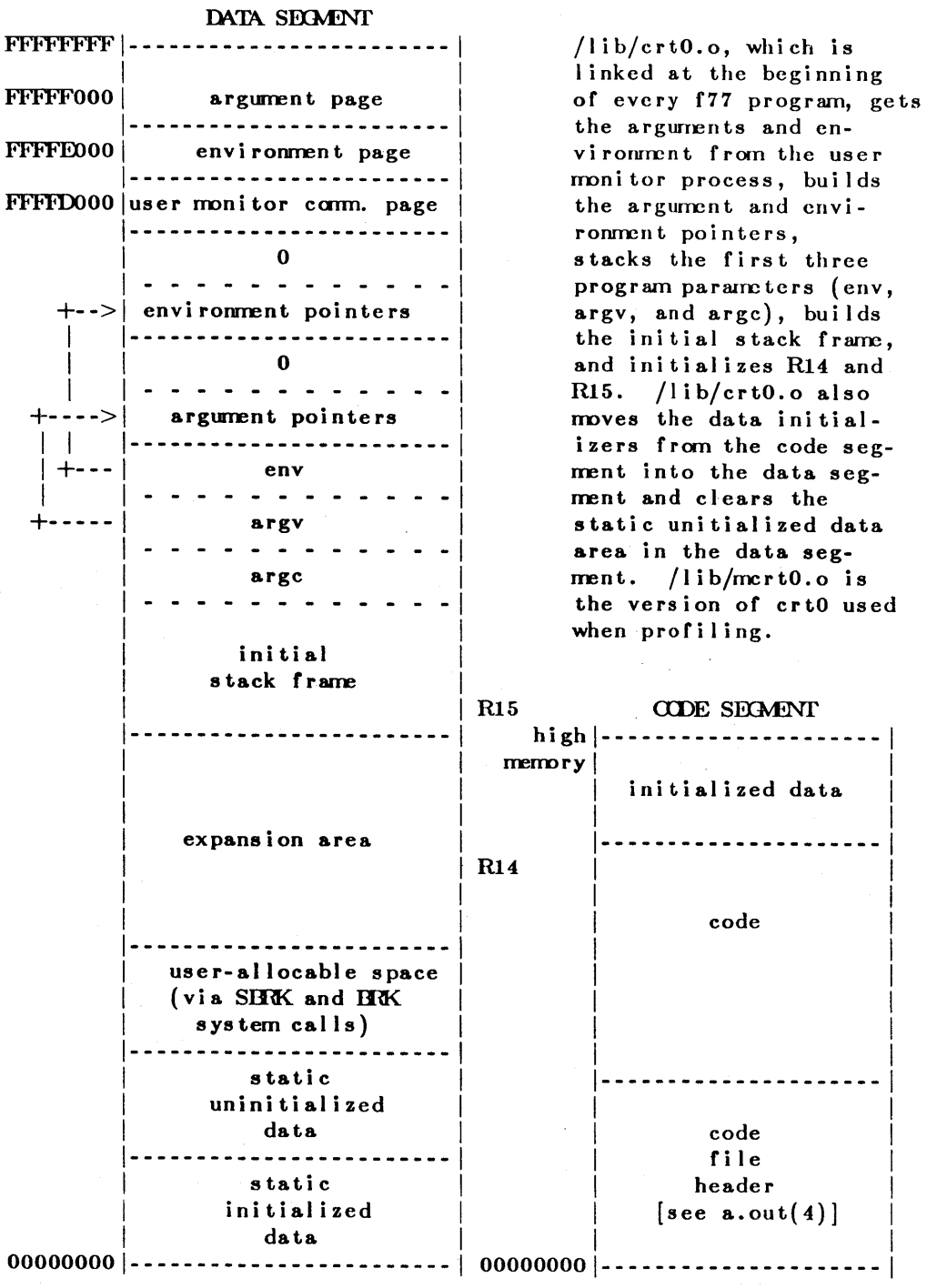
```
. . .
sam_(f, &b[1], s, 0L, 7L);
```

Note that the first element of a C array always has subscript zero, but FORTRAN arrays begin at 1 by default. FORTRAN arrays are stored in column-major order, C arrays are stored in row-major order.

F77 RUNTIME ENVIRONMENT

Code and Data Segments

The separate data and code segments are arranged as follows:



Use of General Registers

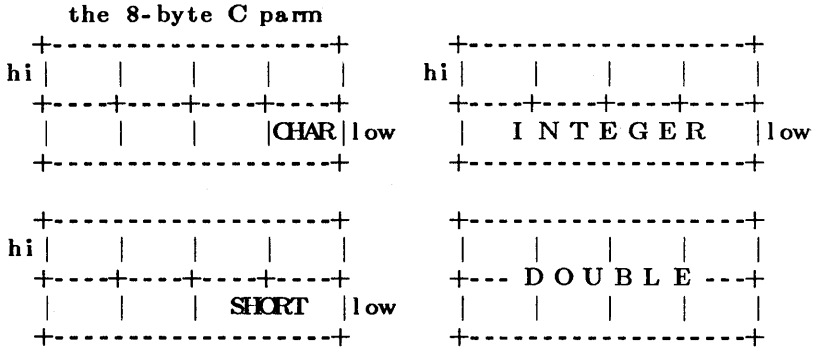
R0 through R5 - scratch registers
R6 through R13 - register variables
R14 - top-of-stack pointer
R15 - current frame pointer

Stack Frames

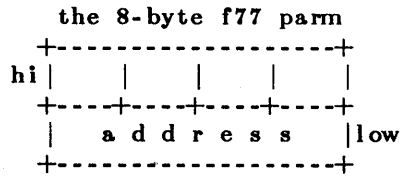
With each procedure call, the current runtime environment is recorded and pushed onto the data stack in a structure called the stack frame. The stack frame is arranged as follows:

```
|-------------------------|
|                  ...    | ...
|  parameters      ...    | ...       Parameters are aligned on
|                  parm3  | R15 + 40  8-byte boundaries.
|                  parm2  | R15 + 32
|                  parm1  | R15 + 24
|-------------------------|
|  return structure length | R15 + 20
| return structure address | R15 + 16
|#of parms passed to procdr| R15 + 12  (if -g option is used)
|     old register 15      | R15 + 8
|         (unused)         | R15 + 4
|     return address       | R15
|-------------------------|
|                    R6    | R15 - 4
| register           R7    | R15 - 8
|   save             R8    | R15 -12
|   area             R9    | R15 -16
|                    R10   | R15 -20
|                    R11   | R15 -24
|                    R12   | R15 -28
|                    R13   | R15 -32
|-------------------------|
|                  var1   | R15 -36
|                  var2   | R15 -40
|                  var3   | R15 -44
|                  var4   | ...
|                  var5   | ...
|                  ...    | ...
|                  ...    | R14 (TOP-OF-STACK)
|-------------------------|
```

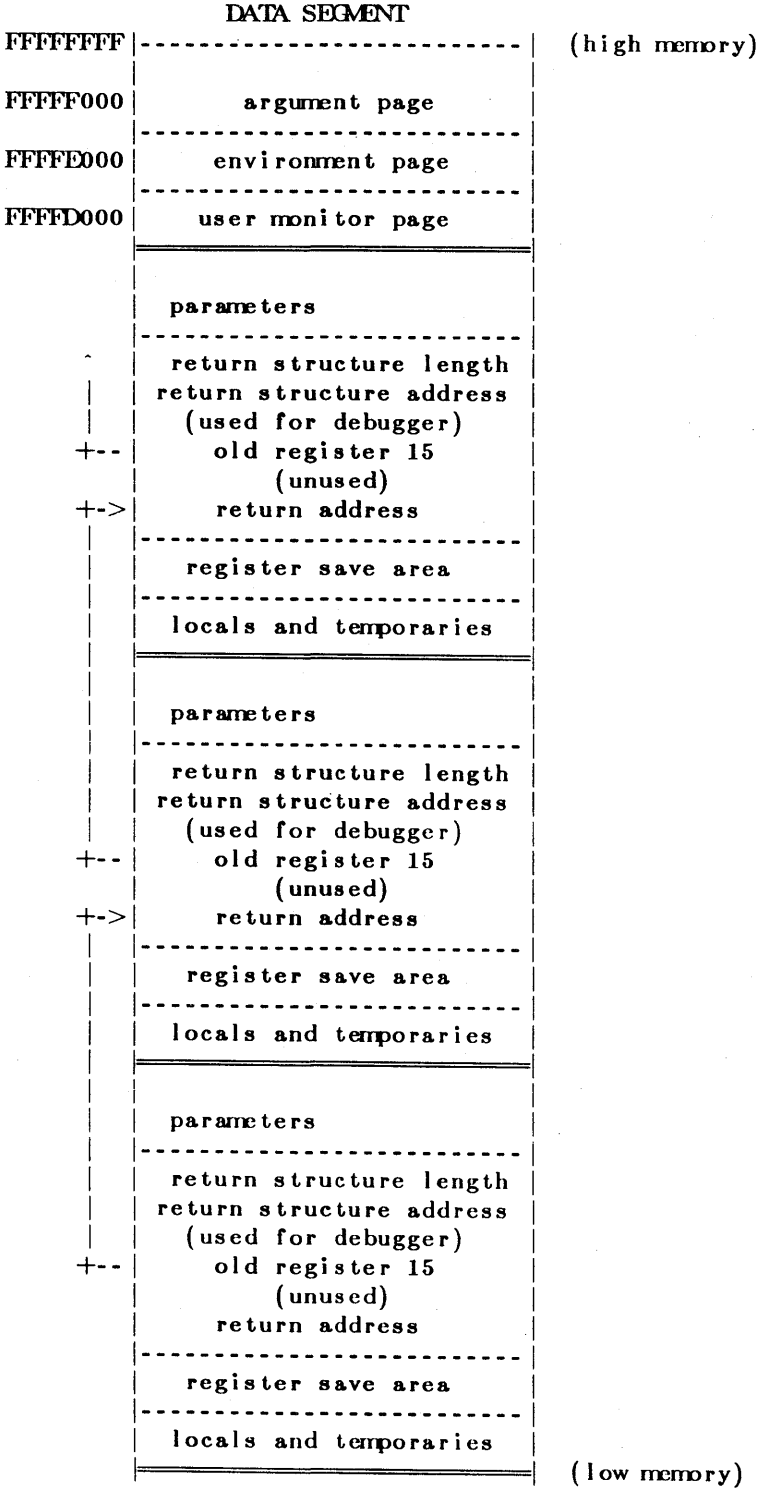
C passes parameters by value, so a C parameter looks like this:



FORTRAN always passes parameters by address, so a FORTRAN parameter always looks like:



A stack of environments, therefore, has the form:



Example of calling C from FORTRAN and FORTRAN from C

Here is a FORTRAN program (named **f.f**) that calls a C subroutine. The C subroutine calls a subroutine in the FORTRAN program. The variable "i" starts with the value 10, and gets 5 subtracted from it and 3 added to it.

```
                program FandC
                i = 10
                print *,'FandC: i= ',i
                call csub5(i)
                end

                subroutine fadd3(i)
c Fortran routine to ADD 3
                i = i + 3
                print *,'fadd3: i= ',i
                return
                end
```

Here is the C program (named **c.c**) that calls and is called by the f77 program **f.f** . Notice that f77 routines must have a '_' character appended to them in a C program, and that f77 passes variables by address.

```
csub5_(x)
/* C program that SUBtracts 5 */
int *x;
{
                *x = *x - 5;
                printf(" csub5: i=  ");
                printf("%d\n", *x);
                fadd3_(x);
}
```

Now the programs are compiled with the f77(1) command. By default, the executable output file is named **a.out** .

```
$ f77 f.f c.c

$ a.out

 FandC: i=  10
 csub5: i=  5
 fadd3: i=  8
```

FORTRAN FILE I/O

The f77 I/O Library implements ANSI 77 FORTRAN standard input and output with a few minor exceptions. Where the standard is vague, we have tried to provide flexibility within the constraints of the UNIX operating system.

The f77 I/O library, libI77.a, includes routines to perform all of the standard types of FORTRAN input and output. Several enhancements and extensions to FORTRAN I/O have been added. The f77 library routines use the C stdio library routines to provide efficient buffering for file I/O.

FORTRAN I/O

The requirements of the Standard impose significant overhead on programs that do large amounts of I/O. Formatted I/O can be very "expensive" while direct access binary I/O is usually very efficient. Because of the complexity of FORTRAN I/O, some general concepts deserve clarification.

Types of I/O

There are three forms of I/O: **formatted**, **unformatted**, and **list-directed**. List-directed is related to formatted but does not obey all the rules for formatted I/O. There are two modes of access to **external** and **internal** files: **direct** and **sequential**. The definition of a logical record depends upon the combination of I/O form and mode specified by the FORTRAN I/O statement.

Direct access

A logical record in a **direct** access **external** file is a string of bytes of a length specified when the file is opened. Read and write statements must not specify logical records longer than the original record size definition. Shorter logical records are allowed. **Unformatted** direct writes leave the unfilled part of the record undefined. **Formatted** direct writes cause the unfilled record to be padded with blanks.

Sequential access

Logical records in **sequentially** accessed **external** files may be of arbitrary and variable length. Logical record length for **unformatted** sequential files is determined by the size of items in the iolist. The requirements of this form of I/O cause the external physical record size to be somewhat larger than the logical record size. For **formatted** write statements, logical record length is determined by the format statement interacting with the iolist at execution time. The "newline" character is the logical record delimiter. Formatted sequential access causes one or more logical records ending with "newline" characters to be read or written.

List-directed I/O

Logical record length for **list-directed** I/O is relatively meaningless. On output, the record length is dependent on the magnitude of the data items. On input, the record length is determined by the data types and the file contents.

Internal I/O

The logical record length for an **internal** read or write is the length of the character variable or array element. Thus a simple character variable is a single logical record. A character variable array is similar to a fixed length direct access file, and obeys the same rules. **Unformatted** I/O is not allowed on "internal" files.

I/O execution

Note that each execution of a FORTRAN **unformatted** I/O statement causes a single logical record to be read or written. Each execution of a FORTRAN **formatted** I/O statement causes one or more logical records to be read or written.

A slash, "/", will terminate assignment of values to the input list during **list-directed** input and the remainder of the current input line is skipped. The standard is rather vague on this point but seems to require that a new external logical record be found at the start of any formatted input. Therefore data following the slash is ignored and may be used to comment the data file.

"Direct access list-directed" I/O is not allowed. **"Unformatted internal"** I/O is not allowed. Both the above will be caught by the compiler. All other flavors of I/O are allowed, although some are not part of the Standard.

Any error detected during I/O processing will cause the program to abort unless alternative action has been provided specifically in the program. Any I/O statement may include an **err=** clause (and **iostat=** clause) to specify an alternative branch to be taken on errors (and return the specific error code). Read statements may include **end=** to branch on end-of-file. File position and the value of I/O list items is undefined following an error.

Implementation details

Some details of the current implementation may be useful in understanding constraints on FORTRAN I/O.

Number of logical units

A program may reference logical units in the range 0 – 999, but only 20 may be open at one time.

Standard logical units

By default, logical units 0, 5, and 6 are opened to "stderr", "stdin", and "stdout" respectively. However they can be re-defined with an **open** statement. To preserve error reporting, it is an error to close logical unit 0 although it may be reopened to another file.

If you want to open the default file name for any preconnected logical unit, remember to **close** the unit first. Redefining the standard units may impair normal console I/O. An alternative is to use shell re-direction to externally re-define the above units. To re-define default blank control or format of the standard input or output files, use the **open** statement specifying the unit number and no file name.

The standard units, 0, 5, and 6, are named internally "stderr", "stdin", and "stdout" respectively. These are not actual file names and cannot be used for opening these units. **Inquire** will not return these names and will indicate that the above units are not named unless they have been opened to real files. The names are meant to make error reporting more meaningful.

Vertical format control

Simple vertical format control is implemented. The logical unit must be opened for sequential access with **"form = 'print'"** . Control codes "0" and "1" are replaced in the output file with "\n" and "\f" respectively. The control character "+" is not implemented and, like any other character in the first position of a record written to a "print" file, is dropped. No vertical format control is recognized for **"direct formatted"** output or **"list-directed"** output.

The open statement

An **open** statement need not specify a file name. If it refers to a logical unit that is already open, the **blank=** and **form=** specifiers may be redefined without affecting the current file position. Otherwise, if **"status = 'scratch'"** is specified, a temporary file with a name of the form "tmp.FXXXXXX" will be opened, and, by default, will be deleted when closed or during termination of program execution. Any other **status=** specifier without an associated file name results in opening a file named "fort.N" where N is the specified logical unit number.

It is an error to try to open an existing file with **"status = 'new'"** . It is an error to try to open a nonexistent file with **"status = 'old'"** . By default, **"status = 'unknown'"** will be assumed, and a file will be created if necessary.

By default, files are positioned at their beginning upon opening, but see *ioinit*(3f) for alternatives. Existing files are never truncated on opening. Sequentially accessed external files are truncated to the current file position on **close** , **backspace** , or **rewind** only if the last access to the file was a write. An **endfile** always causes such files to be truncated to the current file position.

Format interpretation

Formats are parsed at the beginning of each execution of a formatted I/O statement. Upper as well as lower case characters are recognized in format statements and all the alphabetic arguments to the I/O library routines.

If the external representation of a datum is too large for the field width specified, the specified field is filled with asterisks (*). On **E**w.d**E**e output, the exponent field will be filled with asterisks if the exponent representation is too large. This will only happen if "e" is zero.

On output, a real value that is truly zero will display as "0." to distinguish it from a very small non-zero value. This occurs in **F** and **G** format conversions. This was not done for **E** and **D** since the embedded blanks in the external datum causes problems for other input systems.

Non-destructive tabbing is implemented for both internal and external formatted I/O. Tabbing left or right on output does not affect previously written portions of a record. Tabbing right on output causes unwritten portions of a record to be filled with blanks. Tabbing right off the end of an input logical record is an error. Tabbing left beyond the beginning of an input logical record leaves the input pointer at the beginning of the record. The format specifier **T** must be followed by a positive non-zero number. If it is not, it will have a different meaning.

Tabbing left requires seek ability on the logical unit. Therefore it is not allowed in I/O to a terminal or pipe. Likewise, nondestructive tabbing in either direction is possible only on a unit that can seek. Otherwise tabbing right or spacing with **X** will write blanks on the output.

List-directed output

In formatting list-directed output, the I/O system tries to prevent output lines longer than 80 characters. Each external datum will be separated by two spaces. List-directed output of **complex** values includes an appropriate comma. List-directed output distinguishes between **real** and **"double precision"** values and formats them differently. Output of a character string that includes "\n" is interpreted reasonably by the output system.

I/O errors

If I/O errors are not trapped by the user's program an appropriate error message will be written to "stderr" before aborting. An error number will be printed in [ ] along with a brief error message showing the logical unit and I/O state. Error numbers < 100 refer to ROS errors, and are described in the introduction to chapter 2 of the ROS Programmer's Manual. Error numbers ≥ 100 come from the I/O library, and are described further in the **F77 I/O Error Messages** section of this report. For internal I/O, part of the string will be printed with "|" at the current position in the string. For external I/O, part of the current record will be displayed if the error was caused during reading from a file that can backspace.

Non-"ANSI Standard" extensions

Several extensions have been added to the I/O system to provide for functions omitted or poorly defined in the standard. Programmers should be aware that these are non-portable.

Format specifiers

O is a data type specifier for octal numbers. **12O5** is the format specification for 12 fields of 5– character octal numbers.

B is an acceptable edit control specifier. It causes return to the default mode of blank interpretation. This is consistent with **S** which returns to default sign control.

P by itself is equivalent to **0P** . It resets the scale factor to the default value, 0.

The form of the **E**w.d**E**e format specifier has been extended to **D** also. The form **E**w.d.e is allowed but is not standard. The "e" field specifies the minimum number of digits or spaces in the exponent field on output. If the value of the exponent is too large, the exponent notation **e** or **d** will be dropped from the output to allow one more character position. If this is still not adequate, the "e" field will be filled with asterisks (*). The default value for "e" is 2.

An additional form of tab control specification has been added. The Standard forms **TR**n, **TL**n, and **T**n are supported where *n* is a positive non-zero number. If **T** or n**T** is specified, tabbing will be to the next (or n-th) 8-column tab stop. Thus columns of alphanumerics can be lined up without counting.

A format control specifier has been added to suppress the newline at the end of the last record of a formatted sequential write. The specifier is a dollar sign ($). It is constrained by the same rules as the colon (:). It is used typically for console prompts. For example:

```
write (*, "('enter value for x: ',$)")
read (*,*) x
```

Radices other than 10 can be specified for formatted integer I/O conversion. The specifier is patterned after **P**, the scale factor for floating point conversion. It remains in effect until another radix is specified or format interpretation is complete. The specifier is defined as [n]**R** where $2 \leq n \leq 36$. If n is omitted, the default decimal radix is restored.

In conjunction with the above, a sign control specifier has been added to cause integer values to be interpreted as unsigned during output conversion. The specifier is **SU** and remains in effect until another sign control specifier is encountered, or format interpretation is complete. Radix and "unsigned" specifiers could be used to format a hexadecimal dump, as follows:

```
2000  format ( SU, 16R, 8I10.8 )
```

Note: Unsigned integer values greater than (2**31 - 1), i.e. any signed negative value, cannot be read by FORTRAN input routines. All internal values will be output correctly.

Print files

The Standard is ambiguous regarding the definition of a "print" file. Since UNIX has no default "print" file, an additional **form=** specifier is now recognized in the **open** statement. Specifying **"form = 'print'"** implies **formatted** and enables vertical format control for that logical unit. Vertical format control is interpreted only on sequential formatted writes to a "print" file.

The **inquire** statement will return **print** in the **form=** string variable for logical units opened as "print" files. It will return -1 for the unit number of an unconnected file.

If a logical unit is already open, an **open** statement including the **form=** option or the **blank=** option will do nothing but re-define those options. This instance of the **open** statement need not include the file name, and must not include a file name if **unit=** refers to a standard input or output. Therefore, to re-define the standard output as a "print" file, use:

```
open (unit=6, form='print')
```

Scratch files

A **close** statement with **"status = 'keep'"** may be specified for temporary files. This is the default for all other files. Remember to get the scratch file's real name, using **inquire** , if you want to re-open it later.

List-directed I/O

List-directed read has been modified to allow input of a string not enclosed in quotes. The string must not start with a digit, and cannot contain a separator (, or /) or blank (space or tab). A newline will terminate the string unless escaped with \. Any string not meeting the above restrictions must be enclosed in quotes (" or ').

Internal list-directed I/O has been implemented. During internal list reads, bytes are consumed until the iolist is satisfied, or the 'end-of-file' is reached. During internal list writes, records are filled until the iolist is satisfied. The length of an internal array element should be at least 20 bytes to avoid logical record overflow when writing double precision values. Internal list read was implemented to make command line decoding easier. Internal list write should be avoided.

Running older programs

Traditional FORTRAN environments usually assume carriage control on all logical units, usually interpret blank spaces on input as "0"s, and often provide attachment of global file names to logical units at run time. There are several routines in the I/O library to provide these functions.

Traditional unit control parameters

If FORTRAN 66 carriage control features must be maintained, call the *ioinit(3F)* routine to specify control parameters separately.

Preattachment of logical units

The *iinit* routine also can be used to attach logical units to specific files at run time. It will look for names of a user-specified form in the environment and open the corresponding logical unit for **"sequential formatted"** I/O. Names must be of the form **PREFIX***nn* where **PREFIX** is specified in the call to *ioinit* and *nn* is the logical unit to be opened. Unit numbers < 10 must include the leading "0".

Ioinit should prove adequate for most programs as written. However, it is written in FORTRAN 77 specifically so that it may serve as an example for similar user-supplied routines. A copy may be retrieved by "ar x /usr/lib/libI77.a ioinit.f".

Magnetic tape I/O

Because the I/O library uses stdio buffering, reading or writing magnetic tapes should be done with great caution, or avoided if possible. A set of routines has been provided to read and write arbitrary sized buffers to or from tape directly. The buffer must be a **character** object. **Internal** I/O can be used to fill or interpret the buffer. These routines do not use normal FORTRAN I/O processing and do not obey FORTRAN I/O rules. See *tapeio*(3f).

F77 I/O System Exceptions to the Standard

A few exceptions to the Standard remain.

1) Vertical format control

The "+" carriage control specifier is not implemented. It would be difficult to implement it correctly and still provide UNIX-like file I/O.

Furthermore, the carriage control implementation is asymmetrical. A file written with carriage control interpretation cannot be read again with the same characters in column 1.

An alternative to interpreting carriage control internally is to run the output file through a "FORTRAN output filter" before printing [like asa(1) and fpr(1)]. These filters recognize a broader range of carriage control.

2) Default files

Files created by default use of **rewind** or **endfile** statements are opened for **"sequential formatted"** access. There is no way to redefine such a file to allow **direct** or **unformatted** access.

3) Lower case strings

It is not clear if the Standard requires internally generated strings to be upper case or not. As currently written, the **inquire** statement will return lower case strings for any alphanumeric data.

4) Exponent representation on Ew.dEe output

If the field width for the exponent is too small, the standard allows dropping the exponent character but only if the exponent is > 99. This system does not enforce that restriction. Further, the standard implies that the entire field, 'w', should be filled with asterisks if the exponent cannot be displayed. This system fills only the exponent field in the above case since that is more informative.

F77 I/O System Error Messages

The following error messages are generated by the I/O library. The error numbers are returned in the **iostat=** variable if the **err=** return is taken. Error numbers < 100 are generated by the ROS kernel. See the introduction to chapter 2 of the ROS Programmers Manual for their description.

/* 100 */	"error in format" See error message output for the location of the error in the format. Can be caused by more than 10 levels of nested (), or an extremely long format statement.
/* 101 */	"illegal unit number" It is illegal to close logical unit 0. Negative unit numbers are not allowed. The upper limit 999.
/* 102 */	"formatted io not allowed" The logical unit was opened for unformatted I/O.
/* 103 */	"unformatted io not allowed" The logical unit was opened for formatted I/O.
/* 104 */	"direct io not allowed" The logical unit was opened for sequential access, or the logical record length was specified as 0.
/* 105 */	"sequential io not allowed" The logical unit was opened for direct access I/O.
/* 106 */	"can't backspace file" The file associated with the logical unit can't seek. May be a device or a pipe.
/* 107 */	"off beginning of record" The format specified a left tab beyond the beginning of an internal input record.
/* 108 */	"can't stat file"

The system can't return status information about the file. Perhaps the directory is unreadable.

/* 109 */ "no * after repeat count"
Repeat counts in list-directed I/O must be followed by an * with no blank spaces.

/* 110 */ "off end of record"
A formatted write tried to go beyond the logical end-of-record. An unformatted read or write will also cause this.

/* 111 */ "truncation failed"
The truncation of an external sequential file on 'close', 'backspace', 'rewind' or 'endfile' failed.

/* 112 */ "incomprehensible list input"
List input has to be just right.

/* 113 */ "out of free space"
The library dynamically creates buffers for internal use. You ran out of memory for this. Your program is too big!

/* 114 */ "unit not connected"
The logical unit was not open.

/* 115 */ "read unexpected character"
Certain format conversions can't tolerate non-numeric data. Logical data must be T or F.

/* 116 */ "blank logical input field"

/* 117 */ "'new' file exists"
You tried to open an existing file with "status='new'".

/* 118 */ "can't find 'old' file"
You tried to open a non-existent file with "status='old'".

/* 119 */ "unknown system error"
Shouldn't happen, but .....

/* 120 */ "requires seek ability"
Direct access requires seek ability. Sequential unformatted I/O requires seek ability on the file due to the special data structure required. Tabbing left also requires seek ability.

/* 121 */ "illegal argument"

Certain arguments to 'open', etc. will be checked for legitimacy. Often only non-default forms are looked for.

/* 122 */"negative repeat count"
The repeat count for list-directed input must be a positive integer.

/* 123 */"illegal operation for unit"
An operation was requested for a device associated with the logical unit which was not possible. This error is returned by the tape I/O routines if attempting to read past end-of-tape, etc.

APPENDIX. Differences Between Fortran 66 and Fortran 77

The following is a very brief description of the differences between the 1966 [2] and the 1977 [1] Standard languages. We assume that the reader is familiar with Fortran 66. We do not pretend to be complete, precise, or unbiased, but plan to describe what we feel are the most important aspects of the new language. At present the only current information on the 1977 Standard is in publications of the X3J3 Subcommittee of the American National Standards Institute. The following information is from the "/92" document. This draft Standard is written in English rather than a meta-language, but it is forbidding and legalistic. No tutorials or textbooks are available yet.

1. Features Deleted from Fortran 66

1.1. Hollerith

All notions of "Hollerith" (*n*h) as data have been officially removed, although our compiler, like almost all in the foreseeable future, will continue to support this archaism.

1.2. Extended Range

In Fortran 66, under a set of very restrictive and rarely-understood conditions, it is permissible to jump out of the range of a **do** loop, then jump back into it. Extended range has been removed in the Fortran 77 language. The restrictions are so special, and the implementation of extended range is so unreliable in many compilers, that this change really counts as no loss.

2. Program Form

2.1. Blank Lines

Completely blank lines are now legal comment lines.

2.2. Program and Block Data Statements

A main program may now begin with a statement that gives that program an external name:

```
program work
```

Block data procedures may also have names.

```
block data stuff
```

There is now a rule that only *one* unnamed block data procedure may appear in a program. (This rule is not enforced by our system.) The Standard does not specify the effect of the program and block data names, but they are clearly intended to aid conventional loaders.

2.3. ENTRY Statement

Multiple entry points are now legal. Subroutine and function subprograms may have additional entry points, declared by an **entry** statement with an optional argument list.

```
entry extra(a, b, c)
```

Execution begins at the first statement following the **entry** line. All variable declarations must precede all executable statements in the procedure. If the procedure begins with a **subroutine** statement, all entry points are subroutine names. If it begins with a **function** statement, each entry is a function entry point, with type determined by the type declared for the entry name. If any entry is a character-valued function, then all entries must be. In a function, an entry name of the same type as that where control entered must be assigned a value. Arguments do not retain their values between calls. (The ancient trick

of calling one entry point with a large number of arguments to cause the procedure to "remember" the locations of those arguments, then invoking an entry with just a few arguments for later calculation, is still illegal. Furthermore, the trick doesn't work in our implementation, since arguments are not kept in static storage.)

2.4. DO Loops

do variables and range parameters may now be of integer, real, or double precision types. (The use of floating point **do** variables is very dangerous because of the possibility of unexpected roundoff, and we strongly recommend against their use). The action of the **do** statement is now defined for all values of the **do** parameters. The statement

```
do 10 i = 1, u, d
```

performs $\max(0, \lfloor (u-l)/d \rfloor)$ iterations. The **do** variable has a predictable value when exiting a loop: the value at the time a **goto** or **return** terminates the loop; otherwise the value that failed the limit test.

2.5. Alternate Returns

In a **subroutine** or subroutine **entry** statement, some of the arguments may be noted by an asterisk, as in

```
subroutine s(a, *, b, *)
```

The meaning of the "alternate returns" is described in section 5.2 of the Appendix.

3. Declarations

3.1. CHARACTER Data Type

One of the biggest improvements to the language is the addition of a character-string data type. Local and common character variables must have a length denoted by a constant expression:

```
character*17 a, b(3,4)
character*(6+3) c
```

If the length is omitted entirely, it is assumed equal to 1. A character string argument may have a constant length, or the length may be declared to be the same as that of the corresponding actual argument at run time by a statement like

```
character*(*) a
```

(There is an intrinsic function **len** that returns the actual length of a character string). Character arrays and common blocks containing character variables must be packed: in an array of character variables, the first character of one element must follow the last character of the preceding element, without holes.

3.2. IMPLICIT Statement

The traditional implied declaration rules still hold: a variable whose name begins with **i, j, k, l, m,** or **n** is of type **integer,** other variables are of type **real,** unless otherwise declared. This general rule may be overridden with an **implicit** statement:

```
implicit real(a-c,g), complex(w-z), character*(17) (s)
```

declares that variables whose name begins with an **a ,b, c,** or **g** are **real,** those beginning with **w, x, y,** or **z** are assumed **complex,** and so on. It is still poor practice to depend on implicit typing, but this statement is an industry standard.

3.3. PARAMETER Statement

It is now possible to give a constant a symbolic name, as in

```
parameter (x=17, y=x/3, pi=3.14159d0, s='hello')
```

The type of each parameter name is governed by the same implicit and explicit rules as for a variable. The right side of each equal sign must be a constant expression (an expression made up of constants, operators, and already defined parameters).

3.4. Array Declarations

Arrays may now have as many as seven dimensions. (Only three were permitted in 1966). The lower bound of each dimension may be declared to be other than 1 by using a colon. Furthermore, an adjustable array bound may be an integer expression involving constants, arguments, and variables in **common.**

```
real a(-5:3, 7, m:n), b(n+1:2*n)
```

The upper bound on the last dimension of an array argument may be denoted by an asterisk to indicate that the upper bound is not specified:

```
integer a(5, *), b(*), c(0:1, -2:*)
```

3.5. SAVE Statement

A poorly known rule of Fortran 66 is that local variables in a procedure do not necessarily retain their values between invocations of that procedure. At any instant in the execution of a program, if a common block is declared neither in the currently executing procedure nor in any of the procedures in the chain of callers, all of the variables in that common block also become undefined. (The only exceptions are variables that have been defined in a **data** statement and never changed). These rules permit overlay and stack implementations for the affected variables. Fortran 77 permits one to specify that certain variables and common blocks are to retain their values between invocations. The declaration

```
save a, /b/, c
```

leaves the values of the variables **a** and **c** and all of the contents of common block **b** unaffected by a return. The simple declaration

```
save
```

has this effect on all variables and common blocks in the procedure. A common block must be **saved** in every procedure in which it is declared if the desired effect is to occur.

3.6. INTRINSIC Statement

All of the functions specified in the Standard are in a single category, "intrinsic functions", rather than being divided into "intrinsic" and "basic external" functions. If an intrinsic function is to be passed to another procedure, it must be declared **intrinsic.** Declaring it **external** (as in Fortran 66) causes a function other than the built-in one to be passed.

4. Expressions

4.1. Character Constants

Character string constants are marked by strings surrounded by apostrophes. If an apostrophe is to be included in a constant, it is repeated:

```
'abc'
'ain''t'
```

There are no null (zero-length) character strings in Fortran 77. Our compiler has two different quotation marks, " ' " and " " ". (See Section 2.9 in the main text.)

4.2. Concatenation

One new operator has been added, character string concatenation, marked by a double slash ("//"). The result of a concatenation is the string containing the characters of the left operand followed by the characters of the right operand. The strings

```
'ab' // 'cd'
'abcd'
```

are equal. The strings being concatenated must be of constant length in all concatenations that are not the right sides of assignments. (The only concatenation expressions in which a character string declared adjustable with a "*(*)" modifier or a substring denotation with nonconstant position values may appear are the right sides of assignments).

4.3. Character String Assignment

The left and right sides of a character assignment may not share storage. (The assumed implementation of character assignment is to copy characters from the right to the left side.) If the left side is longer than the right, it is padded with blanks. If the left side is shorter than the right, trailing characters are discarded.

4.4. Substrings

It is possible to extract a substring of a character variable or character array element, using the colon notation:

```
a(i,j) (m:n)
```

is the string of $(n-m+1)$ characters beginning at the m^{th} character of the character array element a_{ij}. Results are undefined unless $m \leqslant n$. Substrings may be used on the left sides of assignments and as procedure actual arguments.

4.5. Exponentiation

It is now permissible to raise real quantities to complex powers, or complex quantities to real or complex powers. (The principal part of the logarithm is used). Also, multiple exponentiation is now defined:

```
a**b**c = a ** (b**c)
```

4.6. Relaxation of Restrictions

Mixed mode expressions are now permitted. (For instance, it is permissible to combine integer and complex quantities in an expression.)

Constant expressions are permitted where a constant is allowed, except in **data** statements. (A constant expression is made up of explicit constants and **parameters** and the Fortran operators, except for exponentiation to a floating-point power). An adjustable dimension may now be an integer expression involving constants, arguments, and variables in B common..

Subscripts may now be general integer expressions; the old $cv \pm c'$ rules have been removed. **do** loop bounds may be general integer, real, or double precision expressions. Computed **goto** expressions and I/O unit numbers may be general integer expressions.

5. Executable Statements

5.1. IF-THEN-ELSE

At last, the if-then-else branching structure has been added to Fortran. It is called a "Block If". A Block If begins with a statement of the form

```
if ( . . . ) then
```

and ends with an

```
end if
```

statement. Two other new statements may appear in a Block If. There may be several

```
else if(. . .) then
```

statements, followed by at most one

```
else
```

statement. If the logical expression in the Block If statement is true, the statements following it up to the next **elseif**, **else**, or **endif** are executed. Otherwise, the next **elseif** statement in the group is executed. If none of the **elseif** conditions are true, control passes to the statements following the **else** statement, if any. (The **else** must follow all **elseifs** in a Block If. Of course, there may be Block Ifs embedded inside of other Block If structures). A case construct may be rendered

```
if (s .eq. 'ab') then
. . .
else if (s .eq. 'cd') then
. . .
else
. . .
end if
```

5.2. Alternate Returns

Some of the arguments of a subroutine call may be statement labels preceded by an asterisk, as in

```
call joe(j, *10, m, *2)
```

A **return** statement may have an integer expression, such as

```
return k
```

If the entry point has n alternate return (asterisk) arguments and if $1 \leqslant k \leqslant n$, the return is followed by a branch to the corresponding statement label; otherwise the usual return to the statement following the **call** is executed.

6. Input/Output

6.1. Format Variables

A format may be the value of a character expression (constant or otherwise), or be stored in a character array, as in

```
write(6, '(i5)') x
```

6.2. END=, ERR=, and IOSTAT= Clauses

A **read** or **write** statement may contain **end=**, **err=**, and **iostat=** clauses, as in

```
write(6, 101, err=20, iostat=a(4))
read(5, 101, err=20, end=30, iostat=x)
```

Here 5 and 6 are the *units* on which the I/O is done, 101 is the statement number of the associated format, 20 and 30 are statement numbers, and **a** and **x** are integers. If an error occurs during I/O, control returns to the program at statement 20. If the end of the file is reached, control returns to the program at statement 30. In any case, the variable referred to in the **iostat=** clause is given a value when the I/O statement finishes. (Yes, the value is assigned to the name on the right side of the equal sign.) This value is zero if all went well, negative for end of file, and some positive value for errors.

6.3. Formatted I/O

6.3.1. Character Constants

Character constants in formats are copied literally to the output. Character constants cannot be read into.

```
write(6,'(i2," isn""t ",i1)') 7, 4
```

produces

```
 7 isn't 4
```

Here the format is the character constant

```
(i2,' isn"t ',i1)
```

and the character constant

```
 isn't
```

is copied into the output.

6.3.2. Positional Editing Codes

t, tl, tr, and **x** codes control where the next character is in the record. **tr***n* or *n***x** specifies that the next character is *n* to the right of the current position. **tl***n* specifies that the next character is *n* to the left of the current position, allowing parts of the record to be reconsidered. **t***n* says that the next character is to be character number *n* in the record. (See section 3.4 in the main text.)

6.3.3. Colon

A colon in the format terminates the I/O operation if there are no more data items in the I/O list, otherwise it has no effect. In the fragment

```
x='("hello", :, " there", i4)'
write(6, x) 12
write(6, x)
```

the first **write** statement prints **hello there 12**, while the second only prints **hello.**

6.3.4. Optional Plus Signs

According to the Standard, each implementation has the option of putting plus signs in front of non-negative numeric output. The **sp** format code may be used to make the optional plus signs actually appear for all subsequent items while the format is active. The **ss** format code guarantees that the I/O system will not insert the optional plus signs, and the **s** format code restores the default behavior of the I/O system. (Since we never put

out optional plus signs, ss and s codes have the same effect in our implementation.)

6.3.5. Blanks on Input

Blanks in numeric input fields, other than leading blanks will be ignored following a **bn** code in a format statement, and will be treated as zeros following a **bz** code in a format statement. The default for a unit may be changed by using the **open** statement. (Blanks are ignored by default.)

6.3.6. Unrepresentable Values

The Standard requires that if a numeric item cannot be represented in the form required by a format code, the output field must be filled with asterisks. (We think this should have been an option.)

6.3.7. Iw.m

There is a new integer output code, **i***w.m*. It is the same as **i***w*, except that there will be at least *m* digits in the output field, including, if necessary, leading zeros. The case **i***w*.0 is special, in that if the value being printed is 0, the output field is entirely blank. **i***w*.**1** is the same as **i***w*.

6.3.8. Floating Point

On input, exponents may start with the letter **E, D, e,** or **d.** All have the same meaning. On output we always use **e**. The **e** and **d** format codes also have identical meanings. A leading zero before the decimal point in **e** output without a scale factor is optional with the implementation. (We do not print it.) There is a **g***w.d* format code which is the same as **e***w.d* and **f***w.d* on input, but which chooses **f** or **e** formats for output depending. on the size of the number and of *d*.

6.3.9. "A" Format Code

A codes are used for character values. **a***w* use a field width of *w*, while a plain **a** uses the length of the character item.

6.4. Standard Units

There are default formatted input and output units. The statement

```
read 10, a, b
```

reads from the standard unit using format statement 10. The default unit may be explicitly specified by an asterisk, as in

```
read(*, 10) a,b
```

Similarly, the standard output units is specified by a **print** statement or an asterisk unit:

```
print 10
write(*, 10)
```

6.5. List-Directed Formatting

List-directed I/O is a kind of free form input for sequential I/O. It is invoked by using an asterisk as the format identifier, as in

```
read(6, *) a,b,c
```

On input, values are separated by strings of blanks and possibly a comma. Values, except for character strings, cannot contain blanks. End of record counts as a blank, except in character strings, where it is ignored. Complex constants are given as two real constants separated by a comma and enclosed in parentheses. A null input field, such as between two consecutive commas, means the corresponding variable in the I/O list is not changed. Values may be preceded by repetition counts, as in

```
4*(3.,2.)  2*, 4*'hello'
```

which stands for 4 complex constants, 2 null values, and 4 string constants.

For output, suitable formats are chosen for each item. The values of character strings are printed; they are not enclosed in quotes, so they cannot be read back using list-directed input.

6.6. Direct I/O

A file connected for direct access consists of a set of equal-sized records each of which is uniquely identified by a positive integer. The records may be written or read in any order, using direct access I/O statements.

Direct access **read** and **write** statements have an extra argument, **rec=**, which gives the record number to be read or written.

```
read(2, rec=13, err=20) (a(i), i=1, 203)
```

reads the thirteenth record into the array **a**.

The size of the records must be given by an **open** statement (see below). Direct access files may be connected for either formatted or unformatted I/O.

6.7. Internal Files

Internal files are character string objects, such as variables or substrings, or arrays of type character. In the former cases there is only a single record in the file, in the latter case each array element is a record. The Standard includes only sequential formatted I/O on internal files. (I/O is not a very precise term to use here, but internal files are dealt with using **read** and **write**). There is no list-directed I/O on internal files. Internal files are used by giving the name of the character object in place of the unit number, as in

```
character*80 x
read(5,"(a)") x
read(x,"(i3,i4)") n1,n2
```

which reads a card image into x and then reads two integers from the front of it. A sequential **read** or **write** always starts at the beginning of an internal file.

(We also support a compatible extension, direct I/O on internal files. This is like direct I/O on external files, except that the number of records in the file cannot be changed.)

6.8. OPEN, CLOSE, and INQUIRE Statements

These statements are used to connect and disconnect units and files, and to gather information about units and files.

6.8.1. OPEN

The **open** statement is used to connect a file with a unit, or to alter some properties of the connection. The following is a minimal example.

```
open(1, file='fort.junk')
```

open takes a variety of arguments with meanings described below.

unit= a small non-negative integer which is the unit to which the file is to be connected. We allow, at the time of this writing, 0 through 9. If this parameter is the first one in the **open** statement, the **unit=** can be omitted.

iostat= is the same as in **read** or **write.**

err= is the same as in **read** or **write.**

file= a character expression, which when stripped of trailing blanks, is the name of the file to be connected to the unit. The filename should not be given if the **status=scratch.**

status= one of **old, new, scratch,** or **unknown.** If this parameter is not given, **unknown** is assumed. If **scratch** is given, a temporary file will be created. Temporary files are destroyed at the end of execution. If **new** is given, the file will be created if it doesn't exist, or truncated if it does. The meaning of **unknown** is processor dependent; our system treats it as synonymous with **old.**

access= sequential or **direct,** depending on whether the file is to be opened for sequential or direct I/O.

form= formatted or **unformatted.**

recl= a positive integer specifying the record length of the direct access file being opened. We measure all record lengths in bytes. On UNIX systems a record length of 1 has the special meaning explained in section 5.1 of the text.

blank= null or **zero.** This parameter has meaning only for formatted I/O. The default value is **null. zero** means that blanks, other than leading blanks, in numeric input fields are to be treated as zeros.

Opening a new file on a unit which is already connected has the effect of first closing the old file.

6.8.2. CLOSE

close severs the connection between a unit and a file. The unit number must be given. The optional parameters are **iostat=** and **err=** with their usual meanings, and **status=** either **keep** or **delete.** Scratch files cannot be kept, otherwise **keep** is the default. **delete** means the file will be removed. A simple example is

```
close(3, err=17)
```

6.8.3. INQUIRE

The **inquire** statement gives information about a unit ("inquire by unit") or a file ("inquire by file"). Simple examples are:

```
inquire(unit=3, namexx)
inquire(file='junk', number=n, exist=l)
```

file= a character variable specifies the file the **inquire** is about. Trailing blanks in the file name are ignored.

unit= an integer variable specifies the unit the **inquire** is about. Exactly one of **file=** or **unit=** must be used.

iostat=, err= are as before.

exist= a logical variable. The logical variable is set to **.true.** if the file or unit exists and is set to **.false.** otherwise.

opened= a logical variable. The logical variable is set to **.true.** if the file is connected to a unit or if the unit is connected to a file, and it is set to **.false.** otherwise.

number= an integer variable to which is assigned the number of the unit connected to the file, if any.

named= a logical variable to which is assigned **.true.** if the file has a name, or **.false.** otherwise.

name= a character variable to which is assigned the name of the file (inquire by file) or the name of the file connected to the unit (inquire by unit). The name will be the full name of the file.

access= a character variable to which will be assigned the value **'sequential'** if the connection is for sequential I/O, **'direct'** if the connection is for direct I/O. The value becomes undefined if there is no connection.

sequential= a character variable to which is assigned the value **'yes'** if the file could be connected for sequential I/O, **'no'** if the file could not be connected for sequential I/O, and **'unknown'** if we can't tell.

direct= a character variable to which is assigned the value **'yes'** if the file could be connected for direct I/O, **'no'** if the file could not be connected for direct I/O, and **'unknown'** if we can't tell.

form= a character variable to which is assigned the value **'formatted'** if the file is connected for formatted I/O, or **'unformatted'** if the file is connected for unformatted I/O.

formatted= a character variable to which is assigned the value **'yes'** if the file could be connected for formatted I/O, **'no'** if the file could not be connected for formatted I/O, and **'unknown'** if we can't tell.

unformatted= a character variable to which is assigned the value **'yes'** if the file could be connected for unformatted I/O, **'no'** if the file could not be connected for unformatted I/O, and **'unknown'** if we can't tell.

recl= an integer variable to which is assigned the record length of the records in the file if the file is connected for direct access.

nextrec= an integer variable to which is assigned one more than the number of the the last record read from a file connected for direct access.

blank= a character variable to which is assigned the value **'null'** if null blank control is in effect for the file connected for formatted I/O, **'zero'** if blanks are being converted to zeros and the file is connected for formatted I/O.

The gentle reader will remember that the people who wrote the standard probably weren't thinking of his needs. Here is an example. The declarations are omitted.

```
open(1, file="/dev/console")
```

On a UNIX system this statement opens the console for formatted sequential I/O. An **inquire** statement for either unit 1 or file "/dev/console" would reveal that the file exists, is connected to unit 1, has a name, namely "/dev/console", is opened for sequential I/O, could be connected for sequential I/O, could not be connected for direct I/O (can't seek), is connected for formatted I/O, could be connected for formatted I/O, could not be connected for unformatted I/O (can't seek), has neither a record length nor a next record number, and is ignoring blanks in numeric fields.

In the UNIX system environment, the only way to discover what permissions you have for a file is to open it and try to read and write it. The **err=** parameter will return system error numbers. The **inquire** statement does not give a way of determining permissions.

f77 FORTRAN Functions

TABLE OF CONTENTS

For convenience, these FORTRAN function descriptions were removed from the ROS Reference Manual section (3F) and organized here.

f77 FORTRAN Functions

min	minimum-value functions
mod	remaindering intrinsic functions
perror	get system error messages
putc	write a character to a fortran logical unit
qsort	quick sort
rand	uniform random-number generator
rename	rename a file
round	nearest integer functions
sign	transfer-of-sign intrinsic function
signal	specify action on receipt of system signal
sin	sine intrinsic function
sinh	hyperbolic sine intrinsic function
sleep	suspend execution for an interval
sqrt	square root intrinsic function
stat	get file status
system	execute a system command
tan	tangent intrinsic function
tanh	hyperbolic tangent intrinsic function
time	return system time
ttynam	find name of a terminal port
unlink	remove a directory entry
wait	wait for a process to terminate

NAME

abort – terminate Fortran program

SYNTAX

call abort ( )

DESCRIPTION

Abort terminates the program which calls it, closing all open files truncated to the current position of the file pointer.

DIAGNOSTICS

When invoked, *abort* prints "Fortran abort routine called" on the standard error output.

SEE ALSO

abort(3C).

NAME

abs, iabs, dabs, cabs, zabs – Fortran absolute value

SYNTAX

```
integer i1, i2
real r1, r2
double precision dp1, dp2
complex cx1, cx2
double complex dx1, dx2

r2 = abs(r1)

i2 = iabs(i1)
i2 = abs(i1)

dp2 = dabs(dp1)
dp2 = abs(dp1)

cx2 = cabs(cx1)
cx2 = abs(cx1)

dx2 = zabs(dx1)
dx2 = abs(dx1)
```

DESCRIPTION

Abs returns the absolute value of its argument in the same type as its argument.

Iabs returns the integer absolute value of its integer argument.

Dabs returns the double-precision absolute value of its double-precision argument.

Cabs returns the complex absolute value of its complex argument.

Zabs returns the double-complex absolute value of its double-complex argument.

Abs works for any data type, but the various forms are for programming clarity.

SEE ALSO

floor(3M).

NAME

access – determine accessability of a file

SYNTAX

integer function access (name, mode)
character*(*) name, mode

DESCRIPTION

Access checks the given file, *name*, for accessability with respect to the caller according to *mode*. *Mode* may include in any order and in any combination one or more of:

r	test for read permission
w	test for write permission
x	test for execute permission
(blank)	test for existence

An error code is returned if either argument is illegal, or if the file can not be accessed in all of the specified modes. 0 is returned if the specified access would be successful.

FILES

/usr/lib/libU77.a

SEE ALSO

access(2), perror(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <*sys/param.h*>.

NAME

acos, dacos – Fortran arccosine intrinsic function

SYNTAX

real r1, r2
double precision dp1, dp2

r2 = acos(r1)

dp2 = dacos(dp1)
dp2 = acos(dp1)

DESCRIPTION

Acos returns the arccosine of its argument, in real or double-precision that matches its argument.

Dacos returns the double-precision arccosine of its double-precision argument.

Although *Acos* works for either data type, *dacos* is for clarity in programming.

SEE ALSO

trig(3M).

NAME

aimag, dimag – Fortran imaginary part of complex argument

SYNTAX

```
real r
complex cxr
double precision dp
double complex cxd
r = aimag(cxr)
dp = dimag(cxd)
```

DESCRIPTION

Aimag returns the imaginary part of its single-precision complex argument.

Dimag returns the double-precision imaginary part of its double-complex argument.

NAME

aint, dint – Fortran integer part intrinsic function

SYNTAX

```
real r1, r2
double precision dp1, dp2
r2 = aint(r1)
dp2 = dint(dp1)
dp2 = aint(dp1)
```

DESCRIPTION

Aint returns the truncated value of its argument, in real or double-precision that matches the argument.

Dint returns the double-precision truncated value of its double precision argument.

Although *Aint* works for either data type, *dint* is for clarity in programming.

NAME

alarm – execute a subroutine after a specified time

SYNTAX

integer function alarm (time, proc)
integer time
external proc

DESCRIPTION

This routine arranges for subroutine *proc* to be called after *time* seconds. If *time* is "0", the alarm is turned off and no routine will be called. The returned value will be the time remaining on the last alarm.

FILES

/usr/lib/libU77.a

SEE ALSO

alarm(3C), sleep(3F), signal(3F)

BUGS

Alarm and *sleep* interact. If *sleep* is called after *alarm*, the *alarm* process will never be called. SIGALRM will occur at the lesser of the remaining *alarm* time or the *sleep* time.

NAME

asin, dasin – Fortran arcsine intrinsic function

SYNTAX

```
real r1, r2
double precision dp1, dp2
r2 = asin(r1)
dp2 = dasin(dp1)
dp2 = asin(dp1)
```

DESCRIPTION

Asin returns the arcsine of its argument, in the real or double type that matches its argument.

Dasin returns the double-precision arcsine of its double-precision argument.

Asin works with real or double types, but *Dasin* is for programming clarity.

SEE ALSO

trig(3M).

NAME

atan, datan – Fortran arctangent intrinsic function

SYNTAX

real r1, r2
double precision dp1, dp2

r2 = atan(r1)

dp2 = datan(dp1)
dp2 = atan(dp1)

DESCRIPTION

Atan returns the arctangent of its argument, in real or double-precision type that matches the argument.

Datan returns the double-precision arctangent of its double-precsion argument.

Although *Atan* works for either data type, *datan* is for clarity in programming.

SEE ALSO

trig(3M).

NAME

atan2, datan2 – Fortran arctangent intrinsic function

SYNTAX

real r1, r2, r3
double precision dp1, dp2, dp3

r3 = atan2(r1, r2)

dp3 = datan2(dp1, dp2)
dp3 = atan2(dp1, dp2)

DESCRIPTION

Atan2 returns the arctangent of arg1/arg2, in real or double-precision type that matches the arguments.

Datan2 returns the double-precision arctangent of the double-precsion arguments arg1/arg2.

Although *Atan2* works for either data type, *datan* is for clarity in programming.

SEE ALSO

trig(3M).

NAME

bit – and, or, xor, not, rshift, lshift bitwise functions

SYNTAX

(intrinsic) function and (word1, word2)

(intrinsic) function or (word1, word2)

(intrinsic) function xor (word1, word2)

(intrinsic) function not (word)

(intrinsic) function rshift (word, nbits)

(intrinsic) function lshift (word, nbits)

DESCRIPTION

These bitwise functions are built into the compiler and return the data type of their argument(s). It is recommended that their arguments be **integer** values; inappropriate manipulation of **real** objects may cause unexpected results.

The bitwise combinatorial functions return the bitwise "and" (**and**), "or" (**or**), or "exclusive or" (**xor**) of two operands. **Not** returns the bitwise complement of its operand.

Lshift, or *rshift* with a negative *nbits*, is a logical left shift with no end around carry. *Rshift*, or *lshift* with a negative *nbits*, is an arithmatic right shift with sign extension. No test is made for a reasonable value of *nbits*.

FILES

These functions are generated in-line by the f77 compiler.

NAME

chdir – change default directory

SYNTAX

integer function chdir (dirname)
character*(*) dirname

DESCRIPTION

The default directory for creating and locating files will be changed to *dirname*. Zero is returned if successful; an error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

chdir(2), cd(1), perror(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <*sys/param.h*>.

Use of this function may cause **inquire** by unit to fail.

NAME

chmod – change mode of a file

SYNOPSIS

integer function chmod (name, mode)
character*(*) name, mode

DESCRIPTION

This function changes the filesystem *mode* of file *name*. *Mode* can be any specification recognized by *chmod*(1). *Name* must be a single pathname.

The normal returned value is 0. Any other value will be a system error number.

FILES

/usr/lib/libU77.a
/bin/chmod exec'ed to change the mode.

SEE ALSO

chmod(1)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <*sys/param.h*>.

NAME

conjg, dconjg – Fortran complex conjugate intrinsic function

SYNTAX

complex cx1, cx2
double complex dx1, dx2

cx2 = conjg(cx1)

dx2 = dconjg(dx1)

DESCRIPTION

Conjg returns the complex conjugate of its complex argument.

Dconjg returns the double-complex conjugate of its double-complex argument.

NAME

cos, dcos, ccos – Fortran cosine intrinsic function

SYNTAX

real r1, r2
double precision dp1, dp2
complex cx1, cx2

r2 = cos(r1)

dp2 = dcos(dp1)
dp2 = cos(dp1)

cx2 = ccos(cx1)
cx2 = cos(cx1)

DESCRIPTION

Cos returns the cosine of its argument, in the real, complex, or double-precision type of its argument.

Dcos returns the double-precision cosine of its double-precision argument.

Ccos returns the complex cosine of its complex argument.

Although *Cos* works with any type, the other forms are used for clarity in programming.

SEE ALSO

trig(3M).

NAME

cosh, dcosh – Fortran hyperbolic cosine intrinsic function

SYNTAX

real r1, r2
double precision dp1, dp2

r2 = cosh(r1)

dp2 = dcosh(dp1)
dp2 = cosh(dp1)

DESCRIPTION

Cosh returns the hyperbolic cosine of its argument, in the real or double-precsion type of its argument.

Dcosh returns the double-precision hyperbolic cosine of its double-precision argument.

Although *Cosh* works for either data type, *Dcosh* is for clarity in programming.

SEE ALSO

sinh(3M).

NAME

etime, dtime – return elapsed execution time

SYNTAX

function etime (tarray)
real tarray(2)

function dtime (tarray)
real tarray(2)

DESCRIPTION

Etime returns in tarray(1) the elapsed user time for the calling process since start of execution, and in tarray(2) the elapsed system time for the calling process since start of execution.

Dtime returns in tarray(1) the delta time for the calling process since the last call to **dtime**, and in tarray(2) the delta system time for the calling process since the last call to **dtime**.

The return value of **dtime** and **etime** is the sum of the two **tarray** times they report.

The resolution of all timing is 1/60 of one millisecond.

FILES

/usr/lib/libU77.a

SEE ALSO

times(2)

BUGS

The f77 compiler fails to convert the returned value to single precision from double precision, so the return value must be read as a double.

Doug, do you mean the values in the array, or the return value of the function?

NAME

exp, dexp, cexp – Fortran exponential intrinsic function

SYNTAX

real r1, r2
double precision dp1, dp2
complex cx1, cx2

r2 = exp(r1)

dp2 = dexp(dp1)
dp2 = exp(dp1)

cx2 = clog(cx1)
cx2 = exp(cx1)

DESCRIPTION

Exp returns the real exponential function *e**arg* for its argument, in the real, double-precision, or complex type of the argument.

Dexp returns the double-precision exponential function of its double-precision argument.

Cexp returns the complex exponential function of its complex argument.

Although *exp* works with reals, doubles, and complexes, the other forms are for clarity in programming.

SEE ALSO

exp(3M).

NAME

fdate – return date and time in an ASCII string

SYNOPSIS

subroutine fdate (string)
character*(*) string

character*(*) function fdate()

DESCRIPTION

Fdate returns the current date and time as a 24 character string in the format described under *ctime*(3). Neither 'newline' nor NULL will be included.

Fdate can be called either as a function or as a subroutine. If called as a function, the calling routine must define its type and length. For example:

```
character*24  fdate
external      fdate

write(*,*) fdate()
```

FILES

/usr/lib/libU77.a

SEE ALSO

ctime(3), time(3F), itime(3F), idate(3F), ltime(3F)

NAME

fork – create a copy of this process

SYNTAX

integer function fork ()

DESCRIPTION

Fork creates a copy of the calling process. The only distinction between the 2 processes is that the value returned to one of them (referred to as the 'parent' process) will be the process id if the copy. The copy is usually referred to as the 'child' process. The value returned to the 'child' process will be zero.

All logical units open for writing are flushed before the fork to avoid duplication of the contents of the I/O buffers in the external file(s).

If the returned value is negative, it indicates an error and will be the negation of the system error code. See perror(3F).

A corresponding *exec* routine has not been provided because there is no satisfactory way to retain open logical units across the exec. However, the usual function of *fork/exec* can be performed using *system*(3F).

FILES

/usr/lib/libU77.a

SEE ALSO

fork(2), wait(3F), kill(3F), perror(3F)

NAME

fseek, ftell – reposition a file on a logical unit

SYNTAX

integer function fseek (lunit, offset, from)
integer offset, from

integer function ftell (lunit)

DESCRIPTION

lunit must refer to an open logical unit. *offset* is an offset in bytes relative to the position specified by *from*. Valid values for *from* are:

0 meaning 'beginning of the file'
1 meaning 'the current position'
2 meaning 'the end of the file'

The value returned by *fseek* will be 0 if successful, a system error code otherwise. (See perror(3F))

Ftell returns the current position of the file associated with the specified logical unit. The value is an offset, in bytes, from the beginning of the file. If the value returned is negative, it indicates an error and will be the negation of the system error code. (See perror(3F))

FILES

/usr/lib/libU77.a

SEE ALSO

fseek(3S), perror(3F)

NAME

int, ifix, idint, real, float, sngl, dble, cmplx, dcmplx, ichar, char – explicit Fortran type conversion

SYNTAX

```
integer i, j
real r, s
double precision dp, dq
complex cx
double complex dcx
character*1 ch

i = int(r)
i = int(dp)
i = int(cx)
i = int(dcx)
i = ifix(r)
i = idint(dp)

r = real(i)
r = real(dp)
r = real(cx)
r = real(dcx)
r = float(i)
r = sngl(dp)

dp = dble(i)
dp = dble(r)
dp = dble(cx)
dp = dble(dcx)

cx = cmplx(i)
cx = cmplx(i, j)
cx = cmplx(r)
cx = cmplx(r, s)
cx = cmplx(dp)
cx = cmplx(dp, dq)
cx = cmplx(dcx)

dcx = dcmplx(i)
dcx = dcmplx(i, j)
dcx = dcmplx(r)
dcx = dcmplx(r, s)
dcx = dcmplx(dp)
dcx = dcmplx(dp, dq)
dcx = dcmplx(cx)

i = ichar(ch)
ch = char(i)
```

DESCRIPTION

These functions perform conversion from one data type to another.

int converts to *integer* form its *real, double precision, complex,* or *double complex* argument. If the argument is *real* or *double precision,* **int** returns the integer whose magnitude is the largest integer that does not exceed the magnitude of the argument and whose sign is the same as the sign of the argument (i.e. truncation). For complex types, the above rule is applied to the real part. **ifix** and **idint** convert only *real* and *double precision* arguments respectively.

real converts to *real* form an *integer, double precision, complex,* or *double complex* argument. If the argument is *double precision* or *double complex,* as much precision is kept as is possible. If the argument is one of the complex types, the real part is returned. **float** and **sngl** convert only *integer* and *double precision* arguments respectively.

dble converts any *integer, real, complex,* or *double complex* argument to *double precision* form. If the argument is of a complex type, the real part is returned.

cmplx converts its *integer, real, double precision,* or *double complex* argument(s) to *complex* form.

dcmplx converts to *double complex* form its *integer, real, double precision,* or *complex* argument(s).

Either one or two arguments may be supplied to **cmplx** and **dcmplx** . If there is only one argument, it is taken as the real part of the complex type and a imaginary part of zero is supplied. If two arguments are supplied, the first is taken as the real part and the second as the imaginary part.

ichar converts from a character to an integer depending on the character's position in the collating sequence.

char returns the character in the *i*th position in the processor collating sequence where *i* is the supplied argument.

For a processor capable of representing *n* characters,

ichar(char(i)) = i for 0 <= i < *n*, and

char(ichar(ch)) = ch for any representable character *ch*.

NAME

getarg – return Fortran command-line argument

SYNTAX

character*N c
integer i

getarg(i, c)

DESCRIPTION

Getarg returns the *i*-th command-line argument of the current process. Thus, if a program were invoked via

```
foo arg1 arg2 arg3
```

getarg(2, c) would return the string "arg2" in the character variable *c*.

SEE ALSO

getopt(3C).

NAME

getc, fgetc – get a character from a logical unit

SYNTAX

integer function getc (char)
character char

integer function fgetc (lunit, char)
character char

DESCRIPTION

These routines return the next character from a file associated with a fortran logical unit, bypassing normal fortran I/O. *Getc* reads from logical unit 5, normally connected to the control terminal input.

The value of each function is a system status code. Zero indicates no error occured on the read; – 1 indicates end of file was detected. A positive value will be either a UNIX system error code or an f77 I/O error code. See perror(3F).

FILES

/usr/lib/libU77.a

SEE ALSO

getc(3S), intro(2), perror(3F)

NAME

getcwd – get pathname of current working directory

SYNTAX

integer function getcwd (dirname)
character*(*) dirname

DESCRIPTION

The pathname of the default directory for creating and locating files will be returned in *dirname*. The value of the function will be zero if successful; an error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

chdir(3F), perror(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in *<sys/param.h>*.

NAME

getenv – return Fortran environment variable

SYNTAX

character*N c

getenv("TMPDIR", c)

DESCRIPTION

Getenv returns the character-string value of the environment variable represented by its first argument into the character variable of its second argument. If no such environment variable exists, all blanks will be returned.

SEE ALSO

getenv(3C), environ(5).

NAME

getlog – get user's login name

SYNTAX

subroutine getlog (name)
character*(*) name

character*(*) function getlog()

DESCRIPTION

Getlog will return the user's login name or all blanks if the process is running detached from a terminal.

FILES

/usr/lib/libU77.a

SEE ALSO

getlogin(3)

NAME

getpid – get process id

SYNTAX

integer function getpid()

DESCRIPTION

Getpid returns the process ID number of the current process.

FILES

/usr/lib/libU77.a

SEE ALSO

getpid(2)

NAME

getuid, getgid – get user or group ID of the caller

SYNTAX

integer function getuid()

integer function getgid()

DESCRIPTION

These functions return the real user or group ID of the user of the process.

FILES

/usr/lib/libU77.a

SEE ALSO

getuid(2)

NAME

hostnm – get name of current host

SYNTAX

integer function hostnm (name)
character*(*) name

DESCRIPTION

This function puts the name of the current host into character string *name*. The return value should be 0; any other value indicates an error.

FILES

/usr/lib/libU77.a

SEE ALSO

gethostname(2)

NAME

idate, itime – return date or time in numerical form

SYNTAX

subroutine idate (iarray)
integer iarray(3)

subroutine itime (iarray)
integer iarray(3)

DESCRIPTION

Idate returns the current date in *iarray*. The order is: day, mon, year. Month will be in the range 1-12. Year will be $\geq$ 1969.

Itime returns the current time in *iarray*. The order is: hour, minute, second.

FILES

/usr/lib/libU77.a

SEE ALSO

ctime(3F), fdate(3F)

NAME

index – return location of Fortran substring

SYNTAX

character*N1 ch1
character*N2 ch2
integer i

i = index(ch1, ch2)

DESCRIPTION

Index returns the location of substring *ch2* in string *ch1*. The value returned is the position at which substring *ch2* starts, or 0 if it is not present in string *ch1*.

NAME

ioinit – change f77 I/O initialization

SYNTAX

logical function ioinit (cctl, bzro, apnd, prefix, vrbose)
logical cctl, bzro, apnd, vrbose
character*(*) prefix

DESCRIPTION

This routine will initialize several global parameters in the f77 I/O system, and attach externally defined files to logical units at run time. The effect of the flag arguments applies to logical units opened after *ioinit* is called. The exception is the preassigned units, 5 and 6, to which *cctl* and *bzro* will apply at any time. *Ioinit* is written in Fortran-77.

By default, carriage control is not recognized on any logical unit. If *cctl* is **.true.** then carriage control will be recognized on formatted output to all logical units except unit 0, the diagnostic channel. Otherwise the default will be restored.

By default, trailing and embedded blanks in input data fields are ignored. If *bzro* is **.true.** then such blanks will be treated as zero's. Otherwise the default will be restored.

By default, all files opened for sequential access are positioned at their beginning. It is sometimes necessary or convenient to open at the END-OF-FILE so that a write will append to the existing data. If *apnd* is **.true.** then files opened subsequently on any logical unit will be positioned at their end upon opening. A value of **.false.** will restore the default behavior.

Many systems provide an automatic association of global names with fortran logical units when a program is run. There is no such automatic association in f77. However, if the argument *prefix* is a non-blank string, then names of the form **prefix**NN will be sought in the program environment. The value associated with each such name found will be used to open logical unit NN for formatted sequential access. For example, if f77 program *myprogram* included the call

```
call ioinit (.true., .false., .false., 'FORT', .false.)
```

then when the following C-Shell sequence:

```
% setenv FORT01 mydata
% setenv FORT12 myresults
% myprogram
```

or the following Bourne-Shell sequnece:

```
$ FORT01=mydata
$ FORT02=myresults
$ export FORT01 FORT02
$ myprogram
```

would result in logical unit 1 opened to file *mydata* and logical unit 12 opened to file *myresults*. Both files would be positioned at their beginning. Any formatted output would have column 1 removed and interpreted as carriage control. Embedded and trailing blanks would be ignored on input.

If the argument *vrbose* is **.true.** then *ioinit* will report on its activity.

The internal flags are stored in a labeled common block with the following definition:

```
integer*2 ieof, ictl, ibzr
common /ioiflg/ ieof, ictl, ibzr
```

FILES

/usr/lib/libI77.a f77 I/O library

SEE ALSO

getarg(3F), getenv(3F),

BUGS

Prefix can be no longer than 30 characters. A pathname associated with an environment name can be no longer than 255 characters.

The "+" carriage control does not work.

NAME

kill – send a signal to a process

SYNTAX

function kill (pid, signum)
integer pid, signum

DESCRIPTION

Pid must be the process id of one of the user's processes. *Signum* must be a valid signal number (see sigvec(2)). The returned value will be 0 if successful; an error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

kill(2), sigvec(2), signal(3F), fork(3F), perror(3F)

NAME

len – return length of Fortran string

SYNTAX

```
character*N ch
integer i
i = len(ch)
```

DESCRIPTION

Len returns the length of string *ch*.

NAME

link – make a link to an existing file

SYNTAX

function link (name1, name2)
character*(*) name1, name2

integer function symlnk (name1, name2)
character*(*) name1, name2

DESCRIPTION

Name1 must be the pathname of an existing file. *Name2* is a pathname to be linked to file *name1*. *Name2* must not already exist. The returned value will be 0 if successful; a system error code otherwise.

Symlnk creates a symbolic link to *name1*.

FILES

/usr/lib/libU77.a

SEE ALSO

link(2), symlink(2), perror(3F), unlink(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <*sys/param.h*>.

NAME

loc – return the address of an object

SYNTAX

function loc (arg)

DESCRIPTION

The returned value will be the address of *arg*.

FILES

/usr/lib/libU77.a

NAME

log, alog, dlog, clog – Fortran natural logarithm intrinsic function

SYNTAX

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2

r2 = alog(r1)
r2 = log(r1)

dp2 = dlog(dp1)
dp2 = log(dp1)

cx2 = clog(cx1)
cx2 = log(cx1)
```

DESCRIPTION

Log return the natural logarithm of its argument, in the same type as the argument.

Alog returns the real natural logarithm of its real argument.

Dlog returns the double-precision natural logarithm of its double-precision argument.

Clog returns the complex logarithm of its complex argument.

Log works for different types, but the various forms are for programming clarity.

SEE ALSO

exp(3M).

NAME

log10, alog10, dlog10 – Fortran common logarithm intrinsic function

SYNTAX

real r1, r2
double precision dp1, dp2

r2 = alog10(r1)
r2 = log10(r1)

dp2 = dlog10(dp1)
dp2 = log10(dp1)

DESCRIPTION

Log10 returns the common logarithm of its argument, in real or double-precision type of the argument.

Alog10 returns the real common logarithm of its real argument.

Dlog returns the double-precision common logarithm of its double-precision argument.

Log works for real or double arguments, but the other forms are for programming clarity.

SEE ALSO

exp(3M).

NAME

max, max0, amax0, max1, amax1, dmax1 – Fortran maximum-value functions

SYNTAX

```
integer i, j, k, l
real a, b, c, d
double precision dp1, dp2, dp3

l = max(i, j, k)
c = max(a, b)
dp = max(a, b, c)
k = max0(i, j)
a = amax0(i, j, k)
i = max1(a, b)
d = amax1(a, b, c)
dp3 = dmax1(dp1, dp2)
```

DESCRIPTION

The maximum-value functions return the largest of their arguments (of which there may be any number). *Max* is the generic form which can be used for all data types and takes its return type from that of its arguments (which must all be of the same type). *Max0* returns the integer form of the maximum value of its integer arguments; *amax0*, the real form of its integer arguments; *max1*, the integer form of its real arguments; *amax1*, the real form of its real arguments; and *dmax1*, the double-precision form of its double-precision arguments.

SEE ALSO

min(3F).

NAME

mclock – return Fortran time accounting

SYNTAX

integer i

i = mclock( )

DESCRIPTION

Mclock returns time accounting information about the current process and its child processes. The value returned is the sum of the current process's user time and the user and system times of all child processes.

SEE ALSO

times(2), clock(3C), system(3F).

NAME

min, min0, amin0, min1, amin1, dmin1 – Fortran minimum-value functions

SYNTAX

```
integer i, j, k, l
real a, b, c, d
double precision dp1, dp2, dp3

l = min(i, j, k)
c = min(a, b)
dp = min(a, b, c)
k = min0(i, j)
a = amin0(i, j, k)
i = min1(a, b)
d = amin1(a, b, c)
dp3 = dmin1(dp1, dp2)
```

DESCRIPTION

The minimum-value functions return the minimum of their arguments (of which there may be any number). *Min* is the generic form which can be used for all data types and takes its return type from that of its arguments (which must all be of the same type). *Min0* returns the integer form of the minimum value of its integer arguments; *amin0*, the real form of its integer arguments; *min1*, the integer form of its real arguments; *amin1*, the real form of its real arguments; and *dmin1*, the double-precision form of its double-precision arguments.

SEE ALSO

max(3F).

NAME

mod, amod, dmod – Fortran remaindering intrinsic functions

SYNTAX

```
integer i, j, k
real r1, r2, r3
double precision dp1, dp2, dp3

k = mod(i, j)

r3 = amod(r1, r2)
r3 = mod(r1, r2)

dp3 = dmod(dp1, dp2)
dp3 = mod(dp1, dp2)
```

DESCRIPTION

Mod returns the integer remainder of its first argument divided by its second argument, in the real or double-precision type of the arguments.

Amod and *dmod* return the real and double-precision whole number remainder of arg1 divided by arg2.

mod works with real or double types, but the various forms are for programming clarity.

NAME

perror, gerror, ierrno – get system error messages

SYNTAX

subroutine perror (string)
character*(*) string

subroutine gerror (string)
character*(*) string

character*(*) function gerror()

function ierrno()

DESCRIPTION

Perror will write a message to fortran logical unit 0 appropriate to the last detected system error. *String* will be written preceding the standard error message.

Gerror returns the system error message in character variable *string*. *Gerror* may be called either as a subroutine or as a function.

Ierrno will return the error number of the last detected system error. This number is updated only when an error actually occurs. Most routines and I/O statements that might generate such errors return an error code after the call; that value is a more reliable indicator of what caused the error condition.

FILES

/usr/lib/libU77.a

SEE ALSO

intro(2), perror(3)

BUGS

String in the call to *perror* can be no longer than 127 characters.

The length of the string returned by *gerror* is determined by the calling program.

NOTES

UNIX system error codes are described in *intro*(2). The f77 I/O error codes and their meanings are:

100	"error in format"
101	"illegal unit number"
102	"formatted io not allowed"
103	"unformatted io not allowed"
104	"direct io not allowed"
105	"sequential io not allowed"
106	"can't backspace file"
107	"off beginning of record"
108	"can't stat file"
109	"no * after repeat count"
110	"off end of record"
111	"truncation failed"
112	"incomprehensible list input"
113	"out of free space"
114	"unit not connected"
115	"read unexpected character"
116	"blank logical input field"

117 "'new' file exists"
118 "can't find 'old' file"
119 "unknown system error"
120 "requires seek ability"
121 "illegal argument"
122 "negative repeat count"
123 "illegal operation for unit"

NAME

putc, fputc – write a character to a fortran logical unit

SYNTAX

integer function putc (char)
character char

integer function fputc (lunit, char)
character char

DESCRIPTION

These funtions write a character to the file associated with a fortran logical unit bypassing normal fortran I/O. *Putc* writes to logical unit 6, normally connected to the control terminal output.

The value of each function will be zero unless some error occurred; a system error code otherwise. See perror(3F).

FILES

/usr/lib/libU77.a

SEE ALSO

putc(3S), intro(2), perror(3F)

NAME

qsort – quick sort

SYNTAX

subroutine qsort (array, len, isize, compar)
external compar
integer*2 compar

DESCRIPTION

One dimensional *array* contains the elements to be sorted. *len* is the number of elements in the array. *isize* is the size of an element, typically -

4 for **integer** and **real**
8 for **double precision** or **complex**
16 for **double complex**
(length of character object) for **character** arrays

Compar is the name of a user supplied integer*2 function that will determine the sorting order. This function will be called with 2 arguments that will be elements of *array*. The function must return -

negative if arg 1 is considered to precede arg 2
zero if arg 1 is equivalent to arg 2
positive if arg 1 is considered to follow arg 2

On return, the elements of *array* will be sorted.

FILES

/usr/lib/libU77.a

SEE ALSO

qsort(3)

NAME

srand, rand – Fortran uniform random-number generator

SYNTAX

```
integer i
double precision x, rand

call srand(i)
x = rand( )
```

DESCRIPTION

Srand takes its integer argument as the seed of a random-number generator, the values of which are returned through successive invocations of *rand*.

The value returned by *rand* are double-precision floating point numbers evenly distributed between 0 and 1, exclusive of both 0 and 1.

SEE ALSO

rand(3C).

NAME

rename – rename a file

SYNTAX

integer function rename (from, to)
character*(*) from, to

DESCRIPTION

From must be the pathname of an existing file. *To* will become the new pathname for the file. If *to* exists, then both *from* and *to* must be the same type of file, and must reside on the same filesystem. If *to* exists, it will be removed first.

The returned value will be 0 if successful; a system error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

rename(2), perror(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <*sys/param.h*>.

NAME

anint, dnint, nint, idnint – Fortran nearest integer functions

SYNTAX

```
integer i
real r1, r2
double precision dp1, dp2

r2 = anint(r1)
i = nint(r1)

dp2 = anint(dp1)
dp2 = dnint(dp1)

i = nint(dp1)
i = idnint(dp1)
```

DESCRIPTION

Nint returns the number that is nearest to its real or double-precision argument, in the type of the argument.

Idnint returns the integer that is nearest to its double-precision argument.

Anint returns the number nearest to its real or double-precision argument, in the type of the argument.

Dnint returns the double real number nearest to its double argument.

The various forms are for programming clarity.

NAME

sign, isign, dsign – Fortran transfer-of-sign intrinsic function

SYNTAX

```
integer i, j, k
real r1, r2, r3
double precision dp1, dp2, dp3

k = isign(i, j)
k = sign(i, j)

r3 = sign(r1, r2)

dp3 = dsign(dp1, dp2)
dp3 = sign(dp1, dp2)
```

DESCRIPTION

Sign returns the magnitude of its first argument with the sign of its second argument, in the integer, real, or double-precsion type of its arguments.

ISign returns the integer magnitude of its integer arguments.

Dsign returns the double magnitude of its double arguments.

Although *sign* works for integers, reals, and doubles, the various forms are for programming clarity.

NAME

signal – change the action for a signal

SYNTAX

integer function signal(signum, proc, flag)
integer signum, flag
external proc

DESCRIPTION

When a process incurs a signal (see *signal*(3C)) the default action is usually to clean up and abort. The user may choose to write an alternative signal handling routine. A call to *signal* is the way this alternate action is specified to the system.

Signum is the signal number (see *signal*(3C)). If *flag* is negative, then *proc* must be the name of the user signal handling routine. If *flag* is zero or positive, then *proc* is ignored and the value of *flag* is passed to the system as the signal action definition. In particular, this is how previously saved signal actions can be restored. Two possible values for *flag* have specific meanings: 0 means "use the default action" (See NOTES below), 1 means "ignore this signal".

A positive returned value is the previous action definition. A value greater than 1 is the address of a routine that was to have been called on occurrence of the given signal. The returned value can be used in subsequent calls to *signal* in order to restore a previous action definition. A negative returned value is the negation of a system error code. (See *perror*(3F))

FILES

/usr/lib/libU77.a

SEE ALSO

signal(3C), kill(3F), kill(1)

NOTES

If the user signal handler is called, it will be passed the signal number as an integer argument.

NAME

sin, dsin, csin – Fortran sine intrinsic function

SYNTAX

real r1, r2
double precision dp1, dp2
complex cx1, cx2

r2 = sin(r1)

dp2 = dsin(dp1)
dp2 = sin(dp1)

cx2 = csin(cx1)
cx2 = sin(cx1)

DESCRIPTION

Sin returns the sine of its argument, in the real, double-precision, or complex type of its argument.

Dsin returns the double-precision sine of its double-precision argument.

Csin returns the complex sine of its complex arguemnt.

Sin works with double, real, and complex types, but the various forms are for programming clarity.

SEE ALSO

trig(3M).

NAME

sinh, dsinh – Fortran hyperbolic sine intrinsic function

SYNTAX

real r1, r2
double precision dp1, dp2

r2 = sinh(r1)

dp2 = dsinh(dp1)
dp2 = sinh(dp1)

DESCRIPTION

Sinh returns the hyperbolic sine of its argument, either real or double-precision, in the type of its argument.

Dsinh returns the double-precision hyperbolic sine of its double-precision argument.

Sinh works with either type, but *Dsinh* is for programming clarity.

SEE ALSO

sinh(3M).

NAME

sleep – suspend execution for an interval

SYNTAX

subroutine sleep (itime)

DESCRIPTION

Sleep causes the calling process to be suspended for *itime* seconds. The actual time can be up to 1 second less than *itime* due to granularity in system timekeeping.

FILES

/usr/lib/libU77.a

SEE ALSO

sleep(3)

NAME

sqrt, dsqrt, csqrt – Fortran square root intrinsic function

SYNTAX

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2

r2 = sqrt(r1)

dp2 = dsqrt(dp1)
dp2 = sqrt(dp1)

cx2 = csqrt(cx1)
cx2 = sqrt(cx1)
```

DESCRIPTION

Sqrt returns the square root of its argument, in the real, double-precision, or complex type of its argument.

Dsqrt returns the double-precision square root of its double-precision arguement.

Csqrt returns the complex square root of its complex argument.

Sqrt works with each type, but the other forms are available for programming clarity.

SEE ALSO

exp(3M).

NAME

stat, lstat, fstat – get file status

SYNTAX

integer function stat (name, statb)
character*(*) name
integer statb(12)

integer function lstat (name, statb)
character*(*) name
integer statb(12)

integer function fstat (lunit, statb)
integer statb(12)

DESCRIPTION

These routines return detailed information about a file. *Stat* and *lstat* return information about file *name*; *fstat* returns information about the file associated with fortran logical unit *lunit*. The order and meaning of the information returned in array *statb* is as described for the structure *stat* under *stat*(2). The "spare" values are not included.

The value of either function will be zero if successful; an error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

stat(2), access(3F), perror(3F), time(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <*sys/param.h*>.

NAME

system – issue a shell command from Fortran

SYNTAX

character*N c

call system(c)

DESCRIPTION

System causes its character argument to be given to *sh*(1) as input, as if the string had been typed at a terminal. The current process waits until the shell has completed.

SEE ALSO

sh(1), exec(2), system(3S).

NAME

tan, dtan – Fortran tangent intrinsic function

SYNTAX

```
real r1, r2
double precision dp1, dp2
r2 = tan(r1)
dp2 = dtan(dp1)
dp2 = tan(dp1)
```

DESCRIPTION

Tan returns the tangent of its argument, in the type of its argument.

Dtan returns the double-precision tangent of its double-precision argument.

Tan works with real or double-precision arguments, but *Dtan* is for programming clarity.

SEE ALSO

trig(3M).

NAME

tanh, dtanh – Fortran hyperbolic tangent intrinsic function

SYNTAX

real r1, r2
double precision dp1, dp2

r2 = tanh(r1)

dp2 = dtanh(dp1)
dp2 = tanh(dp1)

DESCRIPTION

Tanh returns the hyperbolic tangent of its argument, in the type of its argument.

Dtanh returns the double-precision hyperbolic tangent of its double precision argument.

Tanh works with real or double-precision arguments, but *dtanh* is for programming clarity.

SEE ALSO

sinh(3M).

NAME

time, ctime, ltime, gmtime – return system time

SYNTAX

integer function time()

character*(*) function ctime (stime)
integer stime

subroutine ltime (stime, tarray)
integer stime, tarray(9)

subroutine gmtime (stime, tarray)
integer stime, tarray(9)

DESCRIPTION

Time returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds. This is the value of the UNIX system clock.

Ctime converts a system time to a 24 character ASCII string. The format is described under *ctime*(3). No 'newline' or NULL will be included.

Ltime and *gmtime* disect a UNIX time into month, day, etc., either for the local time zone or as GMT. The order and meaning of each element returned in *tarray* is described under *ctime*(3).

FILES

/usr/lib/libU77.a

SEE ALSO

ctime(3), itime(3F), idate(3F), fdate(3F)

NAME

ttynam, isatty – find name of a terminal port

SYNOPSIS

character*(*) function ttynam (lunit)

logical function isatty (lunit)

DESCRIPTION

Ttynam returns a blank padded path name of the terminal device associated with logical unit *lunit*.

Isatty returns **.true.** if *lunit* is associated with a terminal device, **.false.** otherwise.

FILES

/dev/*
/usr/lib/libU77.a

DIAGNOSTICS

Ttynam returns an empty string (all blanks) if *lunit* is not associated with a terminal device in directory '/dev'.

NAME

unlink – remove a directory entry

SYNTAX

integer function unlink (name)
character*(*) name

DESCRIPTION

Unlink causes the directory entry specified by pathname *name* to be removed. If this was the last link to the file, the contents of the file are lost. The returned value will be zero if successful; a system error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

unlink(2), link(3F), filsys(5), perror(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <*sys/param.h*>.

NAME

wait – wait for a process to terminate

SYNTAX

integer function wait (status)
integer status

DESCRIPTION

Wait causes its caller to be suspended until a signal is received or one of its child processes terminates. If any child has terminated since the last *wait,* return is immediate; if there are no children, return is immediate with an error code.

If the returned value is positive, it is the process ID of the child and *status* is its termination status (see *wait*(2)). If the returned value is negative, it is the negation of a system error code.

FILES

/usr/lib/libU77.a

SEE ALSO

wait(2), signal(3F), kill(3F), perror(3F)

RATFOR — A Preprocessor for a Rational FORTRAN

This document is based on a paper by Brian W. Kernighan of Bell Laboratories. It supplements the ratfor(1) pages of the **ROS Reference Manual (9010)**.

1. INTRODUCTION

Although often considered clumsy, FORTRAN is the closest thing to a universal programming language. FORTRAN is often the most "efficient" language available, particularly for programs requiring much computation. With care, it is possible to write truly portable FORTRAN programs[1].

FORTRAN's worst deficiency may be in the control flow statements — conditional branches and loops — which express the logic of the program. The conditional statements in FORTRAN are primitive. The Arithmetic IF forces the user into at least two statement numbers and two (implied) GOTO's; it leads to unintelligible code, and is eschewed by good programmers. The Logical IF is better, in that the test part can be stated clearly, but hopelessly restrictive because the statement that follows the IF can only be one FORTRAN statement (with some *further* restrictions!). And of course there can be no ELSE part to a FORTRAN IF: there is no way to specify an alternative action if the IF is not satisfied.

The FORTRAN DO restricts the user to going forward in an arithmetic progression. It is fine for "1 to N in steps of 1 (or 2 or ...)", but there is no direct way to go backwards, or even (in ANSI FORTRAN[2]) to go from 1 to N-1. And of course the DO is useless if one's problem doesn't map into an arithmetic progression.

The result of these failings is that FORTRAN programs must be written with numerous labels and branches. The resulting code is particularly difficult to read and understand, and thus hard to debug and modify.

When one is faced with an unpleasant language, a useful technique is to define a new language that overcomes the deficiencies, and to translate it into the unpleasant one with a preprocessor. **This is the approach taken with Ratfor.** (The preprocessor idea is of course not new, and preprocessors for FORTRAN are especially popular today. A recent listing [3] of preprocessors shows more than 50, of which at least half a dozen are widely available.)

2. LANGUAGE DESCRIPTION

Design

Ratfor attempts to retain the merits of FORTRAN (universality, portability, efficiency) while hiding the worst FORTRAN inadequacies. The language *is* FORTRAN except for two aspects. First, since control flow is central to any program, regardless of the specific application, the primary task of Ratfor is to conceal this part of FORTRAN from the user, by providing decent control flow structures. These structures are sufficient and comfortable for structured programming in the narrow sense of programming without GOTO's. Second, since the preprocessor must examine an entire program to translate the control structure, it is possible at the same time to clean up many of the "cosmetic" deficiencies of FORTRAN, and thus provide a language which is easier and more pleasant to read and write.

Beyond these two aspects — control flow and cosmetics — Ratfor does nothing about the host of other weaknesses of FORTRAN. Although it would be straightforward to extend it to provide character strings, for example, they are not needed by everyone, and of course the preprocessor would be harder to implement. Throughout, the design principle which has determined what should be in Ratfor and what should not has been *Ratfor doesn't know any FORTRAN.* Any language feature which would require that Ratfor really understand FORTRAN has been omitted. We will return to this point in the section on implementation.

Even within the confines of control flow and cosmetics, we have attempted to be selective in what features to provide. The intent has been to provide a small set of the most useful constructs, rather than to throw in everything that has ever been thought useful by someone.

The rest of this section contains an informal description of the Ratfor language. The control flow aspects will be quite familiar to readers used to languages like Algol, PL/I, Pascal, etc., and the cosmetic changes are equally straightforward. We shall concentrate on showing what the language looks like.

Statement Grouping

FORTRAN provides no way to group statements together, short of making them into a subroutine. The standard construction "if a condition is true, do this group of things," for example,

```
if (x > 100)
        { call error("x>100"); err = 1; return }
```

cannot be written directly in FORTRAN. Instead a programmer is forced to translate this relatively clear thought into murky FORTRAN, by stating the negative condition and branching around the group of statements:

```
        if (x .le. 100) goto 10
                call error(5hx>100)
                err = 1
                return
10      ...
```

When the program doesn't work, or when it must be modified, this must be translated back into a clearer form before one can be sure what it does.

Ratfor eliminates this error-prone and confusing back-and-forth translation; the first form *is* the way the computation is written in Ratfor. A group of statements can be treated as a unit by enclosing them in the braces { and }. This is true throughout the language: wherever a single Ratfor statement can be used, there can be several enclosed in braces. (Braces seem clearer and less obtrusive than **begin** and **end** or **do** and **end**, and of course **do** and **end** already have FORTRAN meanings.)

Cosmetics contribute to the readability of code, and thus to its understandability. The character ">" is clearer than ".GT.", so Ratfor translates it appropriately, along with several other similar shorthands. Although many FORTRAN compilers permit character strings in quotes (like "x>100"), quotes are not allowed in ANSI FORTRAN, so Ratfor converts it into the right number of H's: computers count better than people do.

Ratfor is a free-form language: statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. The example above could also be written as

```
if (x > 100) {
        call error("x>100")
        err = 1
        return
}
```

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 (Ratfor or otherwise), no braces are needed:

```
if (y <= 0.0 & z <= 0.0)
        write(6, 20) y, z
```

No continuation need be indicated because the statement is clearly not finished on the first line. In general Ratfor continues lines when it seems obvious that they are not yet done. (The continuation convention is discussed in detail later.)

Although a free-form language permits wide latitude in formatting styles, it is wise to pick one that is readable, then stick to it. In particular, proper indentation is vital, to make the logical structure of the program obvious to the reader.

The "else" Clause

Ratfor provides an **else** statement to handle the construction "if a condition is true, do this thing, *otherwise* do that thing."

```
if (a <= b)
        { sw = 0; write(6, 1) a, b }
else
        { sw = 1; write(6, 1) b, a }
```

This writes out the smaller of **a** and **b**, then the larger, and sets **sw** appropriately.

The FORTRAN equivalent of this code is circuitous indeed:

```
        if (a .gt. b) goto 10
                sw = 0
                write(6, 1) a, b
                goto 20
10      sw = 1
        write(6, 1) b, a
20      ...
```

This is a mechanical translation; shorter forms exist, as they do for many similar situations. But all translations suffer from the same problem: since they are translations, they are less clear and understandable than code that is not a translation. To understand the FORTRAN version, one must scan the entire program to make sure that no other statement branches to statements 10 or 20 before one knows that indeed this is an **if-else** construction. With the Ratfor version, there is no question about how one gets to the parts of the statement. The **if-else** is a single unit, which can be read, understood, and ignored if not relevant. The program says what it means.

As before, if the statement following an **if** or an **else** is a single statement, no braces are needed:

```
if (a <= b)
        sw = 0
else
        sw = 1
```

The syntax of the **if** statement 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. Ratfor 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.

Nested if's

Since the statement that follows an **if** or an **else** can be any Ratfor statement, this leads immediately to the possibility of another **if** or **else**. As a useful example, consider this problem: the variable **f** is to be set to − 1 if **x** is less than zero, to + 1 if **x** is greater than 100, and to 0 otherwise. Then in Ratfor, we write

```
if (x < 0)
        f = -1
else if (x > 100)
        f = +1
else
        f = 0
```

Here the statement after the first **else** is another **if-else**. Logically it is just a single statement, although it is rather complicated.

This code says what it means. Any version written in straight FORTRAN will necessarily be indirect because FORTRAN does not let you say what you mean. And as always, clever shortcuts may turn out to be too clever to understand a year from now.

Following an **else** with an **if** is one way to write a multi-way branch in Ratfor. In general the structure

```
if (...)
        - - -
else if (...)
        - - -
else if (...)
        - - -
  ...
else
        - - -
```

provides a way to specify the choice of exactly one of several alternatives. (Ratfor also provides a **switch** statement which does the same job in certain special cases; in more general situations, we have to make do with spare parts.) The tests are laid out in sequence, and each one is followed by the code associated with it. Read down the list of decisions until one is found that is satisfied. The code associated with this condition is executed, and then the entire structure is finished. The trailing **else** part handles the "default" case, where none of the other conditions apply. If there is no default action, this final **else** part is omitted:

```
if (x < 0)
        x = 0
else if (x > 100)
        x = 100
```

if-else ambiguity

There is one thing to notice about complicated structures involving nested **if**'s and **else**'s. Consider

```
if (x > 0)
        if (y > 0)
                write(6, 1) x, y
        else
                write(6, 2) y
```

There are two **if**'s and only one **else**. Which **if** does the **else** go with?

This is a genuine ambiguity in Ratfor, as it is in many other programming languages. The ambiguity is resolved in Ratfor (as elsewhere) by saying that in such cases the **else** goes with the closest previous un-**else**'ed **if**. Thus in this case, the **else** goes with the inner **if**, as we have indicated by the indentation.

It is a wise practice to resolve such cases by explicit braces, just to make your intent clear. In the case above, we would write

```
if (x > 0) {
        if (y > 0)
                write(6, 1) x, y
        else
                write(6, 2) y
}
```

which does not change the meaning, but leaves no doubt in the reader's mind. If we want the other association, we *must* write

```
if (x > 0) {
        if (y > 0)
                write(6, 1) x, y
}
else
        write(6, 2) y
```

The "switch" Statement

The **switch** statement provides a clean way to express multi-way 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
}
```

Each **case** is followed by a list of comma-separated integer expressions. The *expression* inside **switch** is compared against the case expressions *expr1, expr2,* and so on in turn until one matches, at which time the statements following that **case** are executed. If no cases match *expression,* and there is a **default** section, the statements with it are done; if there is no **default,** nothing is done. In all situations, as soon as some block of statements is executed, the entire **switch** is exited immediately. (Readers familiar with C[4] should beware that this behavior is not the same as the C **switch**.)

The "do" Statement

The **do** statement in Ratfor is quite similar to the DO statement in FORTRAN, except that it uses no statement number. The statement number, after all, serves only to mark the end of the DO, and this can be done just as easily with braces. Thus

```
do i = 1, n {
        x(i) = 0.0
        y(i) = 0.0
        z(i) = 0.0
}
```

is the same as

```
do 10 i = 1, n
        x(i) = 0.0
        y(i) = 0.0
        z(i) = 0.0
10      continue
```

The syntax is:

```
do legal-FORTRAN-DO-text
        Ratfor statement
```

The part that follows the keyword **do** has to be something that can legally go into 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), they can be used in a Ratfor **do.**

The *Ratfor statement* part will often be enclosed in braces, but as with the **if**, a single statement need not have braces around it. This code sets an array to zero:

```
do i = 1, n
        x(i) = 0.0
```

Slightly more complicated,

```
do i = 1, n
        do j = 1, n
                m(i, j) = 0
```

sets the entire array **m** to zero, and

```
do i = 1, n
        do j = 1, n
                if (i < j)
                        m(i, j) = -1
                else if (i == j)
                        m(i, j) = 0
                else
                        m(i, j) = +1
```

sets the upper triangle of **m** to – 1, the diagonal to zero, and the lower triangle to +1. (The operator == is "equals", that is, ".EQ.".) In each case, the statement that follows the **do** is logically a *single* statement, even though complicated, and thus needs no braces.

"break" and "next"

Ratfor provides a statement for leaving a loop early, and one for beginning the next iteration. **break** causes an immediate exit from the **do**; in effect it is a branch to the statement *after* the **do**. **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
}
```

break and **next** also work in the other Ratfor looping constructions that we will talk about in the next few sections.

break and **next** can be followed by an integer to indicate breaking or iterating that level of enclosing loop; thus

```
break 2
```

exits from two levels of enclosing loops, and **break 1** is equivalent to **break**. **next 2** iterates the second enclosing loop. (Realistically, multi-

level **break**'s and **next**'s are not likely to be much used because they lead to code that is hard to understand and somewhat risky to change.)

The "while" Statement

One of the problems with the FORTRAN DO statement is that it generally insists upon being done once, regardless of its limits. If a loop begins

```
DO I = 2, 1
```

this will typically be done once with I set to 2, even though common sense would suggest that perhaps it shouldn't be. Of course a Ratfor **do** can easily be preceded by a test

```
if (j <= k)
        do i = j, k {
                ---
        }
```

but this has to be a conscious act, and is often overlooked by programmers.

A more serious problem with the DO statement is that it encourages that a program be written in terms of an arithmetic progression with small positive steps, even though that may not be the best way to write it. If code has to be contorted to fit the requirements imposed by the FORTRAN DO, it is that much harder to write and understand.

To overcome these difficulties, Ratfor provides a **while** statement, which is simply a loop: "while some condition is true, repeat this group of statements". It has no preconceptions about why one is looping. For example, this routine to compute sin(x) by the Maclaurin series combines two termination criteria.

```
real function sin(x, e)
        # returns sin(x) to accuracy e, by
        # sin(x) = x - x**3/3! + x**5/5! - ...

        sin = x
        term = x

        i = 3
        while (abs(term)>e & i<100) {
                term = -term * x**2 / float(i*(i-1))
                sin = sin + term
                i = i + 2
        }

        return
        end
```

Notice that if the routine is entered with **term** already smaller than **e**, the loop will be done *zero times*, that is, no attempt will be made to compute **x******3** and thus a potential underflow is avoided. Since the test is made at the top of a **while** loop instead of the bottom, a special case disappears — the code works at one of its boundaries. (The test **i<100** is the other boundary — making sure the routine stops after some maximum number of iterations.)

As an aside, a sharp character "#" in a line marks the beginning of a comment; the rest of the line is comment. Comments and code can co-exist on the same line — one can make marginal remarks, which is not possible with FORTRAN's "C in column 1" convention. Blank lines are also permitted anywhere (they are not in FORTRAN); they should be used to emphasize the natural divisions of a program.

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 in braces.

The **while** encourages a style of coding not normally practiced by FORTRAN programmers. For example, suppose **nextch** is a function which returns the next input character both as a function value and in its argument. Then a loop to find the first non-blank character is just

```
while (nextch(ich) == iblank)
        ;
```

A semicolon by itself is a null statement, which is necessary here to mark the end of the **while**; if it were not present, the **while** would control the next statement. When the loop is broken, **ich** contains the first non-blank. Of course the same code can be written in FORTRAN as

```
100     if (nextch(ich) .eq. iblank) goto 100
```

but many FORTRAN programmers (and a few compilers) believe this line is illegal. The language at one's disposal strongly influences how one thinks about a problem.

The "for" Statement

The **for** statement is another Ratfor loop, which attempts to carry the separation of loop-body from reason-for-looping a step further than the **while**. A **for** statement allows explicit initialization and increment steps as part of the statement. For example, a DO loop is just

```
for (i = 1; i <= n; i = i + 1) ...
```

This is equivalent to

```
i = 1
while (i <= n) {
        ...
        i = i + 1
}
```

The initialization and increment of **i** have been moved into the **for** statement, making it easier to see at a glance what controls the loop.

The **for** and **while** versions have the advantage that they will be done zero times if **n** is less than 1; this is not true of the **do**.

The loop of the sine routine in the previous section can be re-written with a **for** as

```
for (i=3; abs(term) > e & i < 100; i=i+2) {
        term = -term * x**2 / float(i*(i-1))
        sin = sin + term
}
```

The syntax of the **for** statement is

for (*init* ; *condition* ; *increment*)
Ratfor statement

init is any single FORTRAN statement, which gets done once before the loop begins. *increment* is any single FORTRAN statement, which gets done at the end of each pass through the loop, before the test. *condition* is again anything that is legal in a logical IF. Any of *init, condition,* and *increment* may be omitted, although the semicolons *must* always be present. A non-existent *condition* is treated as always true, so **for(;;)** is an indefinite repeat. (But see the **repeat-until** in the next section.)

The **for** statement is particularly useful for backward loops, chaining along lists, loops that might be done zero times, and similar things which are hard to express with a DO statement, and obscure to write out with IF's and GOTO's. For example, here is a backwards DO loop to find the last non-blank character on a card:

```
for (i = 80; i > 0; i = i - 1)
        if (card(i) != blank)
                break
```

("!=" is the same as ".NE."). The code scans the columns from 80 through to 1. If a non-blank is found, the loop is immediately broken. (**break** and **next** work in **for**'s and **while**'s just as in **do**'s). If **i** reaches zero, the card is all blank.

This code is rather nasty to write with a regular FORTRAN DO, since the loop must go forward, and we must explicitly set up proper conditions when we fall out of the loop. (Forgetting this is a common error.) Thus:

```
      DO 10 J = 1, 80
          I = 81 - J
          IF (CARD(I) .NE. BLANK) GO TO 11
10    CONTINUE
      I = 0
11    ...
```

The version that uses the **for** handles the termination condition properly for free; **i** *is* zero when we fall out of the **for** loop.

The increment in a **for** need not be an arithmetic progression; the following program walks along a list (stored in an integer array **ptr**) until a zero pointer is found, adding up elements from a parallel array of values:

```
sum = 0.0
for (i = first; i > 0; i = ptr(i))
        sum = sum + value(i)
```

Notice that the code works correctly if the list is empty. Again, placing the test at the top of a loop instead of the bottom eliminates a potential boundary error.

The "repeat-until" statement

In spite of the dire warnings, there are times when one really needs a loop that tests at the bottom after one pass through. This service is provided by the **repeat-until**:

repeat
Ratfor statement
until (*legal FORTRAN condition*)

The *Ratfor statement* part 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 bare **repeat** is the cleanest way to specify an infinite loop. Of course such a loop must ultimately be broken by some transfer of control such as **stop**, **return**, or **break**, or an implicit stop such as running out of input with a READ statement.

As a matter of observed fact[8], the **repeat-until** statement is *much* less used than the other looping constructions; in particular, it is typically outnumbered ten to one by **for** and **while**. Be cautious about using it, for loops that test only at the bottom often don't handle null cases well.

More on break and next

break exits immediately from **do**, **while**, **for**, and **repeat-until**. **next** goes to the test part of **do**, **while** and **repeat-until**, and to the increment step of a **for**.

"return" Statement

The standard FORTRAN mechanism for returning a value from a function uses the name of the function as a variable which can be assigned to; the last value stored in it is the function value upon return. For example, here is a routine **equal** which returns 1 if two arrays are identical, and zero if they differ. The array ends are marked by the special value – 1.

```
# equal _ compare str1 to str2;
#     return 1 if equal, 0 if not
      integer function equal(str1, str2)
      integer str1(100), str2(100)
      integer i

      for (i = 1; str1(i) == str2(i); i = i + 1)
            if (str1(i) == - 1) {
                  equal = 1
                  return
            }
      equal = 0
      return
      end
```

In many languages (e.g., PL/I) one instead says

return (*expression*)

to return a value from a function. Since this is often clearer, Ratfor provides such a **return** statement — in a function F, **return**(expression) is equivalent to

{ F = expression; return }

For example, here is **equal** again:

```
# equal _ compare str1 to str2;
#     return 1 if equal, 0 if not
      integer function equal(str1, str2)
      integer str1(100), str2(100)
      integer i

      for (i = 1; str1(i) == str2(i); i = i + 1)
            if (str1(i) == - 1)
                  return(1)
      return(0)
      end
```

If there is no parenthesized expression after **return**, a normal RETURN is made. (Another version of **equal** is presented shortly.)

Cosmetics

As we said above, the visual appearance of a language has a substantial effect on how easy it is to read and understand programs. Accordingly, Ratfor provides a number of cosmetic facilities which may be used to make programs more readable.

Free-form Input

Statements can be placed anywhere on a line; long statements are continued automatically, as are long conditions in **if**, **while**, **for**, and **until**. 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 others 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 1-5 upon output. Thus

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

is converted into

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

Translation Services

Text enclosed in matching single or double quotes is converted to **nH...** but is otherwise unaltered (except for formatting — it may get split across card boundaries during the reformatting process). Within quoted strings, the backslash '\' serves as an escape character: the next character is taken literally. This provides a way to get quotes (and of course the backslash itself) into quoted strings:

```
"\\\'"
```

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 transmogrified (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.	¬	.not.

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

[	{	]	}
$(	{	$)	}

"define" Statement

Any string of alphanumeric characters can be defined as a name; thereafter, whenever that name occurs in the input (delimited by non-alphanumerics) 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.

define is typically used to create symbolic parameters:

```
define ROWS 100
define COLS 50

dimension a(ROWS), b(ROWS, COLS)

        if (i > ROWS | j > COLS) ...
```

Alternately, definitions may be written as

```
define(ROWS, 100)
```

In this case, the defining text is everything after the comma up to the balancing right parenthesis; this allows multi-line definitions.

It is generally a wise practice to use symbolic parameters for most constants, to help make clear the function of what would otherwise be mysterious numbers. As an example, here is the routine **equal** again, this time with symbolic constants.

```
define  YES     1
define  NO      0
define  EOS     -1
define  ARB     100

# equal _ compare str1 to str2;
#   return YES if equal, NO if not
    integer function equal(str1, str2)
    integer str1(ARB), str2(ARB)
    integer i

    for (i = 1; str1(i) == str2(i); i = i + 1)
        if (str1(i) == EOS)
            return(YES)
    return(NO)
    end
```

"include" Statement

The statement

```
include file
```

inserts the file found on input stream *file* into the Ratfor input in place of the **include** statement. The standard usage is to place COMMON blocks on a file, and **include** that file whenever a copy is needed:

```
subroutine x
        include commonblocks
        ...
        end

suroutine y
        include commonblocks
        ...
        end
```

This ensures that all copies of the COMMON blocks are identical

Pitfalls, Botches, Blemishes and other Failings

Ratfor catches certain syntax errors, such as missing braces, **else** clauses without an **if**, and most errors involving missing parentheses in statements. Beyond that, since Ratfor knows no FORTRAN, any errors you make will be reported by the FORTRAN compiler, so you will from time to time have to relate a FORTRAN diagnostic back to the Ratfor source.

Keywords are reserved — using **if**, **else**, etc., as variable names will typically wreak havoc. Don't leave spaces in keywords. Don't use the Arithmetic IF.

The FORTRAN **nH** convention is not recognized anywhere by Ratfor; use quotes instead.

3. IMPLEMENTATION

Ratfor was originally written in C on the UNIX operating system. The language is specified by a context-free grammar and the compiler constructed using the YACC compiler-compiler.

The Ratfor grammar is simple and straightforward, being essentially

```
prog    : stat
        | prog  stat
stat    : if (...) stat
        | if (...) stat else stat
        | while (...) stat
        | for (...; ...; ...) stat
        | do ... stat
        | repeat stat
        | repeat stat until (...)
        | switch (...) { case ...: prog ...
                         default: prog }
        | return
        | break
        | next
        | digits  stat
        | { prog }
        | anything unrecognizable
```

The observation that Ratfor knows no FORTRAN follows directly from the rule that says a statement is "anything unrecognizable". In fact most of FORTRAN falls into this category, since any statement that does not begin with one of the keywords is by definition "unrecognizable."

Code generation is also simple. If the first thing on a source line is not a keyword (like **if**, **else**, etc.) the entire statement is simply copied to the output with appropriate character translation and formatting. (Leading digits are treated as a label.) Keywords cause only slightly more complicated actions. For example, when **if** is recognized, two consecutive labels L and L+1 are generated and the value of L is stacked. The condition is then isolated, and the code

```
if (.not. (condition)) goto L
```

is output. The *statement* part of the **if** is then translated. When the end of the statement is encountered (which may be some distance away and include nested **if**'s, of course), the code

```
L       continue
```

is generated, unless there is an **else** clause, in which case the code is

```
        goto L+1
L       continue
```

In this latter case, the code

```
L+1     continue
```

is produced after the *statement* part of the **else.** Code generation for the various loops is equally simple.

One might argue that more care should be taken in code generation. For example, if there is no trailing **else**,

```
if (i > 0) x = a
```

should be left alone, not converted into

```
        if (.not. (i .gt. 0)) goto 100
        x = a
100     continue
```

But what are optimizing compilers for, if not to improve code? It is a rare program indeed where this kind of "inefficiency" will make even a measurable difference. In the few cases where it is important, the offending lines can be protected by '%'.

The use of a compiler-compiler is definitely the preferred method of software development. The language is well-defined, with few syntactic irregularities. Implementation is quite simple; the original construction took under a week. The language is sufficiently simple, however, that an *ad hoc* recognizer can be readily constructed to do the same job if no compiler-compiler is available.

C compilers are not as widely available as FORTRAN, however, so there is also a Ratfor written in itself and originally bootstrapped with the C version. The Ratfor version was written so as to translate into the portable subset of FORTRAN described in [1], so it is portable, having been run essentially without change on at least twelve distinct machines. (The main restrictions of the portable subset are: only one character per machine word; subscripts in the form *c*v± c;* avoiding expressions in places like DO loops; consistency in subroutine argument usage, and in COMMON declarations. Ratfor itself will not gratuitously generate non-standard FORTRAN.)

The Ratfor version is about 1500 lines of Ratfor (compared to about 1000 lines of C); this compiles into 2500 lines of FORTRAN. This expansion ratio is somewhat higher than average, since the compiled code contains unnecessary occurrences of COMMON declarations. The execution time of the Ratfor version is dominated by two routines that read and write cards. Clearly these routines could be replaced by machine coded local versions; unless this is done, the efficiency of other parts of the translation process is largely irrelevant.

4. EXPERIENCE

Good Things

"It's so much better than FORTRAN" is the most common response of users when asked how well Ratfor meets their needs. Although cynics might consider this to be vacuous, it does seem to be true that decent control flow and cosmetics converts FORTRAN from a bad language into quite a reasonable one, assuming that FORTRAN data structures are adequate for the task at hand.

Although there are no quantitative results, users feel that coding in Ratfor is at least twice as fast as in FORTRAN. More important, debugging and subsequent revision are much faster than in FORTRAN. Partly this is simply because the code can be *read*. The looping statements which test at the top instead of the bottom seem to eliminate or at least reduce the occurrence of a wide class of boundary errors. And of course it is easy to do structured programming in Ratfor; this self-discipline also contributes markedly to reliability.

One interesting and encouraging fact is that programs written in Ratfor tend to be as readable as programs written in more modern languages like Pascal. Once one is freed from the shackles of FORTRAN's clerical detail and rigid input format, it is easy to write code that is readable, even esthetically pleasing. For example, here is a Ratfor implementation of the linear table search discussed by Knuth [7]:

```
A(m+1) = x
for (i = 1; A(i) != x; i = i + 1)
        ;
if (i > m) {
        m = i
        B(i) = 1
}
else
        B(i) = B(i) + 1
```

A large corpus (5400 lines) of Ratfor, including a subset of the Ratfor preprocessor itself, can be found in [8].

Bad Things

The biggest single problem is that many FORTRAN syntax errors are not detected by Ratfor but by the local FORTRAN compiler. The compiler then prints a message in terms of the generated FORTRAN, and in a few cases this may be difficult to relate back to the offending Ratfor line, especially if the implementation conceals the generated FORTRAN. This problem could be dealt with by tagging each generated line with some indication of the source line that created it, but this is inherently implementation-dependent, so no action has yet been taken. Error message interpretation is actually not so arduous as might be thought. Since Ratfor generates no variables, only a simple pattern of IF's and GOTO's, data-related errors like missing DIMENSION statements are easy to find in the FORTRAN. Furthermore, there has been a steady improvement in Ratfor's ability to catch trivial syntactic errors like unbalanced parentheses and quotes.

There are a number of implementation weaknesses that are a nuisance, especially to new users. For example, keywords are reserved. This rarely makes any difference, except for those hardy souls who want to use an Arithmetic IF. A few standard FORTRAN constructions are not accepted by Ratfor, and this is perceived as a problem by users with a large corpus of existing FORTRAN programs. Protecting every line with a '%' is not really a complete solution, although it serves as a stop-gap. The best long-term solution is provided by the program Struct [9], which converts arbitrary FORTRAN programs into Ratfor.

Users who export programs often complain that the generated FORTRAN is "unreadable" because it is not tastefully formatted and contains extraneous CONTINUE statements. To some extent this can be ameliorated (Ratfor now has an option to copy Ratfor comments into the generated FORTRAN), but it has always seemed that effort is better spent on the input language than on the output esthetics.

One final problem is partly attributable to success — since Ratfor is relatively easy to modify, there are now several dialects of Ratfor. Fortunately, so far most of the differences are in character set, or in invisible aspects like code generation.

5. CONCLUSIONS

Ratfor demonstrates that with modest effort it is possible to convert FORTRAN from a bad language into quite a good one. A preprocessor is clearly a useful way to extend or ameliorate the facilities of a base language.

When designing a language, it is important to concentrate on the essential requirement of providing the user with the best language possible for a given effort. One must avoid throwing in "features" — things which the user may trivially construct within the existing framework.

One must also avoid getting sidetracked on irrelevancies. For instance it seems pointless for Ratfor to prepare a neatly formatted listing of either its input or its output. The user is presumably capable of the self-discipline required to prepare neat input that reflects his thoughts. It is much more important that the language provide free-form input so he *can* format it neatly. No one should read the output anyway except in the most dire circumstances.

Acknowledgements

C. A. R. Hoare once said that "One thing [the language designer] should not do is to include untried ideas of his own." Ratfor follows this precept very closely — everything in it has

been stolen from someone else. Most of the control flow structures are taken directly from the language C[4] developed by Dennis Ritchie; the comment and continuation conventions are adapted from Altran[10].

I am grateful to Stuart Feldman, whose patient simulation of an innocent user during the early days of Ratfor led to several design improvements and the eradication of bugs. He also translated the C parse-tables and YACC parser into FORTRAN for the first Ratfor version of Ratfor.

References

[1] B. G. Ryder, "The PFORT Verifier," *Software—Practice & Experience*, October 1974.

[2] American National Standard FORTRAN. American National Standards Institute, New York, 1966.

[3] *For-word: FORTRAN Development Newsletter*, August 1975.

[4] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., 1978.

[5] D. M. Ritchie and K. L. Thompson, "The UNIX Time-sharing System." *CACM*, July 1974.

[6] S. C. Johnson, "YACC — Yet Another Compiler-Compiler." Bell Laboratories Computing Science Technical Report #32, 1978.

[7] D. E. Knuth, "Structured Programming with goto Statements." *Computing Surveys*, December 1974.

[8] B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, 1976.

[9] B. S. Baker, "Struct — A Program which Structures FORTRAN", Bell Laboratories internal memorandum, December 1975.

[10] A. D. Hall, "The Altran System for Rational Function Manipulation — A Survey." *CACM*, August 1971.

Ridge C Programming Notes

This document was previously **Ridge C Programming Notes, Second Edition: 9014-B (FEB 84)**. It is now a section of the **ROS Programmer's Guide, part 9050.**

When programming in C on the Ridge 32, use this document to supplement the information found in: Kernighan & Ritchie, *The C Programming Language*, Prentice Hall, Inc., 1978.

CHARACTERISTICS OF C ON THE RIDGE 32

Data Structures

The C compiler uses the following Ridge 32 data types to represent data:

C type	*data type*	*bits*
char	character (unsigned)	8
int	integer (signed)	32
unsigned int	integer (unsigned)	32
unsigned short int	short integer (unsigned)	16
short int	short integer (signed)	16
unsigned long int	long integer (unsigned)	32
long int	long integer (signed)	32
float	floating point	32 *
double	double precision	64 *

* IEEE 754 standard representation of 32- and 64-bit floating point numbers

Ridge C data structures start on a 4-byte (1-word) boundary, except any structure containing a double-precision number starts on an 8-byte (2-word) boundary. Most structures are an integral number of 32-bit words in length. Structures that are members of other structures, however, start on a 1-, 2-, 4-, or 8-byte boundary corresponding to the size of the largest type within. Therefore, member structures that contain only 1-byte chars or 2-byte shorts will not be an integral number of 32-bit words in length.

To load a signed short integer into a register, both LOAD and SEH (sign-extend halfword) are required. All other data types can be loaded with one LOAD instruction.

Bit Handling

Bit fields are considered unsigned. Bit fields are assigned left to right.

Up to 31 bits can be shifted by the shift operations. Left shifts are performed with logical shift instructions. Right shifts are performed with arithmetic shift instructions, except in the case of a right shift on an unsigned target, in which case the right shift is logical.

Variable Names

Ridge C variables contain any number of characters, all of which are significant. When passed to the linker, an underscore "_" is added to the front of the name.

Thus, "ABCdef" becomes "_ABCdef". The following entry should be added to the table at the bottom of page 179 in the standard text:

Ridge any number of characters, 2 cases

Although the compiler accepts any number of characters, the C preprocessor cpp(1) restricts #define variable names to 128 characters.

Void Data Type

The void data type declares that a function has no return value, or that the return value is ignored. It may be used in a cast or in a function declaration (unless the function returns a structure). Example:

```
void func ( )                   /* declares a function    */
                                /* with no return value   */

(void) printf ("hi");           /* indicates return value */
                                /* should be ignored      */
```

ASM Statement Type

Asm is a statement type which allows the inclusion of one line of up to 50 characters of Ridge Assembler code in the C program. Multiple ASM statements are allowed in a C program. Example:

```
asm ("P2  LOAD R1, var1");
asm ("    LOAD R2, var2");
asm ("    EADD R1, R2");
```

The syntax of the included assembly statements must conform to the rules in the *Ridge Assembler Reference Manual* (part 9005).

Traps

bit of the "trapword"	*trap enabled by setting bit*
16	integer overflow
17	integer divide-by-zero
18	real overflow
19	real underflow
20	real divide-by-zero

See signal(2) in the Ridge Operating System Reference Manual for a discussion on enabling, disabling, and ignoring traps.

Order of Evaluation

The ROS C compiler pushes arguments onto the stack from left to right, except that any nested function calls are performed first (with the results of the nested calls left in temporary variables). Example:

```
fun(a, f(a=3));
```

can be thought of as:

```
temp = f(a=3);
fun (a, temp);
```

If the variable "a" were initially equal to 5, the result of this bad programming practice would be:

```
fun (3, temp);
```

not

```
fun (5, f(5));
```

as might have been expected.

Structure Assignment

Structures may be assigned, passed as arguments to functions, and returned by functions. The types of operands taking part must be the same. Other plausible operators, such as equality comparison, have not been implemented.

Structures that are Passed as Arguments

Structures that are passed as arguments are passed by value. The caller pushes the structure onto the stack, then calls the function which uses the stacked copy of the structure.

Functions Returning Structures

Functions returning structures are completely interruptable. The calling function places the address and the length of a local structure return area in the stack frame. Upon exit, the called function copies the structure which it is returning into this space.

Predefined Names

"ridge" and "unix" are pre-defined to the value "1" by the C preprocessor cpp(1).

Enumeration Type

There is a new data type analogous to the scalar type of Pascal. Add the following to the type-specifiers in the syntax on page 193 of the standard text:

enum-specifier

with syntax:

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

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

enumerator:
identifier
identifier = *constant-expression*

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, winedark };
...
enum color *cp, col;
```

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 identifiers in the enum-list are declared as constants, and may appear wherever constants are required. If no enumerators with = appear, then the values of the 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.

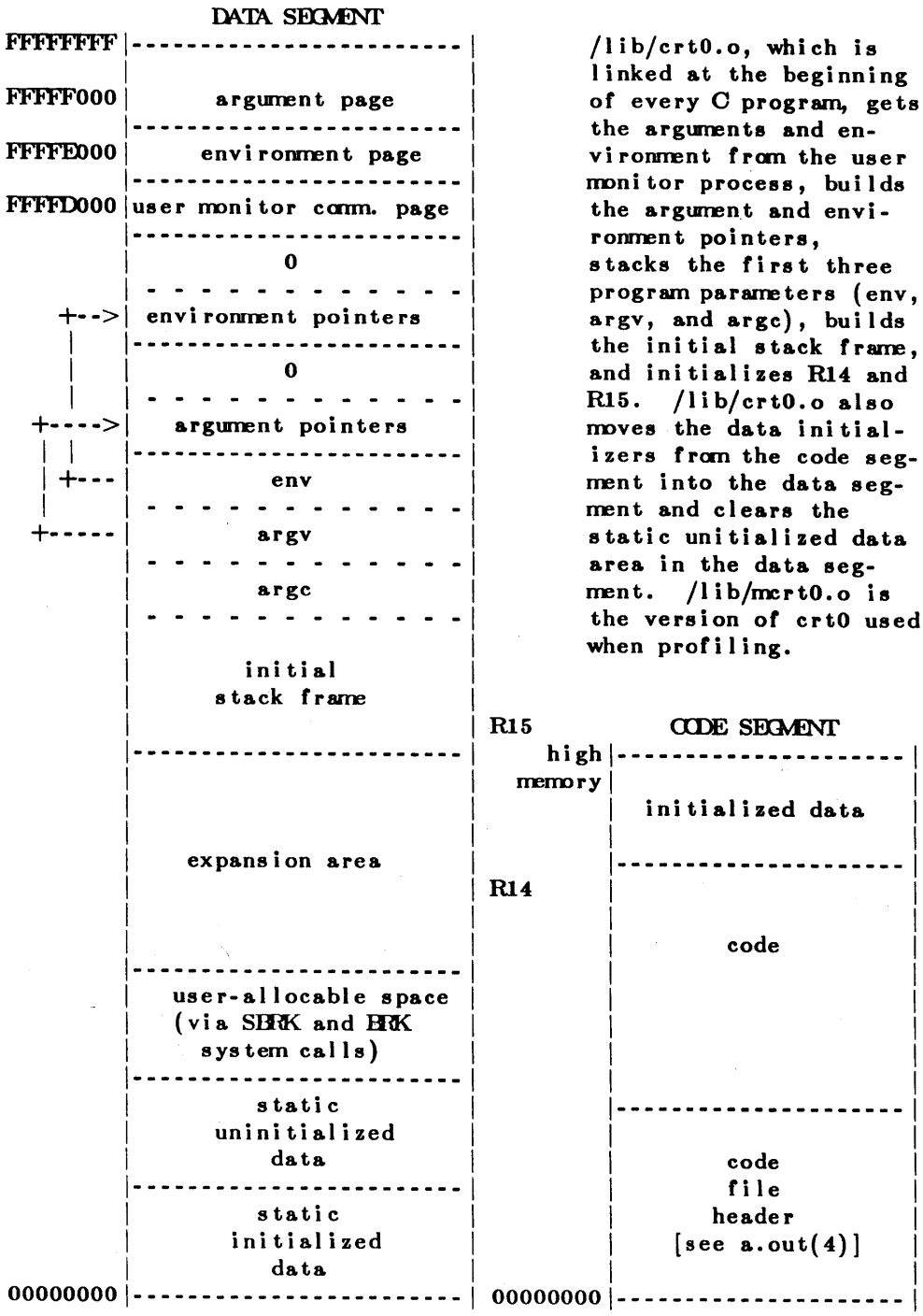
Enumeration tags and constants must all be distinct, and, unlike structure tags and members, are drawn from the same set as the ordinary identifiers.

Objects of a given enumeration type are regarded as having a type distinct from objects of all other types, and *lint*(1) flags type mismatches. In the Ridge implementation, all enumeration variables are treated as if they were **int**.

C RUNTIME ENVIRONMENT

Code and Data Segments

The separate data and code segments are arranged as follows:

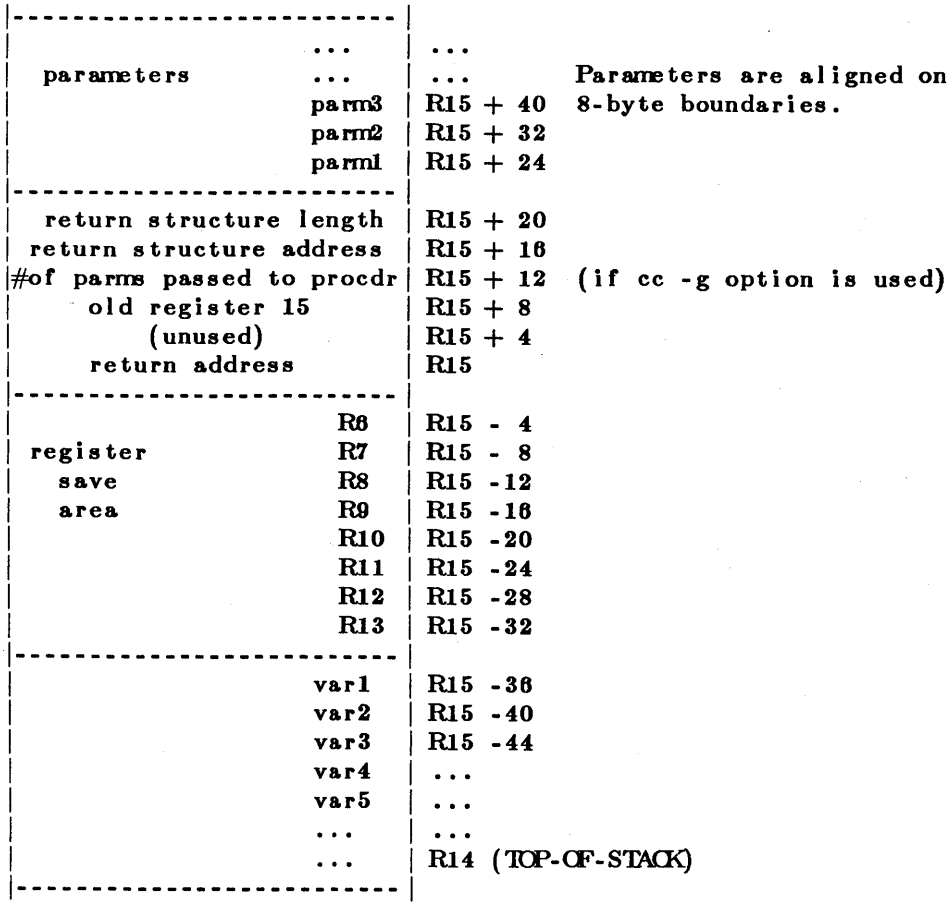


Use of General Registers

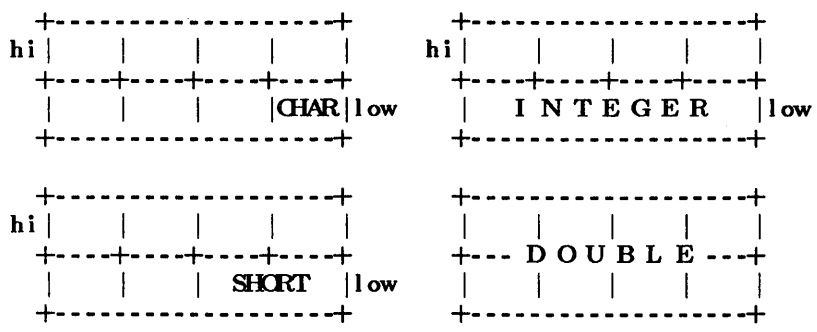
R0 through R5 - scratch registers
R6 through R13 - register variables
R14 - top-of-stack pointer
R15 - current frame pointer

Stack Frames

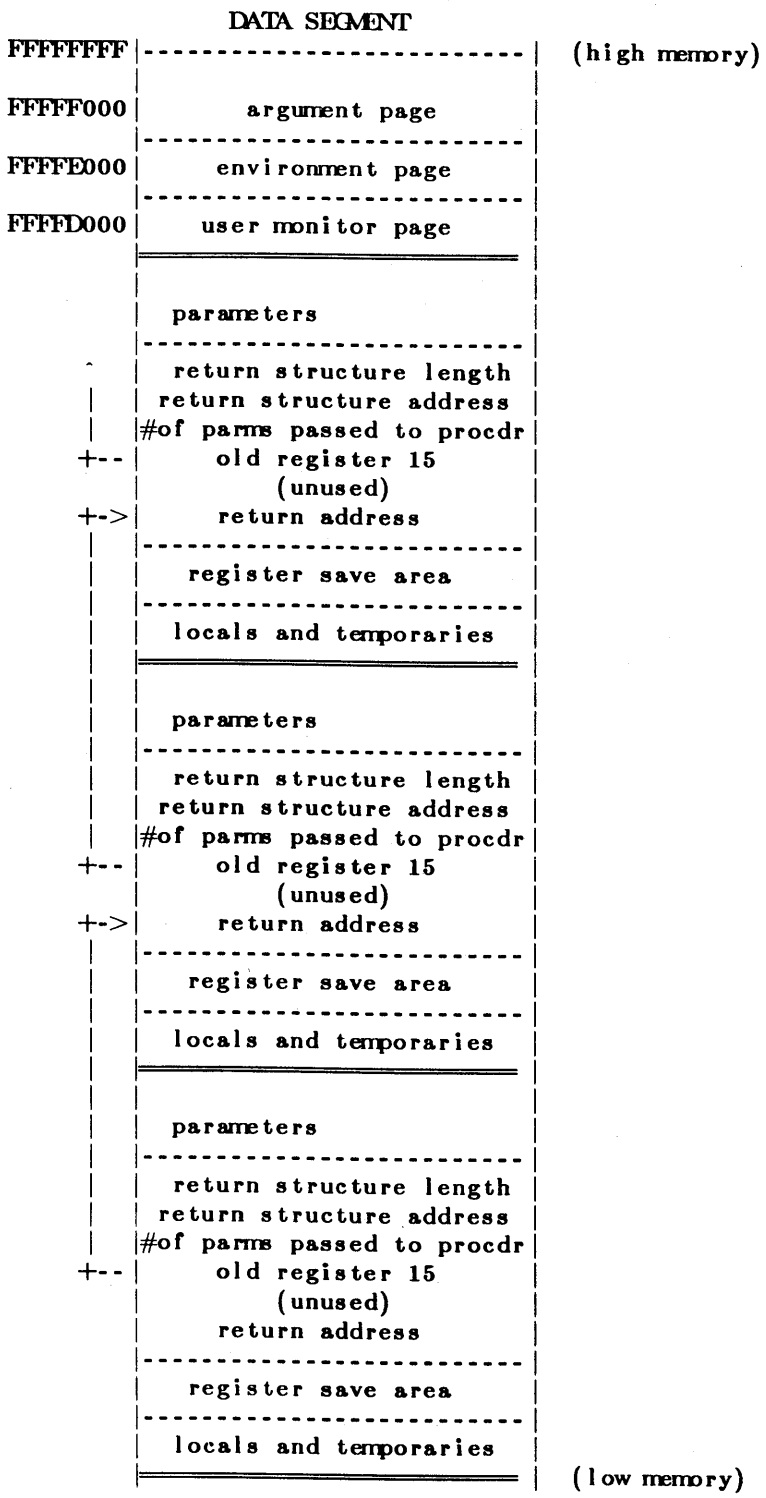
With each procedure call, the current runtime environment is recorded and pushed onto the data stack in a structure called the stack frame. The stack frame is arranged as follows:



the 8-byte C parm:



A stack of environments, therefore, has the form:



Up to eight variables per function can be defined to reside in a register. "Floats" and "doubles", however, are defined as regular stack variables even if the programmer tries to define them as register variables. If more than 8 variables are defined as register variables, only the first eight will actually be stored in registers; the excess will become regular stack variables. When a procedure is called and the current environment is stored on the stack frame, the registers containing variables also must be stored. One of two conventions apply to the method of storing the register variables.

By default, C procedure calls are compiled as follows:

```
            store R11,R14,0          ;Store return address in R14.
            store R15,R14,8          ;Save previous R15 in (R14 + 8).
            move  R15,R14            ;Point R15 to new stack frame.
            laddr R14,R14,x          ;Add x to the top-of-stack pointer,
                                     ;where x equals number of bytes
                                     ;to represent new environment.
            br regsave               ;Now the registers must be saved.

regreturn:

            ...code of the procedure...

                                     ;Now it't time to return to caller.
            move  R14,R15            ;Restore previous Top-Of-Stack pointer.
            load  R15,R14,8          ;Restore previous frame pointer.
            load  R11,R14,0          ;Load R11 with return address.
            ret   R11,R11            ;Return to calling function.
regsave:
            store R13,R15,-32        ;Save register variables in register-save area
            store R12,R15,-28        ;Save another register variable.
            br regreturn             ;
```

Ridge Pascal Reference Manual

TABLE of CONTENTS

(continued)

This manual used to be part 9003.
It is now incorporated into the ROS Programmer's Guide (9050).

PREFACE

This manual documents the Ridge Pascal language, which is based on the standard language as defined by Jensen and Wirth in the "Pascal User Manual and Report." The Ridge language shares various modifications to the base language, including traditional improvements to case statements, character synonyms, comments, and declarations, with other Pascal implementations. These and other changes arose from the desire for performance trade-offs and the need to meet implementation requirements, creating a language suitable for production.

Since a knowledge of Pascal on the part of the reader is assumed, the differences between the Jensen-Wirth language and the Ridge language are documented in this manual but not the Pascal language in its entirety.

This manual is divided into three sections:

- Ridge Pascal Language Notes
- The Pascal Runtime Environment
- An Example

The first section describes Ridge Pascal by listing the differences between it and the Jensen-Wirth language. Topics are arranged alphabetically.

The second section describes the Pascal runtime environment. Much of this information is pictorial: memory diagrams are provided that illustrate the relationships among the various components of a Pascal user process running under the Ridge Operating System (ROS).

The third section gives an example of how to write an assembly language routine that can be called by a Pascal program.

SECTION 1

RIDGE PASCAL LANGUAGE NOTES

INTRODUCTION

This section describes the Ridge Pascal language by citing the differences between it and standard Pascal (as defined in Kathleen Jensen and Nicklaus Wirth's "Pascal User Manual and Report").

The reader is referred to the Jensen/Wirth book, second edition, Springer-Verlag, 1975, for a detailed discussion of the base language.

The following list gives an overview of where Ridge Pascal differs from standard Pascal. The list is in alphabetical order for easy reference, and each item is explained in detail in the remainder of this section.

- Case Statements
- Character Synonyms
- Comments
- Compiler Options
- Declarations
- External Procedures and Functions
- Files
    - EOF and EOLN
    - File Manipulations
        - OpenFile
        - CloseFile
        - FileStatus
    - File Types

- GET
- PUT
- READ
- RESET
- REWRITE
- Standard Predefined Files
- WRITE

- GOTO Statements
- Identifiers
- Mixed Mode Expressions
- Numbers
  - Integers
  - Reals

- PACKED Types
- PACK and UNPACK
- Procedures and Functions as Parameters
- Reserved Words
- String Literals
- Strings
  - How to use
  - NewString

- Types

LISTING OF DIFFERENCES

Case Statements

In standard Pascal, if there is no case label equal to the value of the case expression, the action of the case statement is undefined. In Ridge Pascal, however, the statement immediately following the case statement is selected for execution.

The case statement has an optional "otherwise" case label. The reserved word "otherwise" may be affixed to the last case alternative rather than a case label, causing control to be transferred to this last alternative in the event of no prior match with other case labels.

Character Synonyms

The following character synonyms are recognized by the Ridge Pascal compiler:

- "|" can be substituted for "or".
- "&" can be substituted for "and".
- "~" can be substituted for "not".

Comments

In Ridge Pascal, the symbols "(*" and "*)" may be used to delimit comments; the standard symbols "{" and "}" may also be used. Comment delimiters must be matched; that is, if a command starts with "{", then it must end with "}"; if it starts with "(*", then it must end with "*)". Comments having the the same delimiters may not be nested. All text appearing between delimiters is ignored by the compiler; however, if the first symbol after the first delimiter is "$", the comment is interpreted as a compiler option (see Compiler Options).

Compiler Options

Compiler options are communicated to the compiler via special comments (see Comments). The following compiler options are recognized by the Ridge Pascal compiler when they follow a "$" at the beginning of a comment:

- The "E" (eject) option controls pagination of the source listing. The effect is that the next source line will appear at the top of a new page.
- The "G" option controls the starting address of the global outer block variables. This option must appear before the "program" declaration. The "G" option implies absolute addressing as opposed to relocatable addressing (see the "R" option).
    - The form of the "G" option is "G<n>" where "<n>" is a decimal integer. For example, "G16384" would cause the compiler to start allocating global variables at 16K.
    - The default is "G4096".
- The "L" option is for source listing control. This option may appear anywhere in the source program.
    - "L+" turns the listing on.
    - "L-" turns the listing off.
    - "L+" is the default.
- The "O" option instructs the compiler whether or not to optimize the object code.
    - "O+" produces optimized object code.
    - "O-" produces unoptimized object code.
    - "O+" is the default.

- The "P" option controls the packing of data. It informs the compiler that it should pack data closely, which saves data space but increases execution time. See the Runtime Environment section for information about the layout of data and the effect of packing. This option must appear before the "program" declaration.

    - "P+" causes data to be tighty packed.

    - "P-" causes nonpacking of data.

    - "P+" is the default.

- The "R" option causes the compiler to generate code in which the outer block variables are allocated in a relocatable segment rather than being assigned to absolute addresses. This option thus facilitates the construction of a program consisting of a number of separate compilations. With this type of construction, the user will not be burdened with assigning starting addresses for the separate compilations' outer block variables since the linker will perform this task.

  Accessing relocatable outer block variables generally causes a slight performance decrease in comparison to accessing absolute outer block variables. The reason for the decrease is that an extra instruction must be executed to determine the base of the separate relocatable compilations' outer block variables.

  The "R" option must appear before the "program" declaration. Additionally, it is mutually exclusive with the "G" (outer block variables starting address) and "S" (string constant starting address) options. That is, if the "R" is present, then neither a "G" nor "S" option may appear in the same compilation.

- "R+" enables relocatable addressing of global variables.
- "R-" disables relocatable addressing of global variables, i.e., causes absolute addressing.
- "R-" is the default.

- The "S" option controls the starting address from which string constants will be allocated downwards (towards lower addresses). The "S" option implies absolute addressing as opposed to relocatable addressing (see the "R" option). This option must appear before the "program" declaration.
  - The form of the "S" option is "S<n>" where "<n>" is a decimal integer.
  - "S0" is the default.

Declarations

LABEL, CONST, and TYPE declarations may appear in any order and may be repeated. However, as in standard Pascal, they may not appear after the first variable, procedure, or function declaration in the current block.

External Procedures and Functions

The "external" attribute is supported for procedures and functions. It is similar to the "forward" attribute in that it tells the compiler that only a procedure heading appears at this point. However, unlike the "forward" attribute which indicates that the body will appear later in the compilation, the "external" attribute indicates that the body has been compiled separately inside another program and will not appear in this compilation. The name of the "external" procedure will be passed on to the linker, which will resolve the reference at link time.

The names of all procedures and functions are considered global and may be referenced by other separately compiled programs.

Files

- EOF(f) and EOLN(f). EOLN is defined as EOF or (f^ = chr(13)), where "chr(13)" is the ASCII carriage return. Return characters are not, as in standard Pascal, converted to blanks. Nor, unlike standard Pascal, is EOF defined until after the first GET operation.

- File Initialization

  All file variables except the predefined variables "input", "output", and "stderr" must be explicitly opened. There are three file manipulation routines for this purpose, which, since they are not predefined, must be declared as "external." For more information on these routines, see the Ridge "Operating System Reference Manual." The declarations for the routines are as follows (the string type is described later):

```
Procedure OpenFile(
                    var f:Text ;
                    name:String ;
                    mode:Char
                   ) ; External;
Function FileStatus(var f:Text):Integer;External;
Procedure CloseFile(var f:Text);External;
```

  - The function of procedure "OpenFile" is to take a Pascal file variable and bind it to the ROS file indicated by the "name" argument. The argument "mode" must be either "R" for read access, "W" for write access, "A" for append access (writing at the end of a file), or "U" for update access (reading or writing).

  - The function "FileStatus" returns the value zero if no errors were encountered during any input/output operation on the file; otherwise, non-zero is returned.

  - The function of procedure "CloseFile" is to release the binding between the Pascal file variable, "f", and the ROS file.

- File Types. Only "Text" files (Text = File of Char) are currently supported.

- GET must only be applied to open files, otherwise the results are undefined. The Ridge Pascal GET differs from standard Pascal in that the file buffer is not defined until the first GET is performed. This facilitates interfacing with interactive files.

- PAGE outputs an ASCII form-feed, i.e., chr(12).

- PUT must only be applied to open files. Ridge Pascal PUT performs as in standard Pascal.

- READ(f, x) is defined as follows:

```
begin
  get(f) ;
  x := f^ ;
end
```

  while standard Pascal's READ(f, x) is defined as:

```
begin
  x := f^ ;
  get(f) ;
end
```

- RESET is recognized by the compiler but performs no operation at this time.

- REWRITE is recognized by the compiler but performs no operation at this time.

- Standard Predefined Files.

  The files "input", "output", and "stderr" are predefined in the sense that if they appear in the "program" declaration they will be opened automatically and bound to ROS file entities. Specifically, it will appear as if the following statements had been executed, in which "inputName" is a string variable containing the characters "input", "outputName" contains "output", and "stderrName" contains "stderr".

```
OpenFile(input, inputName, 'R') ;
penFile(output, outputName, 'W') ;
OpenFile(stderr, stderrName, 'W') ;
```

- WRITE performs as in standard Pascal.
- WRITELN outputs an ASCII carriage return, i.e., chr(13).

GOTO Statements

GOTO statements may not transfer control out of the current block--jumping out of procedures or functions is not permitted.

Identifiers

Identifiers may be of any length but only the first 16 characters are significant: identifiers which differ only after the sixteenth character position will be regarded as the same identifier. Identifiers must start with an alphabetic character (a letter), but thereafter may contain letters, digits, or underscores. Upper case characters are not distinguished from lower case characters in identifiers.

Mixed Mode Expressions

Ridge Pascal allows mixed mode expressions (e.g., INTEGER and REAL); however, a "var" parameter must be of the same type as the formal parameter.

Numbers

Integer constants in Ridge Pascal differ from standard Pascal in two respects:

- The base (radix) may be specifed.

- Embedded underscores are allowed for improved readability.

A BNF description of the allowable forms follows:

```
integer_number  ::= integer | based_integer ;

integer         ::= digit {['_'] digit} ;

based_integer   ::= base '#' extended_digit {['_']
                        extended_digit} ;

base            ::= integer ;  -- base must be in 2..36

extended_digit  ::= letter | digit ;
```

Here are some examples to illustrate based integers and the use of underscores to improve readability.

```
40_96
65_536
2_147_483_647    (* MAXINT *)
-2_147_483_648   (* MININT *)

2#11111111
2#1111_1111
8#377
16#ff
10#2_147_483_647
```

Ridge Pascal supports 32- and 64-bit real numbers, called REAL and DREAL respectively. A double real (DREAL) number is denoted in a fashion similar to the "E" notation except that a "D" or "d" is used instead. For example:

```
pi = 3.1415926535D0
bignum = 1.0D250
maxreal = 6.8056464E38
minreal = 5.8774728E-39
maxdreal = 3.595386269724630D308
mindreal = 1.112536929253601D-308
```

PACKED Types

In Ridge Pascal, the reserved word "PACKED" is accepted but has no effect. To control storage allocation, the "P" compiler option is used (see Compiler Options).

PACK and UNPACK

PACK and UNPACK are not currently supported by the Ridge compiler.

Procedures and Functions as Parameters

Ridge Pascal does not allow procedures or functions to be passed as parameters.

Reserved Words

Ridge Pascal treats upper and lower case characters identically in reserved words. The only nonstandard reserved word in Ridge Pascal is "otherwise".

String Literals

Character string literals may be a maximum of 80 characters in length.

Strings

Ridge Pascal does not have a predefined string type. However, the Pascal runtime library supports a string type via routines (these are more fully described in the Ridge "Operating System Reference Manual"). The following example illustrates how strings are currently manipulated.

The example opens a file called "data.x," does some processing, and then closes the file. The procedures and functions in the Pascal runtime library accept and return strings as defined in the type declaration section in the program. The two-step method of allocating an empty string, and then copying the characters

one-by-one into the string, should be employed since string constants cannot be assigned directly to the string.

Also, note that the procedures NewString, OpenFile, FileStatus, and CloseFile are not predefined, and must be declared as external procedures.

```
Program Example (stderr) ;

Type
        StringBody = Record
                       length : Integer ;
                       chars : Array[1..1] of Char ;
                     end ;

        String = ^StringBody ;

Var
        CharArray : Array[1..6] of Char ;
        dataFile : Text ;
        fileName : String ;
        i : Integer ;

Function NewString(length:Integer):String ; External ;
Procedure OpenFile(var f:Text ; name:String ; mode:Char) ; External ;
Procedure CloseFile(var f:Text) ; External ;
Function FileStatus(var f:Text) ; Integer ; External ;

begin
  charArray := 'data.x' ;
  fileName := NewString(6) ;
  for i := 1 to 6 do
    fileName^.chars[i] := charArray[i] ;
  OpenFile(dataFile, fileName, 'R') ;

  if FileStatus (dataFile)<>0 then
    Writeln(stderr,'cannot open data.x') ;

{
        do some processing
}
  CloseFile(dataFile) ;
end.
```

Types

Ridge Pascal differs from standard Pascal with respect to types in the following ways:

- DREAL (double REAL) is defined in addition to REAL.
- Sets. The maximum number of set elements is limited to 64. In addition, the following restriction applies to set types: in "set of 1..u", "1" and "u" (or ord(1) and ord(u)) must be in the range of zero to 63 inclusive.

The rules governing data allocation and storage alignment for variables of the various types are heavily dependent on the context of the runtime environment, as well as on the the "P" compiler option. The section on the Runtime Environment provides complete details on this subject.

SECTION 2

THE PASCAL RUNTIME ENVIRONMENT

INTRODUCTION

This section provides a fairly detailed picture of the environment in which Pascal programs perform their computations. Enough information is given for the user to perform debugging while a program is executing.

The Ridge architecture maintains separate data and code spaces, and this separation forms the basic division of information in this section. The following topics are covered:

- Data Segment Overview
    - Data Segment Memory Diagrams
        - Absolute Mode
        - Relocatable Mode
    - Stack Diagrams
    - The Mark Stack Block
    - The Display
    - The Heap
- Code Segment Overview
    - Code Segment Memory Diagrams
    - Preamble and Postamble Code
        - Procedure/Function Entry Code
        - Procedure/Function Exit Code

- Program Entry Code
- Program Exit Code

- Miscellaneous
  - Register Use Conventions
  - Procedure/Function Calling Conventions
  - Data Representation and Alignment Rules

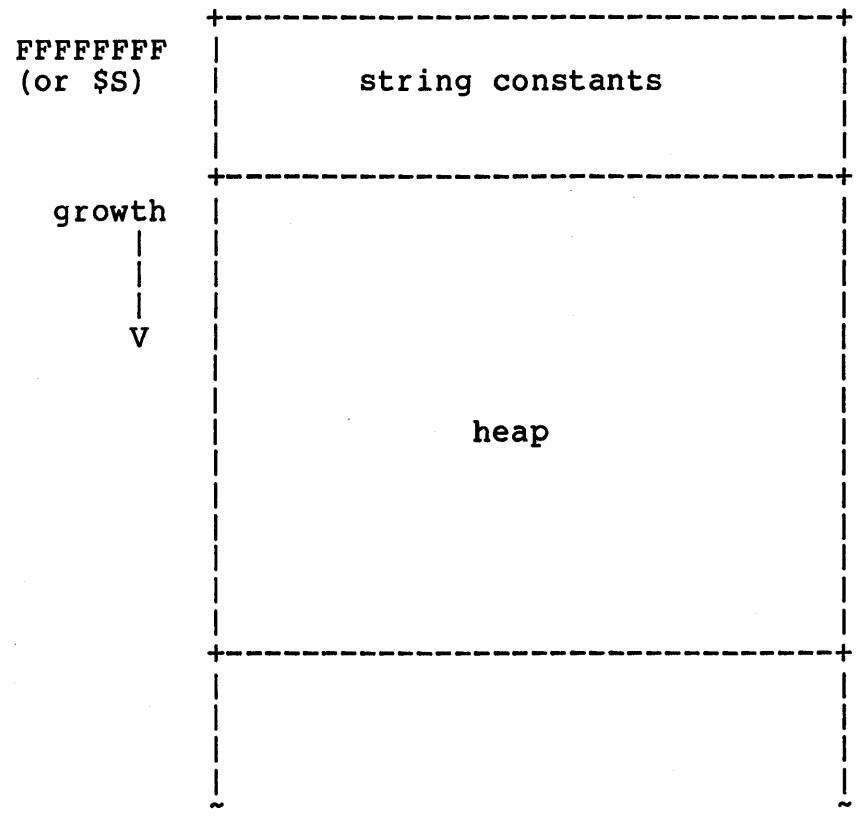
The following discussion assumes some familiarity with the Ridge architecture. Information on this subject can be found in the Ridge "Processor Reference Manual."

DATA SEGMENT OVERVIEW

Data Segment Memory Diagrams

The following two subsections provide information regarding the modes that affect memory storage: absolute and relocatable.

ABSOLUTE MODE. Figure 1 gives an overview of the data segment of a Pascal user process when the compiler has been instructed to generate absolute addressing code (see Compiler Options). The blocks are not necessarily to scale--there is a very large gap between the top of the stack and the bottom of the heap.



(continued on next page)

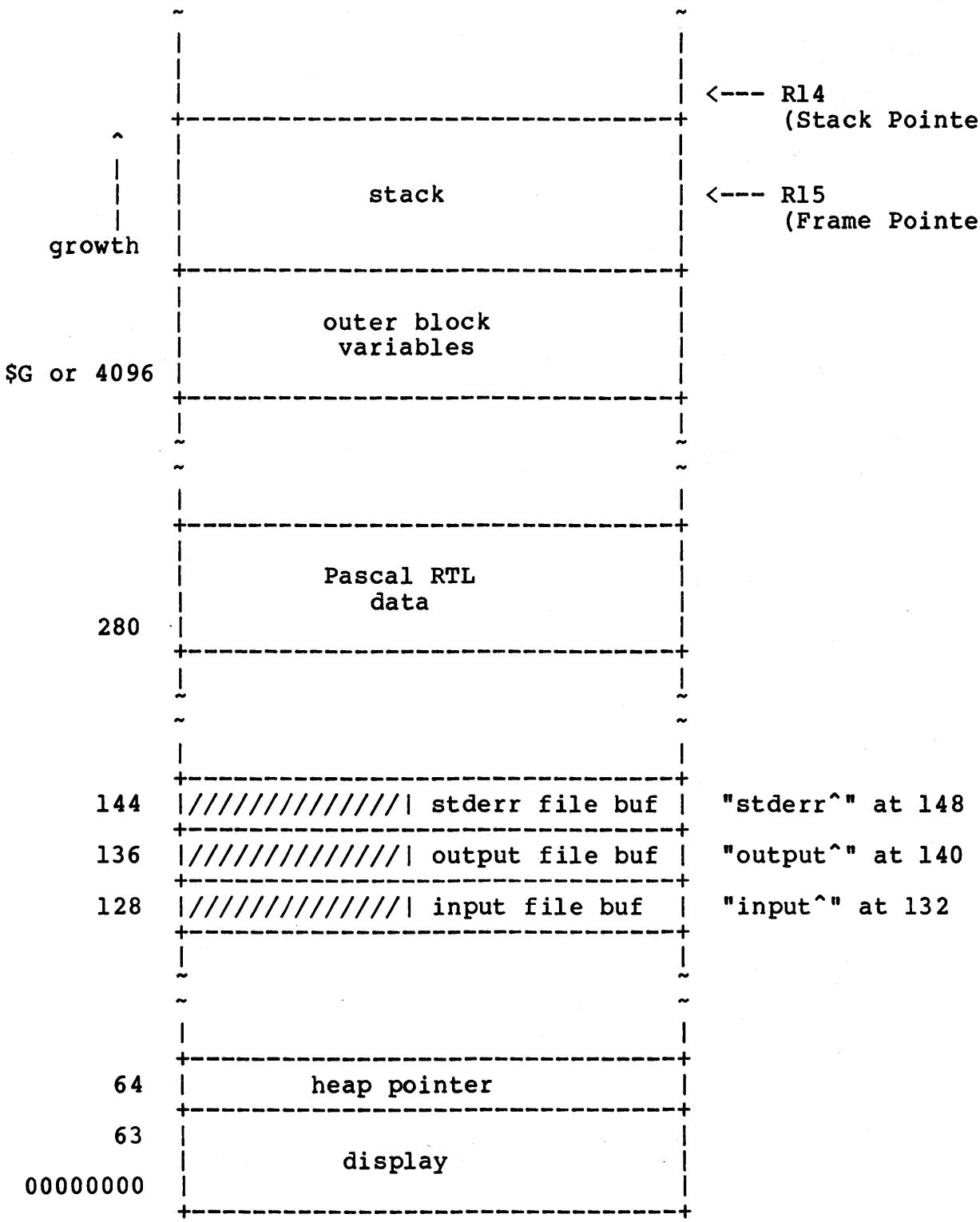
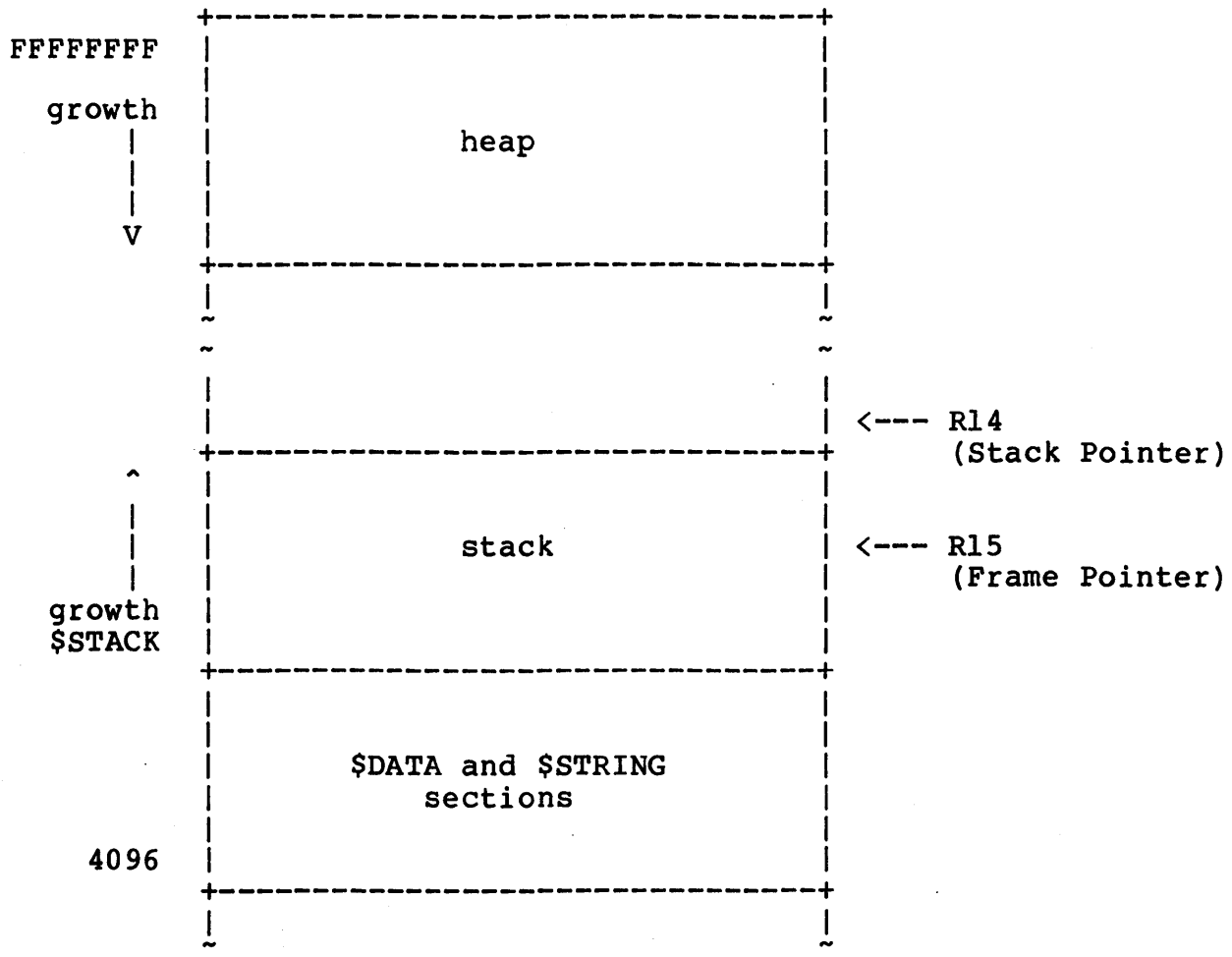


Figure 1. Data Segment: Absolute Mode

RELOCATABLE MODE. Figure 2 gives an overview of the data segment of a Pascal user process when the compiler has been instructed to generate relocatable addressing code (see Compiler Options).



(continued on next page)

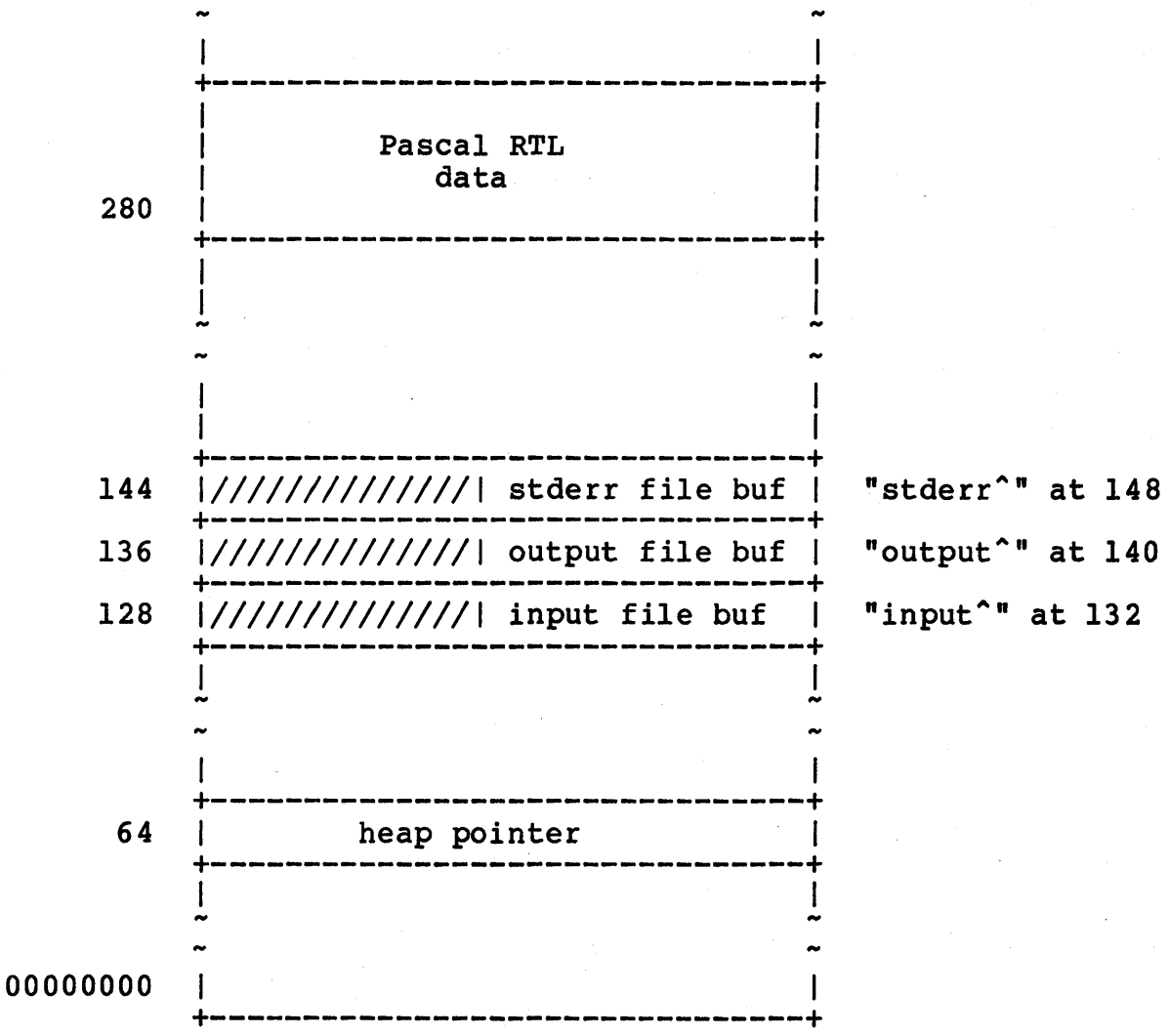


Figure 2. Data Segment: Relocatable Mode

Stack Diagrams

The Pascal runtime stack expands and contracts as procedures are entered and exited. Each time a procedure is invoked, it allocates a new piece of storage, called a stack frame, on top of the stack for its local variables, context information, parameters, and temporaries.

Figures 3 through 6 represent snapshots of the stack at four significant times in a procedure:

- Normal execution of some arbitrary procedure, "p".
- Preparing for a call to another procedure, "q".
- Entering procedure "q".
- Back in procedure "p".

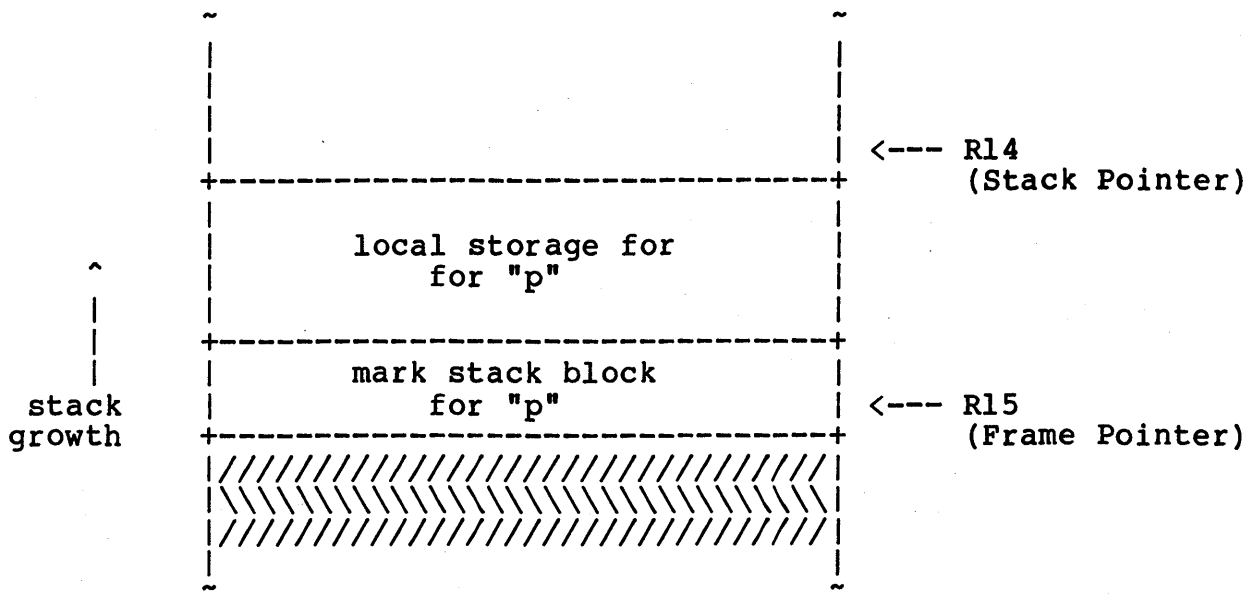


Figure 3. Normal Execution of a Procedure "p"

In Figure 3, some arbitrary procedure "p" is executing. R15, the Frame Pointer, points to the start of the stack frame for procedure "p". All of "p"s references to local data are based on the Frame Pointer.

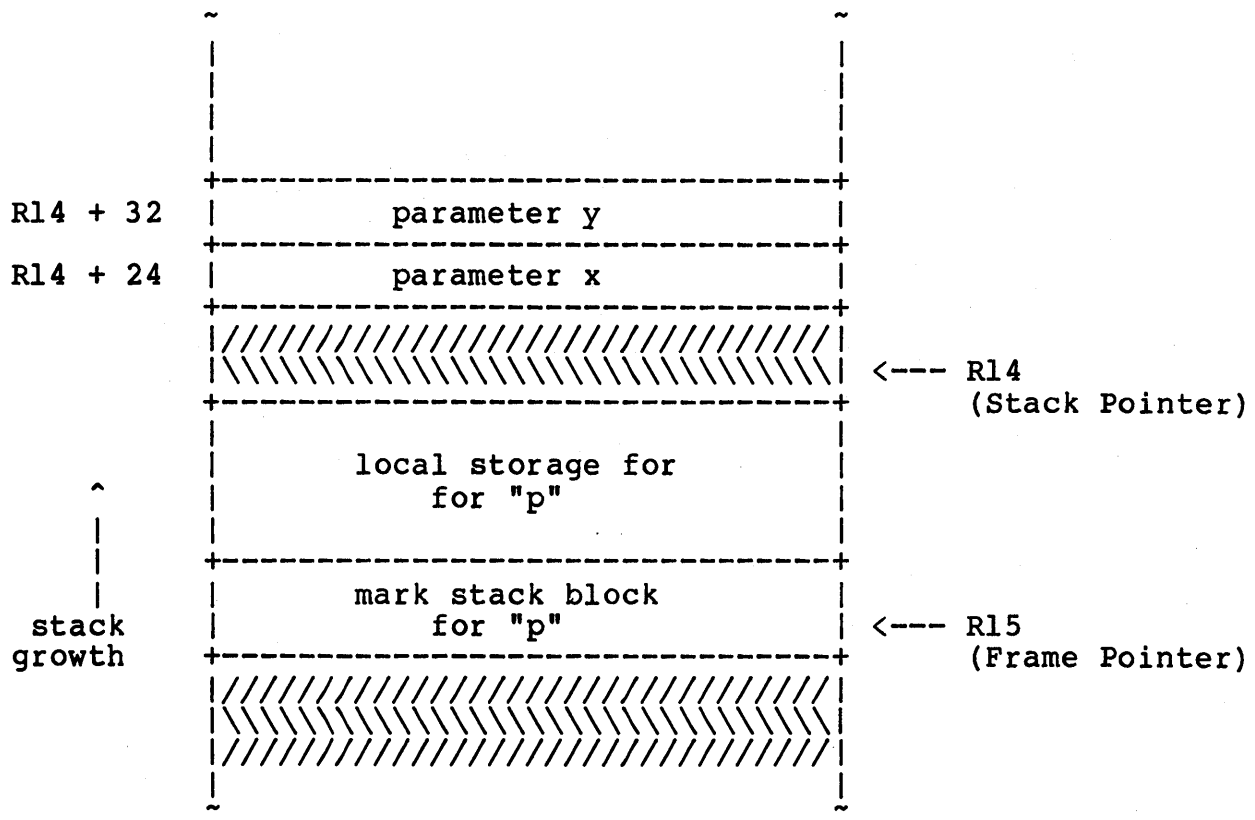


Figure 4. Procedure "p" Preparing to Call Procedure "q"

In Figure 4, procedure "p" is now preparing to call procedure "q(x, y)" by pushing the parameters onto the stack. The Stack Pointer, R14, does not actually move at this time; rather, the parameters are pushed starting at R14+24, thus leaving a gap for "q"s mark stack block.

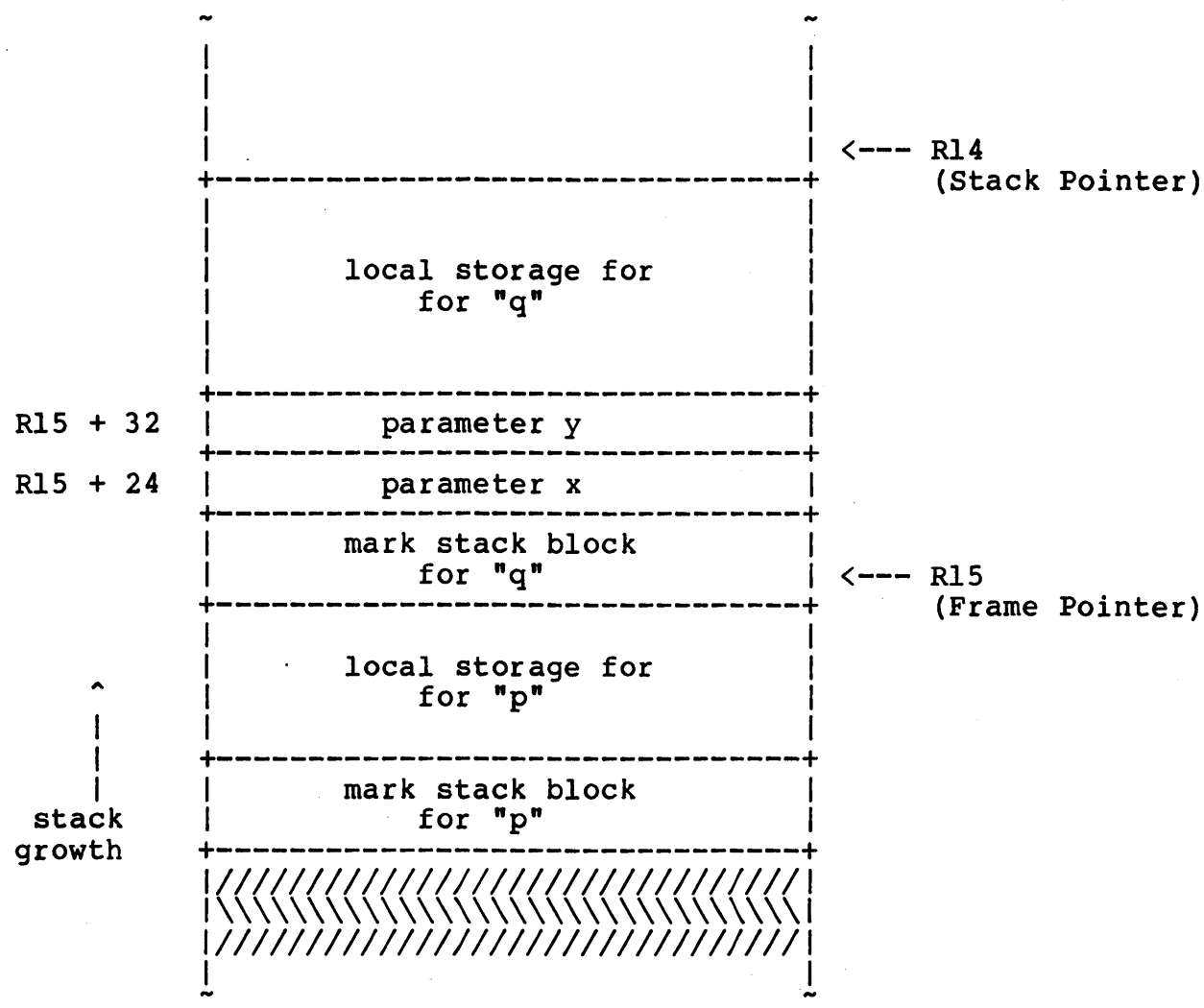


Figure 5. Entering Procedure "q"

Figure 5 shows procedure "q" immediately after it has performed its entry code and the following events have taken place:

- R15 <-- R14
- R14 <-- R14 + <framesize>
- The mark stack block is filled in.

Notice that now "q" will refer to its parameters at "R15+24" and "R15+32," while the caller referred to them at "R14+24" and "R14+32."

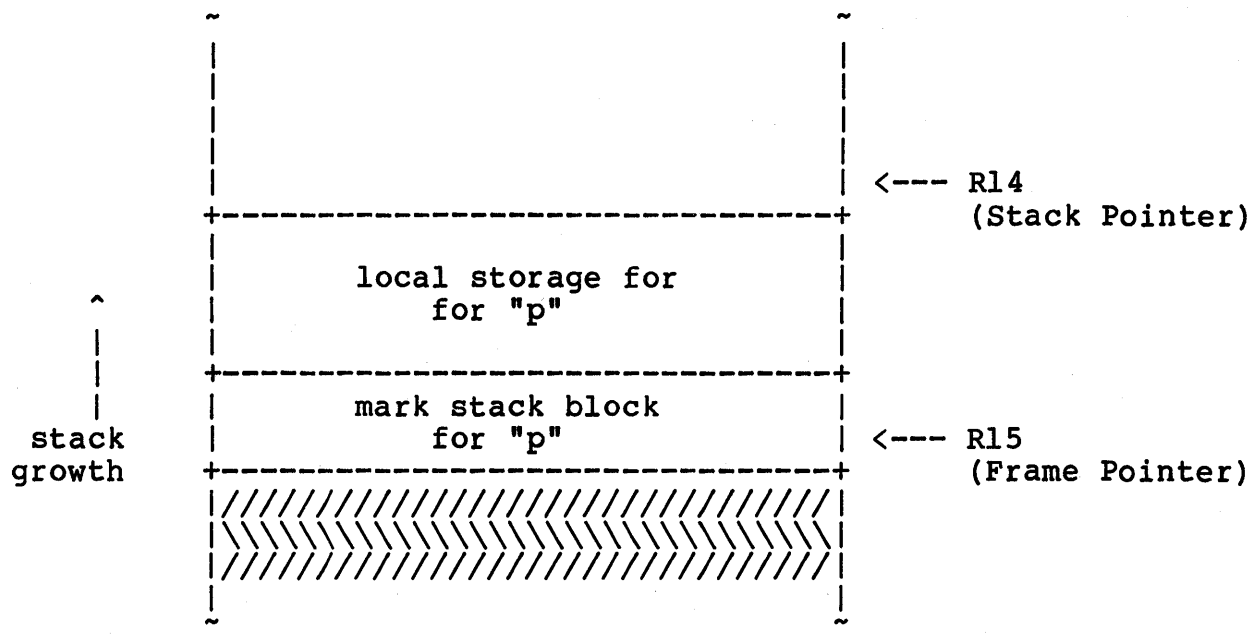


Figure 6. Return to Procedure "p"

Figure 6 shows the stack on return from "q". The stack has been returned to the state it was in just prior to the call to "q".

If "q" had been a Pascal function, then register R0 (or the register pair (R0, R1)) would contain the function value.

The Mark Stack Block

The function of the mark stack block is to store information concerning procedure and function invocations. The mark stack block, therefore, makes it possible to restore the runtime environment when a procedure or function returns to its caller.

Figure 7 shows the format of the mark stack block.

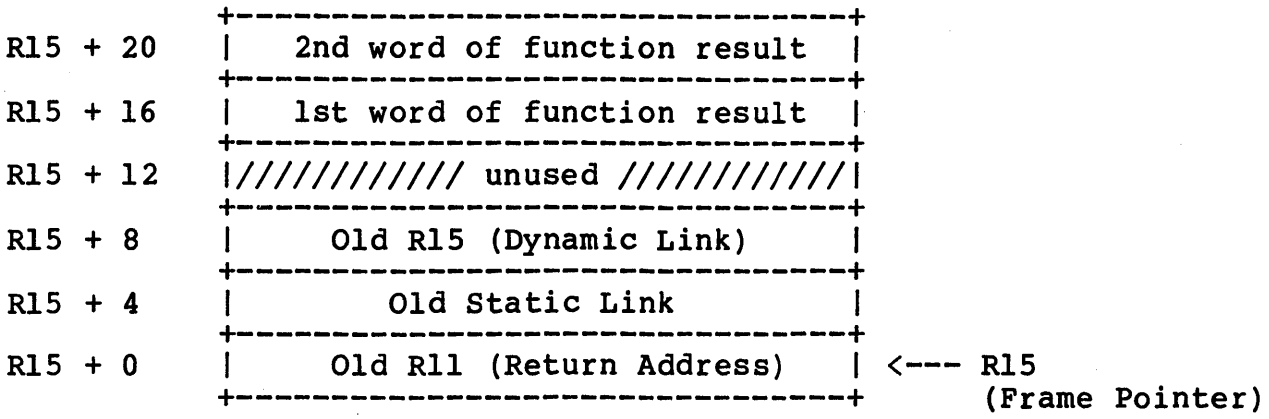


Figure 7. The Mark Stack Block Format

The Display

The display is a sixteen word block which starts at location zero. Figure 8 shows the format for the display block.

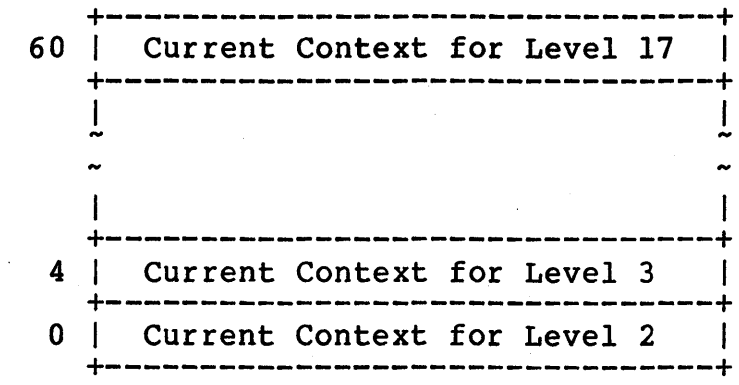


Figure 8. The Display Block Format

When the compiler has been directed to generate absolute addressing code, the display resides at absolute virtual location zero. If the compiler is generating relocatable code, then the display resides at location zero relative to the "$DATA" section.

The Heap

In the case of relocatable addressing, the heap starts at the top of the data segment and grows down towards the lower addresses; in the case of absolute addressing, it starts near the top and grows down. The allocation strategy can be described as follows:

- First, if the number of bytes asked for is "b", then round up "b" to the nearest value such that (b mod 8) = 0. This ensures double word alignment for items that follow it.
- Second, if (b mod 4096) = 0 (i.e., requesting a multiple of pages), then align the allocated block on a page boundary. If (b mod 4096) <> 0, then the requested block will only be aligned on a double word boundry.

CODE SEGMENT OVERVIEW

Code Segment Memory Diagrams

For the purposes of discussion we will assume the following program, "test." A source program compiled by the Pascal compiler is referred to as a "compilation unit."

```
Program test( ... ) ;

  Procedure a( ... ) ;
  begin  { of a }
   ...
  end ;  { of a }

  Procedure b( ... ) ;

    Procedure c( ... ) ;

      Procedure d( ... ) ;
      begin { of d }
       ...
      end ; { of d }

    begin { of c }
     ...
    end ; { of c }

  begin { of b }
   ...
  end ; { of b }

  Procedure e( ... ) ;
  begin { of e }
   ...
  end ; { of e }

begin { of test }
 ...
end : { of test }
```

Figure 9 shows how the code segment corresponding to "test" would look, and represents the output of one compilation. Execution begins at location zero.

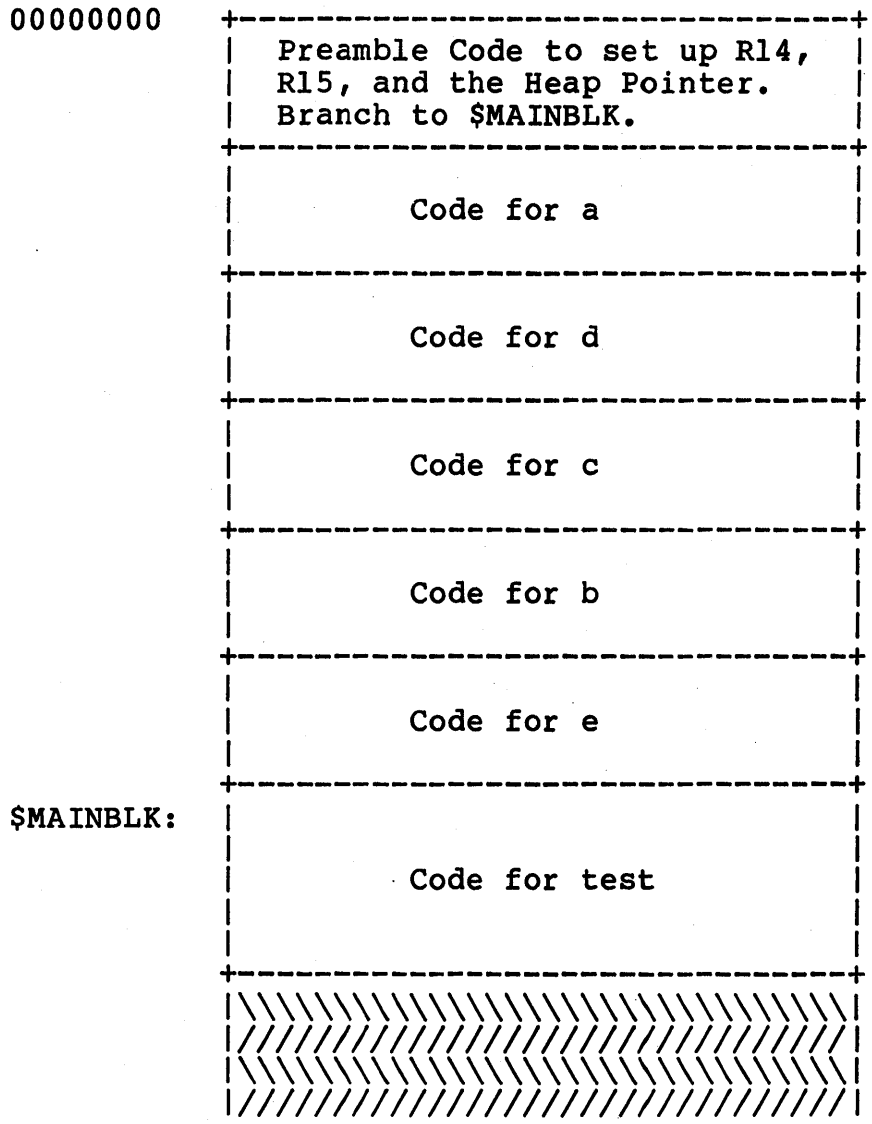


Figure 9. Code Segment

The code segment of a running user process is usually composed of several compilation units which have been consolidated by the "link" program. Figure 10 shows the overall structure of the code segment of a user process.

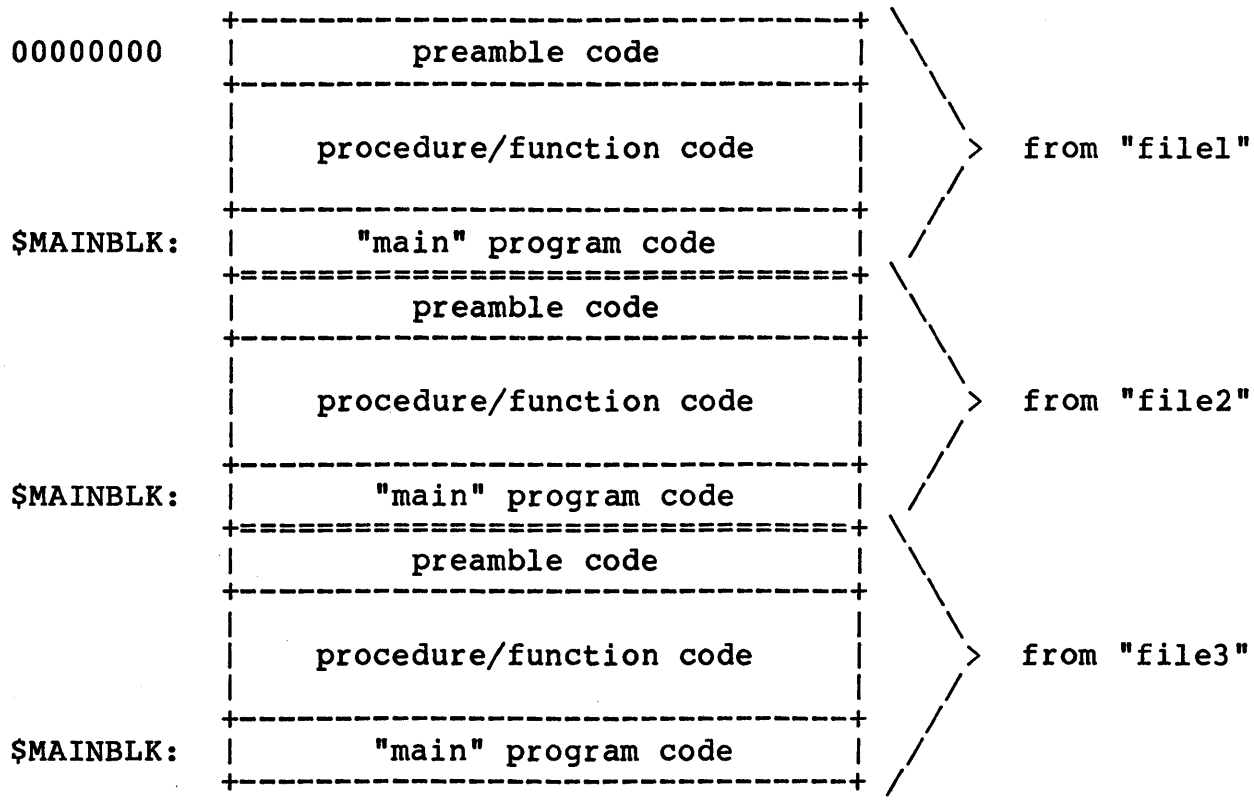


Figure 10. Overall Code Segment Structure

Note the following points:

- "link file1 file2 file3" was the command used to produce the illustrated process.
- Since the operating system passes control to the user process at location zero, execution will start at the "main" program in "file1."
- The preamble code for "file2" and "file3" is never executed.

Preamble and Postamble Code

The Pascal compiler generates code prior to the "begin" and after the "end" of a program, procedure, or function. This code performs such miscellaneous housekeeping tasks as stack adjustments and parameter manipulations. This section explains this code.

The code which follows is meant to be interpreted as a "macro" notation. The code in the boxes is generated per the Pascal-like compile-time instructions. For example:

```
FOR I := 1 TO 3 DO
        +------------------------+
        | ADD   R0,R0            |      -- double R0
        +------------------------+
```

The above "macro" code would cause the instruction "ADD R0,R0" to be generated three times.

```
IF <condition> THEN
        +------------------------+
        |   <some code>          |
        +------------------------+
ELSE
        +------------------------+
        |   <some other code>    |
        +------------------------+
```

"<some code>" would be generated if "<condition>" were to evaluate TRUE; otherwise, "<some other code>" would be generated.

In general, the "conditions" of the "macros" refer to attributes of the current program, procedure, or function being compiled.

PROCEDURE/FUNCTION ENTRY CODE. The following code is generated when a "begin" for a procedure or function is encountered.

```
IF there are calls THEN
        +------------------------+
        | STORE   R11,R14,0      |     -- store return address
        +------------------------+

IF there are no calls or loops THEN
        +------------------------+
        | MOVE    R12,R15        |     -- save dynamic link
        +------------------------+        in R12
ELSE
        +------------------------+
        | STORE   R15,R14,8      |     -- save dynamic link
        +------------------------+        in stack


IF it's an intermediate level
                    procedure THEN      -- update the display
  IF absolute mode addressing THEN
        +------------------------+
        | LOAD  R10,4*(level-1)  |     -- load old static link
        | STORE R10,R14,4        |     -- store it in the stack
        | STORE R14,4*(level-1)  |     -- store new static link
        +------------------------+
  ELSE
        +------------------------+
        | LADDR R8,$DATA,L       |     -- load address of $DATA
        | LOAD  R10,R8,4*(level-1)     -- load old static link
        | STORE R10,R14,4        |     -- store it in the stack
        | STORE R14,R8,4*(level-1)     -- store new static link
        +------------------------+

IF absolute addressing OR
        static level is not 1 THEN      -- allocate local stack
        +------------------------+        frame
        | MOVE  R15,R14          |     -- R15 <-- frame pointer
        | LADDR R14,R14,size     |     -- allocate stack frame
        +------------------------+
```

```
FOR i := 1 TO number_of_parameters DO  -- copy "value" para-
  IF non-VAR array or record THEN          meters into local
        +------------------------+         stack frame
        | LADDR Rx,R15,disp      |    -- load dst address
        | LOAD  Ry,R15,disp      |    -- load src address
        | LADDR R8,-(byte_count) |    -- # of bytes to copy
        | LOADB R9,Ry,0          |    -- load a byte
        | STOREB R9,Rx,0         |    -- store a byte
        | ADDI  Ry,1            |    -- increment src pointer
        | ADDI  Rx,1            |    -- increment dst pointer
        | LOOP  R8,1,*-12       |    -- increment and loop
        +------------------------+
```

In the code which manipulates the static link, the "level" refers to the textual level number of this procedure. The main program is considered level one; procedures which are declared at the program level are at level two; procedures inside these are considered level three; etc.

PROCEDURE/FUNCTION EXIT CODE. The following code is generated when an "end" for a procedure or function is encountered by the compiler.

```
IF it's a function THEN                -- load function value
        +------------------------+
        | LOAD  R0,R15,16        |    -- load first word
        +------------------------+
  IF it's a two word value THEN
        +------------------------+
        | LOAD  R1,R15,20        |    -- load second word
        +------------------------+


IF it's an intermediate level
                    procedure THEN     -- restore the display
  IF absolute mode addressing THEN
        +------------------------+
        | LOAD  R10,R15,4        |    -- load old static link
        | STORE R10,4*(level-1)  |    -- store it in the display
        +------------------------+
  ELSE
        +------------------------+
        | LADDR R8,$DATA,L       |    -- load address of $DATA
        | LOAD  R10,R15,4        |    -- load old static link
        | STORE R10,R8,4*(level-1)    -- store into the display
        +------------------------+
```

```
IF there were calls THEN
        +------------------------+
        | LOAD  R11,R15,0        |     -- load return address
        +------------------------+

{ always do this }
        +------------------------+
        | MOVE  R14,R15          |     -- deallocate stack frame
        +------------------------+

IF there were no (calls or loops) THEN -- restore old R15
        +------------------------+
        | MOVE  R15,R12          |     -- ... from R12
        +------------------------+
ELSE
        +------------------------+
        | LOAD  R15,R15,8        |     -- ... from stack
        +------------------------+

{ always do this }
        +------------------------+
        | RET   R11,R11          |     -- return to caller
        +------------------------+
```

For an explanation of "level" see the preceding section on Procedure/Function Entry Code.

PROGRAM ENTRY CODE. The first three boxes of code are generated when the compiler encounters the "program" declaration. Then at "$MAINBLK", in response to the "begin" of the main program, the standard Procedure/Function Entry Code is generated, followed by code which is particular to the main program.

```
IF absolute addressing mode THEN
        +-------------------------+
00000000| LADDR R10,$HEAP         |     -- load heap start address
        | STORE R10,64            |     -- store into heap pointer
        | MOVEI R14,0             |     -- initialize R14
        +-------------------------+
ELSE
        +-------------------------+
        | LADDR R14,$STACK,L      |     -- initialize stack pointer
        | MOVEI R10,0             |     -- R10 <- 0
        | STORE R10,64            |     -- initialize heap pointer
        +-------------------------+

{ always do this }
        +-------------------------+
        | MOVEI R15,0             |     -- initialize frame pointer
        | BR    $MAINBLK          |     -- branch to main program
        +-------------------------+

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        Code for all local procedures/functions goes here

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

$MAINBLK:
        +-------------------------+
        | Proc/Func Entry Code    |     -- do the same as for
        +-------------------------+        procedures

IF absolute addressing mode THEN
        +-------------------------+
        | MOVE  R15,R14           |     -- initialize frame pointer
        | LADDR R14,R14,size      |     -- allocate outer block
        +-------------------------+        variables

{ always do this }
        +-------------------------+
        | CALL  R11,SYSENTRY      |     -- initialize Pascal RTL
        +-------------------------+
```

```
IF standard "input" file present THEN
        +------------------------+
        | LADDR Rx,132           |     -- load file buffer address
        | STORE Rx,R14,24        |     -- store file buffer address
        | CALL  R11,FDF          |     -- open the file
        +------------------------+

IF standard "output" file present THEN
        +------------------------+
        | LADDR Rx,140           |     -- load file buffer address
        | STORE Rx,R14,24        |     -- store file buffer address
        | CALL  R11,FDF          |     -- open the file
        +------------------------+

IF standard "stderr" file present THEN
        +------------------------+
        | LADDR Rx,148           |     -- load file buffer address
        | STORE Rx,R14,24        |     -- store file buffer address
        | CALL  R11,FDF          |     -- open the file
        +------------------------+
```

PROGRAM EXIT CODE. The compiler generates the following code when it encounters the "end" of a main program.

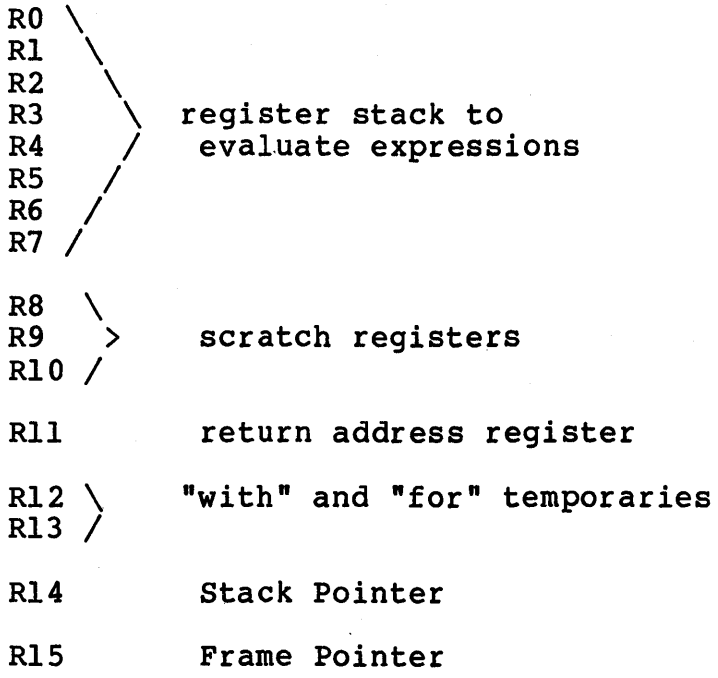
```
{ always do this }
        +------------------------+
        | MOVEI Rx,0             |     -- 0=successful completion
        | STORE Rx,R14,24        |     -- store R0
        | CALL  R11,SYSEXIT      |     -- program stops, SYSEXIT
        +------------------------+        doesn't return


        +------------------------+
        | proc/func exit code    |     -- same as standard exit
        +------------------------+        code
```

MISCELLANEOUS

This section discusses miscellaneous runtime issues that do not fit readily into one of the preceding categories. These include register use conventions, procedure/function calling conventions, and data representation and alignment rules.

Register Use Conventions



R0 (or the register pair (R0, R1)) is also used to return the result of a function call.

Procedure/Function Calling Conventions

The general rules for a procedure or function call are as follows:

p(p1, p2, ... , pN)

- Evaluate parameter 1. Store it at R14,24.
- Evaluate parameter 2. Store it at R14,32.
- Evaluate parameter N. Store it at R14,24+(N-1)*8

The process of evaluating a parameter entails the following:

- Code is generated to evaluate the parameter expression.
- Depending on whether or not the corresponding formal parameter is a "var", there are two cases:
    - "var". In this case the parameter's ADDRESS is stored at R14,24+(j-1)*8, where j is the parameter number, 1 <= j <= N.
    - Non-"var". This case is broken down into two subcases depending on whether the actual parameter is an array or a record.
        - The actual parameter is an array or a record. Pass the ADDRESS as described above.
        - The actual parameter is neither an array nor a record. The VALUE of the parameter is passed.

If "p" is a Pascal function (as opposed to a procedure) then the caller will expect to find the function value in either register R0 (or the register pair (R0,R1).

Data Representation and Alignment Rules

The compiler packing option "P+" or "P-" controls the amount of storage allocated to a variable of the following standard types:

- BOOLEAN: One byte if "P+", four bytes if "P-".
- CHAR: One byte always.

- DREAL: Eight bytes always.
- Enumerated Types: the minimum number of bytes depends on the number of identifiers in the type:
    - One byte for 1 to 255 elements.
    - Two bytes for 256 to 65,535 elements.
    - Four bytes for more than 65,535 elements.
- FILE or TEXT: Eight bytes always. The fifth byte is the file variable "f^". The first four bytes are used as a pointer to a data structure managed by the runtime library routines.
- INTEGER: Four bytes always.
- POINTER: Four bytes always.
- REAL: Four bytes always.
- SET: Eight bytes always. Set elements are allocated one bit each, starting with the most significant bit.
- Subranges:
    - If the packing option is set to "P-" then all subranges occupy four bytes.
    - If the packing option is set to "P+" then the minimum number of bytes is used. This depends on the lower and upper bounds of the subrange, as the following explains:
        - Negative lower bound always results in four bytes.
        - Lower bound of zero or more results in the following:
            - Upper bound of 1 to 255 results in 1 byte.
            - Upper bound of 256 to 65,535 results in 2 bytes.

- Upper bound of 256 to 65,535 results in 2 bytes.
- Upper bound that is more than 65,535 results in four bytes.

The rules for Ridge Pascal data alignment are as follows:

- Half-word items must be aligned on a half-word boundary, i.e., their addresses must be evenly divisible by two.
- Word items must be aligned on a word boundary, i.e., their addresses must be evenly divisible by four.
- Double-word items must be aligned on a double-word boundary, i.e., their addresses must be evenly divisible by eight.

To optimize use of space, the preceding rules should be observed. For example, in declaring variables (or fields in a record) the order of the items may have an impact on the total amount of storage used.

```
ch : Char ;      { 1 byte data }
i : Integer ;    { 3 bytes padding, 4 bytes data }
b : Boolean ;    { 1 byte data }
d : Dreal ;      { 7 bytes padding, 8 bytes data }
k : 1..1000 ;    { 2 bytes data }
```

Storage would be used more efficiently if the items were arranged as follows:

```
ch : Char ;      { 1 byte data }
b : Boolean ;    { 1 byte data }
k : 1..1000 ;    { 2 bytes data }
i : Integer ;    { 4 bytes data }
d : Dreal ;      { 8 bytes data }
```

Declarations of the following sort are also inefficient:

```
a : Array[1..100000] of Integer ;
ch : Char ;
 ...
 ...
i : Integer ;
```

An improvement would be to declare the large array last, then short offsets could be used in the code that accesses "ch", "i", and other scalar variables. Refer to the "Ridge Processor Reference Manual" for more information on this topic.

SECTION 3

AN EXAMPLE

INTRODUCTION

This section illustrates how to write assembly language programs that are Pascal callable. A program written in Ridge Pascal can be compiled into an intermediate form called P-code by the Ridge Pascal compiler, "pasc." The P-code can then be translated into an object module by the translator, "ptrans," and finally linked with other object modules by the linker, "link." (For more information on the compiling process, see the Ridge "Operating System Reference Manual.")

Included in this section are the listings for four files:

- The command file which compiles, assembles, and links the program.
- The Pascal source listing of the main program.
- The assembler listing of the compiler's generated code.
- The assembler listing of the called routines.

The key items to be observed are:

- how the assembler programs are declared in the Pascal program as "external" functions,
- how the assembler programs access their parameters and how they return their values,
- that Pascal compilation is performed with pp(1), as documented in the ROS Reference Manual (9010). See the EXAMPLE section of pp(1).

To compile and link a set of assembly functions with Pascal code:

```
rasm asmfuncs.s
pp -o prog-name asmfuncs.o prog-name.p
```

prog-name.p is the Pascal source, asmfuncs.o is the assembly object file created by rasm, and -o prog-name specifies the name of the resulting executable program.

PASCAL SOURCE LISTING

```
{
    This program reads real numbers and computes
  their square roots using Newton's method.  Two assembler
  language routines are called to manipulate parts of the
  real numbers.

    The routines are part of a suite of routines defined
  in the book "Software Manual for the Elementary Functions"
  by Cody and Waite, Prentice-Hall (1980).
}
program example(input, output) ;

var
        z : real ;
        iterations : integer ;

{
        'intxp' returns the unbiased exponent of 'x'.
}
function intxp(x : real) : integer ; external ;

{
        'setxp' returns a real number whose mantissa is
        that of 'x' and whose exponent is 'n'.
}
function setxp(x : real ; n : integer) : real ; external ;
{$E}
function sqroot(x : real) : real ;

label 99 ;
```

```
const
        EPSILON = 1.0E-30 ;
var
        i : integer ;
        yn, ynminusl : real ;

begin {******* begin of function sqroot *******}

  iterations := 0 ;
  if x = 0.0 then
    sqroot := 0.0
  else
    begin
      if x < 0.0 then
        x := -x ;
      ynminusl := setxp(x, intxp(x) div 2) ;
      while TRUE do
        begin
          yn := (ynminusl + x/ynminusl) / 2.0 ;
          iterations := iterations + 1 ;
          if abs(yn - ynminusl) <= EPSILON then
            goto 99 ;
          ynminusl := yn ;
        end ;
  99: sqroot := yn ;
    end ;
end ;  {******* end of function sqroot *******}

begin  {******* begin of program example *******}

  while not eof(input) do
    begin
      readln(input, z) ;
      writeln(output, 'sqroot(', z, ') = ', sqroot(z),
                      ', iterations = ', iterations) ;
    end ;

end.  {******* end of program example *******}
```

ASSEMBLER LISTING OF MAIN PROGRAM

```
 SOURCE LINE 41SL=P, ABS_AD=T, $S=0

     00000000  DEA0FFFFFFFF  LADDR   R10,-1
     00000006  A6A00040      STORE   R10,64
     0000000A  11E0          MOVEI   R14,0
```

```
    0000000C  11F0          MOVEI   R15,0
    0000000E  9B00FFFFFFFF  BR      $MAINBLK
SQROOT:
    00000014  A7BE0000      STORE   R11,R14,0
    00000018  A7FE0008      STORE   R15,R14,8
    0000001C  01FE          MOVE    R15,R14
    0000001E  DFEEFFFFFFFF  LADDR   R14,R14,-1
    00000024  1100          MOVEI   R0,0
    00000026  A6001004      STORE   R0,4100
SOURCE LINE 42
    0000002A  C71F0018      LOAD    R1,R15,24
    0000002E  1120          MOVEI   R2,0
    00000030  8A12FFFF      BR      R1<>R2,E3
SOURCE LINE 44
    00000034  1130          MOVEI   R3,0
    00000036  A73F0010      STORE   R3,R15,16
    0000003A  8B00FFFF      BR      L4
E3:
SOURCE LINE 46
    0000003E  1130          MOVEI   R3,0
    00000040  2A13          RCOMP   R1,R3
    00000042  5510          TESTLT  R1,0
    00000044  8E11FFFF      BR      R1<>1,E5
SOURCE LINE 47
    00000048  C74F0018      LOAD    R4,R15,24
    0000004C  2254          RNEG    R5,R4
    0000004E  A75F0018      STORE   R5,R15,24
E5:
L6:
SOURCE LINE 48
    00000052  C70F0018      LOAD    R0,R15,24
    00000056  A70E0018      STORE   R0,R14,24
    0000005A  CFEE0020      LADDR   R14,R14,32
    0000005E  A70E0018      STORE   R0,R14,24
    00000062  93B0FFFFFFFF  CALL    R11,INTXP
    00000068  0180          MOVE    R8,R0
    0000006A  5580          TESTLT  R8,0
    0000006C  0308          ADD     R0,R8
    0000006E  7301          ASRI    R0,1
    00000070  A70E0000      STORE   R0,R14,0
    00000074  CFEEFFE0      LADDR   R14,R14,-32
    00000078  93B0FFFFFFFF  CALL    R11,SETXP
    0000007E  A70F0028      STORE   R0,R15,40
    00000082  01C0          MOVE    R12,R0
VARIABLE AT 2,40 ASSIGNED TO REGISTER 12
    00000084  C71F0018      LOAD    R1,R15,24
    00000088  01D1          MOVE    R13,R1
VARIABLE AT 2,24 ASSIGNED TO REGISTER 13
W7:
SOURCE LINE 49
SOURCE LINE 51
    0000008A  010C          MOVE    R0,R12
```

```
    0000008C  011D           MOVE    R1,R13
    0000008E  0120           MOVE    R2,R0
    00000090  2612           RDIV    R1,R2
    00000092  2310           RADD    R1,R0
    00000094  DE3040000000   LADDR   R3,1073741824
    0000009A  2613           RDIV    R1,R3
    0000009C  A71F0024       STORE   R1,R15,36
SOURCE LINE 52
    000000A0  C6401004       LOAD    R4,4100
    000000A4  1341           ADDI    R4,1
    000000A6  A6401004       STORE   R4,4100
SOURCE LINE 53
    000000AA  2410           RSUB    R1,R0
    000000AC  7011           LSLI    R1,1
    000000AE  7111           LSRI    R1,1
    000000B0  DE500DA24260   LADDR   R5,228737632
    000000B6  2A15           RCOMP   R1,R5
    000000B8  5C10           TESTLE  R1,0
    000000BA  8E11FFFF       BR      R1<>1,E9
SOURCE LINE 54
    000000BE  A7CF0028       STORE   R12,R15,40
    000000C2  A7DF0018       STORE   R13,R15,24
    000000C6  8B00FFFF       BR      X2
E9:
L10:
SOURCE LINE 55
    000000CA  C70F0024       LOAD    R0,R15,36
    000000CE  01C0           MOVE    R12,R0
SOURCE LINE 56
    000000D0  8B00FFBB       BR      W7
L8:
    000000D4  A7CF0028       STORE   R12,R15,40
    000000D8  A7DF0018       STORE   R13,R15,24
X2:
SOURCE LINE 57
    000000DC  C70F0024       LOAD    R0,R15,36
    000000E0  A70F0010       STORE   R0,R15,16
L4:
SOURCE LINE 59
    000000E4  C70F0010       LOAD    R0,R15,16
    000000E8  C7BF0000       LOAD    R11,R15,0
    000000EC  01EF           MOVE    R14,R15
    000000EE  C7FF0008       LOAD    R15,R15,8
    000000F2  57BB           RET     R11,R11
SOURCE LINE 63
$MAINBLK:
    000000F4  A7BE0000       STORE   R11,R14,0
    000000F8  A7FE0008       STORE   R15,R14,8
    000000FC  01FE           MOVE    R15,R14
    000000FE  DFEEFFFFFFFF   LADDR   R14,R14,-1
    00000104  93B0FFFFFFFF   CALL    R11,SYSENTRY
    0000010A  CE00008C       LADDR   R0,140
```

```
     0000010E  A70E0018       STORE   R0,R14,24
     00000112  93B0FFFFFFFF   CALL    R11,FDF
     00000118  CE100084       LADDR   R1,132
     0000011C  A71E0018       STORE   R1,R14,24
     00000120  93B0FFFFFFF2   CALL    R11,FDF
W4:
     00000126  CE000084       LADDR   R0,132
     0000012A  C710FFFC       LOAD    R1,R0,-4
     0000012E  C7110000       LOAD    R1,R1,0
     00000132  7811           CSLI    R1,1
     00000134  0181           MOVE    R8,R1
     00000136  7811           CSLI    R1,1
     00000138  0918           OR      R1,R8
     0000013A  1B11           ANDI    R1,1
     0000013C  8611FFFF       BR      R1=1,L5
SOURCE LINE 65
     00000140  CE200084       LADDR   R2,132
     00000144  CE301000       LADDR   R3,4096
     00000148  A72E0018       STORE   R2,R14,24
     0000014C  A73E0020       STORE   R3,R14,32
     00000150  93B0FFFFFFFF   CALL    R11,RDR
     00000156  C70E0018       LOAD    R0,R14,24
     0000015A  A70E0018       STORE   R0,R14,24
     0000015E  93B0FFFFFFFF   CALL    R11,RLN
SOURCE LINE 66
     00000164  CE10008C       LADDR   R1,140
     00000168  CE20FFF8       LADDR   R2,-8
     0000016C  DE807371726F   LADDR   R8,1936814703
     00000172  A680FFF8       STORE   R8,-8
     00000176  DE806F742827   LADDR   R8,1869883431
     0000017C  A680FFFC       STORE   R8,-4
     00000180  1137           MOVEI   R3,7
     00000182  1147           MOVEI   R4,7
     00000184  A71E0018       STORE   R1,R14,24
     00000188  A72E0020       STORE   R2,R14,32
     0000018C  A73E0028       STORE   R3,R14,40
     00000190  A74E0030       STORE   R4,R14,48
     00000194  93B0FFFFFFFF   CALL    R11,WRS
     0000019A  C70E0018       LOAD    R0,R14,24
     0000019E  C6101000       LOAD    R1,4096
     000001A2  112E           MOVEI   R2,14
     000001A4  1130           MOVEI   R3,0
     000001A6  A70E0018       STORE   R0,R14,24
     000001AA  A71E0020       STORE   R1,R14,32
     000001AE  A72E0028       STORE   R2,R14,40
     000001B2  A73E0030       STORE   R3,R14,48
     000001B6  93B0FFFFFFFF   CALL    R11,WRR
     000001BC  C70E0018       LOAD    R0,R14,24
     000001C0  CE10FFF4       LADDR   R1,-12
     000001C4  DE8029203D20   LADDR   R8,689978656
     000001CA  A680FFF4       STORE   R8,-12
     000001CE  1124           MOVEI   R2,4
```

```
    000001D0  1134           MOVEI   R3,4
    000001D2  A70E0018       STORE   R0,R14,24
    000001D6  A71E0020       STORE   R1,R14,32
    000001DA  A72E0028       STORE   R2,R14,40
    000001DE  A73E0030       STORE   R3,R14,48
    000001E2  93B0FFFFFFB2   CALL    R11,WRS
    000001E8  C70E0018       LOAD    R0,R14,24
    000001EC  A70E0000       STORE   R0,R14,0
    000001F0  13E8           ADDI    R14,8
    000001F2  C6101000       LOAD    R1,4096
    000001F6  A71E0018       STORE   R1,R14,24
    000001FA  83B0FE1B       CALL    R11,SQROOT
    000001FE  111E           MOVEI   R1,14
    00000200  1120           MOVEI   R2,0
    00000202  C73EFFF8       LOAD    R3,R14,-8
    00000206  14E8           SUBI    R14,8
    00000208  A73E0018       STORE   R3,R14,24
    0000020C  A70E0020       STORE   R0,R14,32
    00000210  A71E0028       STORE   R1,R14,40
    00000214  A72E0030       STORE   R2,R14,48
    00000218  93B0FFFFFF9E   CALL    R11,WRR
    0000021E  C70E0018       LOAD    R0,R14,24
SOURCE LINE 67
    00000222  188F           NOTI    R8,15
    00000224  CE10FFE4       LADDR   R1,-28
    00000228  E7980024       LOADP   R9,R8,36
    0000022C  B798FFFFFFF4   STORE   R9,R8,-12
    00000232  8784FFF7       LOOP    R8,4,*-10
    00000236  8B000016       BR
    0000024C  112F           MOVEI   R2,15
    0000024E  113F           MOVEI   R3,15
    00000250  A70E0018       STORE   R0,R14,24
    00000254  A71E0020       STORE   R1,R14,32
    00000258  A72E0028       STORE   R2,R14,40
    0000025C  A73E0030       STORE   R3,R14,48
    00000260  93B0FFFFFF82   CALL    R11,WRS
    00000266  C70E0018       LOAD    R0,R14,24
    0000026A  C6101004       LOAD    R1,4100
    0000026E  112C           MOVEI   R2,12
    00000270  A70E0018       STORE   R0,R14,24
    00000274  A71E0020       STORE   R1,R14,32
    00000278  A72E0028       STORE   R2,R14,40
    0000027C  93B0FFFFFFFF   CALL    R11,WRI
    00000282  C70E0018       LOAD    R0,R14,24
    00000286  A70E0018       STORE   R0,R14,24
    0000028A  93B0FFFFFFFF   CALL    R11,WLN
SOURCE LINE 68
    00000290  8B00FE97       BR      W4
L5:
SOURCE LINE 70
    00000294  1100           MOVEI   R0,0
    00000296  A70E0018       STORE   R0,R14,24
```

```
     0000029A  93B0FFFFFFFF  CALL    R11,SYSEXIT
     000002A0  C7BF0000      LOAD    R11,R15,0
     000002A4  01EF          MOVE    R14,R15
     000002A6  C7FF0008      LOAD    R15,R15,8
     000002AA  57BB          RET     R11,R11
   NUMBER OF BYTES OF CODE GENERATED = 684
```

ASSEMBLER LISTING OF CALLED ROUTINES

```
$HEXOUT
;
;       function intxp(x : real) : integer ;
;
;       INTXP returns the unbiased exponent of the given
;       argument, i.e. returns (exponent - 127).
;
;       input : R14,24          -- x
;
;       output: R0              -- the answer
;
        GLOBAL  INTXP
INTXP:
        LOAD    R0,R14,24       ;R0 <- REAL NUMBER, I.E., LOAD x
        CSLI    R0,9            ;SHIFT EXPONENT INTO POSITION
        LADDR   R1,0FFH         ;LOAD MASK
        AND     R0,R1           ;MASK OUT MANTISSA AND SIGN BIT
        LADDR   R0,127          ;LOAD EXPONENT BIAS
        SUB     R0,R1           ;UNBIAS EXPONENT
        RET     R11,R11         ;RETURN TO CALLER

;
;       function setxp(x : real ; n : integer) : real ;
;
;       SETXP returns the real whose mantissa is that of x
;       and whose exponent is n.
;
;       input:  R14,24          -- x
;               R14,32          -- n, unbiased exponent
;
;       output: R0              -- the answer
;
        GLOBAL  SETXP
SETXP:
        LOAD    R0,R14,24       ;R0 <- REAL NUMBER, I.E., LOAD x
        LADDR   R1,0807FFFFFH,L ;LOAD MASK
        AND     R0,R1           ;CLEAR EXPONENT
        LOAD    R1,R14,32       ;LOAD EXPONENT, I.E. LOAD n
        LADDR   R2,127          ;LOAD EXPONENT BIAS
```

```
        ADD     R1,R2           ;ADD IN EXPONENT BIAS
        LADDR   R2,0FFH         ;LOAD MASK
        AND     R1,R2           ;ISOLATE 8-BIT EXPONENT
        CSLI    R1,15           ;SHIFT INTO POSITION
        CSLI    R1,8            ; ... IN TWO SHIFTS
        OR      R0,R1           ;'OR' IN NEW EXPONENT
        RET     R11,R11         ;RETURN TO CALLER
;
;       END OF SOURCE FILE
;
        END
```

Ridge Assembler Reference Manual

TABLE OF CONTENTS

PUBLICATION HISTORY

Manual Title: Ridge Assembler Reference Manual
First Edition: 9005 (OCT 83)
Second Edition: 9005-B (MAY 84)

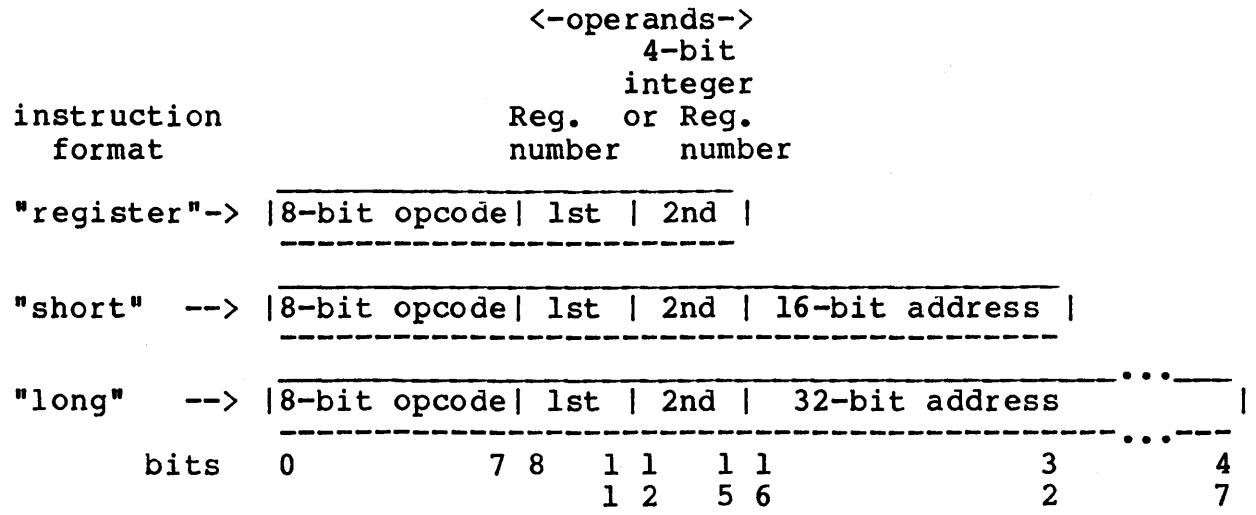
This manual is now a section of the ROS Programmer's Guide (9050).

Chapter 1
Ridge 32-Bit Processor

The Ridge 32 is a general purpose computer with 16 32-bit general registers and 16 32-bit special registers. The special registers are used only by some privileged instructions for process state information and instruction trap routines.

INSTRUCTION FORMATS

The instructions on the Ridge 32 fall into three instruction formats:



The register format (or "half-word" format) is used for instructions that operate on the contents of one or two registers and do not address memory. The short and long format instructions are used for memory-addressing instructions, such as storing and loading. The short format (or "word" format) is used for referencing addresses that can be specified in 16 bits, and the long format (or "word-and-a-half" format) is for referencing addresses that must be specified in 32 bits.

All three instruction types consist of:

- An 8-bit opcode.
- Two 4-bit operand fields.

The first operand always specifies a register. This register is often, but not always, used in calculating the result. The result of the register format instruction is always stored in the first operand register. The second operand field specifies a register or a 4-bit integer (in the range 0 to 15).

Any arithmetic or address operation can be performed on any register. (Registers are not specialized for counting or indexing.)

DATA TYPES

Data is manipulated in 16 32-bit general registers. Instructions exist for manipulating 3 32-bit data types (unsigned integers, two's-complement integers, and reals), and 2 64-bit types (64-bit unsigned integers and 64-bit double-precision real numbers).

The basic addressable unit is the 8-bit byte. Instructions exist for loading and storing 8-bit bytes, 16-bit halfwords, 32-bit words, and 64-bit double-words.

Integers

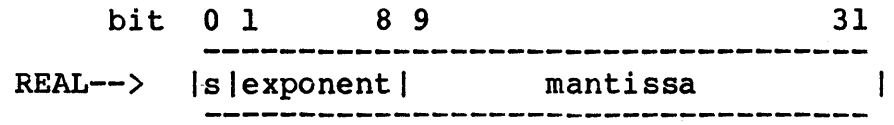
Integers are represented in two's-complement form and are in the range -2147,483,648 to 2,147,483,647, or unsigned in the range 0 to 4,294,967,295.

Real Numbers (Single Precision)

Real numbers (represented in floating-point form) consist of three parts: a sign, a power-of-two exponent, and a mantissa. The value of a real number is:

```
(-1)**s  x  2**(exponent-127)  x  1.mantissa
```

For positive numbers, the sign bit (bit 0) is 0. For negative numbers, the sign bit is 1. The exponent of a real number is 8 bits long, and is biased by +127. The eight bits of the exponent give a range of 0 through 255. Subtracting the bias yields an exponent range of -127 through +128. The mantissa has an implicit leading one, and is 23 bits long. Zero is represented by all zeros.



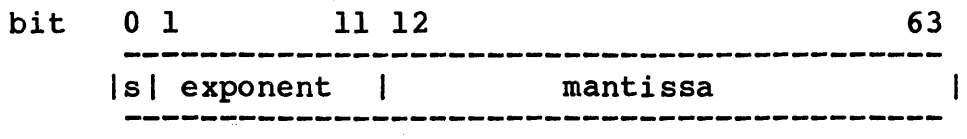

```
example: "1"     =     0 01111111 00000000000000000000000 = 3F80 0000
example: "-10"   =     1 10000010 01000000000000000000000 = C120 0000
```

Real Numbers (Double Precision)

Double reals are similar to reals, except that the mantissa is 52 bits, and the exponent is 11 bits. The exponent is biased by

+1023. The eleven exponent bits give a range of 0 through 2047.

Subtracting the bias yields an exponent range of -1023 through +1024.



"1" = 0 01111111111 00000000000000...000000000000 = 3FF0 0000 0000 0000
"-10"= 1 10000000010 01000000000000...000000000000 = C024 0000 0000 0000

SYNTAX CONVENTIONS

In syntax statements, the 16 general registers are referred to as Rx or Ry. Special registers are called SRx or SRy. In an assembly program, general register 4 and special register 15 would be called R4 and SR15, respectively.

Double words occupy register pairs. A register pair, RPx, consists or Rx and R(x+1) mod 16. Rx holds the most significant bits of RPx, and R(x+1) holds the least significant bits. Example: RP5 refers to R5 and R6, with the most significant bits of the pair in R5, and the least significant bits in R6. RP15 refers to R15 and R0.

Bit 0 is the most significant bit of a data type. For 32-bit data types, bit 31 is the least significant bit. For 64-bit data types, bit 63 is the least significant bit.

Specific bits of a register or word are enclosed in brackets. For example, bit 3 of a register is referred to as Rx[3], or Ry[3]. The symbol ".." denotes a range of bits. For example, consecutive bits 6 through 9 of a register are referred to as Rx[6..9], or Ry[6..9]. Bits 28 through 34 of a register pair are referred to as RPx[28..34] or RPy[28..34].

SEPARATE CODE AND DATA

Code and data reside in separate 4-gigabyte address segments, and are manipulated with separate types of instructions. The 4-gigabyte code address space is called the code segment and the 4-gigabyte data address space is called the data segment.

COMPARE AND BRANCH INSTRUCTIONS

Comparison and conditional or unconditional branching are performed within the single branch instruction BR.

BRANCH PREDICTION

The instruction pipeline of the Ridge 32 is designed so that the next instruction to execute is fetched from memory even while the current instruction is being executed. If the prediction bit of the branch instruction is set, the processor will fetch the instruction specified by the destination bits while the test is being performed. If the prediction bit is not set, the processor fetches the next sequential instruction while the test is being performed. If the programmer sets the prediction bit according to the most likely and frequent result of the branch test, a performance increase will be realized.

KERNEL MODE AND PRIVILEGED USER MODE

When operating in kernel mode, the processor uses real addresses only, rather than virtual addressing, and interrupts are disabled. The kernel mode code is entered at one of 256 entry points by means of the KCALL instruction, or by means of an exception or trap.

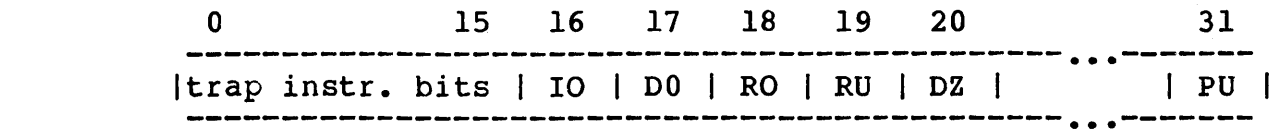
Privileged user mode allows the user to execute certain system maintenance instructions without being in kernel mode (without losing the benefit of virtual addressing and interrupts). Privileged user mode is activated by setting bit 31 of the traps word (see "Traps Word" under EXCEPTIONS and TRAPS).

EXCEPTIONS AND TRAPS

Exceptions

An exception is the abnormal execution of an instruction, like an interrupt, a page fault, or some other unusual condition. Possible exceptions for each instruction are explained in this manual, including how the registers are affected and what action is taken when they occur. Some exceptions can be enabled and disabled under control of the "traps word"--the traps word determines which type of exceptions result in suspension of the user program.

Traps Word



The trap instruction bits are the ones checked by the TRAP instruction.

- IO enables integer overflow trap
- DO enables integer divide-by-zero trap
- RO enables real overflow trap
- RU enables real underflow trap
- DZ enables real divide-by-zero trap
- PU enables privileged user mode

Without privileged user mode, a kernel violation trap occurs upon executing the certain instructions that require that mode.

Chapter 2
Program Structure

The Ridge assembler converts symbolic representations of Ridge machine instructions into a relocatable object file that the Ridge linker program (ld) accepts. This chapter describes the system architecture that must be understood before writing a program in Ridge assembly language.

SECTIONS

The Ridge 32 has two addressable memory spaces for each running program: the code segment and the data segment. The data segment is not created until the program is actually executed. An executable program file contains the code segment portion and information about creating the data segment at run time.

An assembly program is divided into any number of sections: Code Sections and Data Sections, headed by the CSECT and DSECT, pseudo-instructions, respectively.

Code Sections

Code sections contain program code and cannot be written into. Code sections should be named by the CSECT pseudo-instruction. If not specially named by this CSECT pseudo-instruction, the mnemonics are assumed to fall in a code section named "code".

Disjointed code sections of the same name are assembled into one contiguous section of the code segment.

Data Sections

Data sections contain program data initializers, such as the SPACE, WORD, BYTE, DOUBLE, and HALFWORD pseudo-instructions. Data sections must be headed, and should be given a name, by the DSECT pseudo-instruction. If headed by DSECT but not specifically named, a data section assumes the name "data". If not headed by DSECT, the data initializers are assumed to fall into a code section named "code," as described above.

If several unnamed data sections appear in a source file, they are assembled contiguously into one section named "data". Disjointed data sections of the same name are assembled in one contiguous data segment of the given name.

PROGRAM COUNTER (PC)

Each of the various Code and Data sections has a location counter associated with it. Within a section, the symbol "*" evaluates to the current value of its counter. The counter for each section starts at zero when the first CSECT or DSECT pseudo-instruction defining that section is seen.

Chapter 3
Assembly Language Syntax

The instruction mnemonics in an AS program must be structured in the following format:

label field (starts in column 1)	instruction or pseudo-instruction field	operand field	comment field...
Label.	MOVE	R6,R9	;Load R6 with R9

(blank(s) or tab(s) may separate fields)

LABELS

A label consists of any sequence of alphabetic characters, numbers, "_", "$", and ".", which does not begin with a number. When labels are compared, AS considers every character of the name. Lower case and upper case characters are treated distinctly. A colon (:) is required on the label if no instruction appears on the line.

The following pseudo-instructions do not allow labels: CSECT, DSECT, ORIGIN, COMMON, EXTERNAL, EXTERND, GLOBAL, ALIGN, and LCOMM.

The following pseudo-instructions require labels: CODE, DATA, and EQU.

Label Types

Labels provide reference points to code and data in an assembly program. Labels start in column 1 of an assembly program. A label is one of three types: code, data, or absolute.

- A code label is one which appears in a code section (headed by the CSECT pseudo-instruction). Its value is the section's PC at the time it is declared.
- A data label is one which appears in a data section (headed by the DSECT pseudo-instruction). Its value is the section's PC at the time it is declared.
- An absolute label assumes the value assigned to it by the EQU pseudo-instruction.

A reference to a code or data label evaluates to the starting

address of the section in which it appears, plus its own value as determined by the PC. A reference to an absolute label evaluates exactly to the address that was assigned to it.

COMMENTS

Any characters after a ";" on a line are ignored by AS. Use comments liberally to document programs. Blank lines are allowed and ignored.

CONSTANTS

AS recognizes base-10 (decimal) or base-16 (hexadecimal) integers, and single- and double-precision floating-point constants. All constants can be preceded by an optional sign ("+" or "-").

A decimal integer consists of the digits 0 to 9, and must be in the range -2,147,483,648 to +2,147,483,647. Examples:

```
123     +123     -123     +2147483647
```

A hexadecimal integer consists of 0 to 9, and A to F (upper or lowercase), begins with a digit, and ends with the letter "H" or "h". Examples:

```
80000000h  7FFFFh  80000000h  6F2Bh 0C3h
```

A single- or double-precision floating-point constant starts with one or more digits, must have a decimal point, optional digits following the decimal point, and an optional exponent following an "E" or "e". The exponent may be signed "+" or "-". The optional exponent of a double-precision number is flagged with "D" or "d". Examples:

```
123.45E+4   1.   1.3   1.3E-4   1.3E+4   1.3E4   1.3D-4
```

EXPRESSIONS

An expression consists of one or more labels, constants, or location counter symbols ("*") separated by the arithmetic operators "+", "-", "*", or "/". A sub-expression (a sub-part of the expression that is itself an expression) may be enclosed by parentheses.

Expressions are arithmetically evaluated. A label in an expression assumes the value of its location in the section.

The customary order of operator precedence applies to the evaluation of expressions--multiplication and division first, then addition and subtraction. Parenthesized sub-expressions are

evaluated first.

Expressions have the same three types as data: code, data, and absolute. The type is determined by the first non-absolute label to be evaluated in an expression.

- Code expressions contain only code segment labels or absolute labels. But if a code label is subtracted from another in the expression, the expression becomes absolute.
- Data expressions contain only data segment labels or absolute labels. But if a data label is subtracted from another in the expression, the expression becomes absolute.
- Absolute expressions contain only absolute labels, or no labels at all, or are the result of subtracting labels as described above.

An expression preceded by "#" is always considered absolute, regardless of its components.

Code and data labels cannot be mixed in one expression. The type of all the non-absolute labels in an expression must match. A general expression allows one external label or one forward label (one which is defined later in the program).

Chapter 4
Pseudo-Instructions

Pseudo-instructions are "commands" which help the assembler organize code and data into a form that is recognized by the Ridge linker.

LIST OF PSEUDO-INSTRUCTIONS

- ALIGN - align location counter
- BLOCK - assemble block of bytes
- BYTE - assemble strings or expressions
- CODE, DATA, EQU - assign type and value to a label
- COMMON - declare globally known block data
- CSECT, DSECT - assembly section headers
- EXTERNAL, EXTERND - declare external labels
- GLOBAL - make labels global
- LCOMM - declare locally known block data
- ORIGIN - set location counter
- SPACE - reserve bytes
- WORD, HALFWORD, DOUBLE - assemble expression into entity

ALIGN
Align Location Counter

SYNTAX

ALIGN <alignval>

DESCRIPTION

ALIGN assembles dummy bytes so that the location counter ends up on an integral multiple of the <alignval>. ALIGN 8, for example, outputs dummy bytes so that the location counter ends up on an integral multiple of 8 bytes. No pad bytes are output if the location counter is already aligned.

<alignval> contains no forward or external label names, and evaluates to a constant in the range 2 to 4096.

No label field is allowed on the ALIGN pseudo-instruction.

BLOCK
Assemble Block of Bytes

SYNTAX

BLOCK <count>, <byteval>

DESCRIPTION

BLOCK assembles <count> bytes of the character <byteval> into the current section.

<count> contains no forward or external label names, and evaluates to any integer constant.

<byteval> contains no forward or external label names, and evaluates to a constant in the range 0 to 255.

BYTE
Assemble Strings or Expressions

SYNTAX

```
BYTE { <quoted string> | <byteval> }
       [, { <quoted string> | <byteval> } ... ]
```

DESCRIPTION

BYTE assembles the specified list of quoted strings and/or expressions evaluating to a single byte into the current section. Strings are delimited by single quotation marks (') and are assembled "as is." Certain special characters may be included in a quoted string by entering a backslash followed by a special character:

\b	backspace	(08H)	
\f	formfeed	(0cH)	
\n	newline	(0aH)	same as linefeed
\r	return	(0dH)	i.e., carriage return
\t	horiz tab	(09H)	control-I
\'	single quote		
\\	backslash itself		
\0	NULL	(00H)	

CODE, DATA, EQU
Assign Label Type and Value

SYNTAX

```
<label> CODE <expression>
<label> DATA <expression>
<label> EQU  <expression>
```

DESCRIPTION

CODE, DATA, and EQU assign a value and a type to a label. There must be a label field with these pseudo-instructions.

CODE defines a code-section-relative label; the defining expression must have code-section-relative or absolute type.

DATA defines a data-section-relative label; the defining expression must have data-section-relative or absolute type.

EQU defines a label which assumes the type of the defining expression; the defining expression may have any of the three types.

<expression> contains no forward or externally defined labels. If the CODE pseudo-instruction is used with an expression that evaluates to an absolute type, then the label is considered to have been declared relative to the default code section "code". If the DATA pseudo-instruction is used with an expression that evaluates to an absolute type, then the label is considered to have been declared relative to the default data section "data".

COMMON
Declare Fixed Size External Data Label

SYNTAX

COMMON <name>, <size>

DESCRIPTION

COMMON defines globally known label to a data block of fixed size. It is equivalent to declaring the label as an external, except that the linker will allocate an area in the data segment large enough to hold the specified number of bytes. Other modules may also declare common blocks with the same label; the linker guarantees that enough space is allocated to satisfy the largest declaration.

<name> is label.

<size> indicates the number of bytes to reserve for storage. If multiple modules declare the same label common with differing sizes, the largest declared size is actually used.

No label field is allowed on the COMMON pseudo-instruction.

CSECT, DSECT
Assembly Section Headers

SYNTAX

CSECT <section-name>
DSECT <section-name>

DESCRIPTION

CSECT and DSECT declare to the assembler the section into which it should assemble the instructions and data which follow. CSECT heads a code segment section and DSECT heads a data segment section.

<section-name> is a string of characters, delimited by blanks, that indicates the name of the section. If the name has already been encountered, assembly continues at the location held in the location counter for that existing section. If the name has not been seen, assembly starts in a newly-created section with a location counter initialized to zero. CSECT and DSECT do not allow a label field.

The assembler knows about two pre-defined sections: a code-segment-relative section called "code", and a data-segment-relative section called "data". Labels defined with the CODE pseudo-instruction whose defining expression is absolute are defined relative to section "code". Labels defined with the DATA pseudo-instruction whose defining expression is absolute are defined relative to section "data".

If no section declaration is given, AS assumes a code section. If the code section is not given a name, AS assumes the name "code".

EXTERNAL, EXTERND
Declare External Labels

SYNTAX

```
EXTERNAL <label1> [,<label2> ... ]
EXTERND  <label1> [,<label2> ... ]
```

DESCRIPTION

EXTERNAL and EXTERND declare one or more labels to be external to the current assembly module. EXTERNAL declares a code segment label; EXTERND declares a data segment label. The operand field contains a list of <label>s, separated by commas. No label is allowed with EXTERNAL or EXTERND.

GLOBAL
Make Labels Global

SYNTAX

GLOBAL <label1> [,<label2> ...]

DESCRIPTION

GLOBAL declares the labels listed in the operand field as globally known. These labels will satisfy externals declared in other modules. It is permissible to declare a global prior to actually defining it.

No label field is allowed on a GLOBAL pseudo-instruction.

LCOMM
Declare Local Data Block

SYNTAX

LCOMM <name>, <size>

DESCRIPTION

LCOMM reserves <size> bytes in the current data section. <name> is not declared external, and is unknown outside of the module.

No label field is allowed on an LCOMM pseudo-instruction.

ORIGIN
Set Location Counter

SYNTAX

ORIGIN <loc ctr>

DESCRIPTION

ORIGIN sets the location counter of the current section equal to the specified value.

<loc ctr> is an expression containing no forward or external labels, and evaluates to an integer.

SPACE
Reserve Bytes

SYNTAX

SPACE <count>

DESCRIPTION

SPACE reserves <count> bytes of space in the current section. Data sections that are defined by only SPACE and ALIGN pseudo-instructions are considered part of the uninitialized data area. The linker concatenates all uninitialized data sections together and merely notes their starting and ending point in the code file header. The section formed by all otherwise-uninitialized common declarations is also considered a part of the uninitialized data area. This area is occasionally referenced to by the name bss.

A section that has actual information assembled into it by instructions or a pseudo-instruction treats SPACE differently. A SPACE pseudo-instruction in such a section actually causes <count> bytes of zero to be assembled into the section.

WORD, HALFWORD, DOUBLE

Assemble Expression into Entity

SYNTAX

```
WORD      <expression>
HALFWORD  <expression>
DOUBLE    <double constant>
```

DESCRIPTION

WORD and HALFWORD evaluate <expression> and assemble the result into a 32-bit word or 16-bit halfword, respectively.

DOUBLE converts <double constant> into a 64-bit double-precision floating-point number and assembles it into the current section.

<expression> may evaluate to an absolute constant, a value relative to a code or data segment, or a 32-bit floating-point value. Code-segment-relative and data-segment-relative values are marked for later relocation by the linker. Absolute values are assembled "as is." All expressions are evaluated as 32-bit quantities; if the expression of a HALFWORD value evaluates to a number greater than 32767 or less than -32768, an error is printed.

Chapter 5
Instructions

Documentation on the Ridge assembly instructions assumes the following format:

NAME

one-line explanation

SYNTAX

instruction syntax in an assembly program

FUNCTION

a description of how the symbols from "SYNTAX" might be coded in a higher level language

DESCRIPTION

an explanation of the instruction

EXCEPTIONS

the special conditions under which a trap can be generated by this instruction

INSTRUCTION FORMAT

a bit map showing the position of the opcode, operands, and address parts of the assembled instruction

ADD
Add

SYNTAX

```
ADD     Rx, Ry
ADD     Rx, <smallval>
```

FUNCTION

```
    Rx := Rx + Ry
or
    Rx := Rx + smallval
```

DESCRIPTION

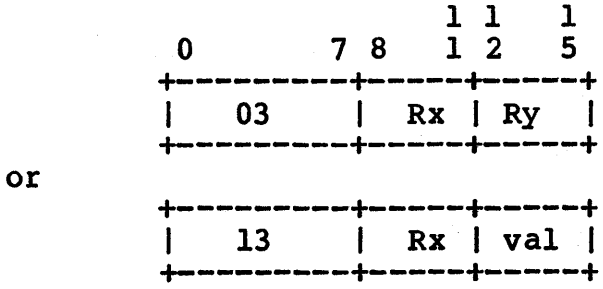
Add the contents of Rx and Ry, or the contents of Rx and <smallval>, and put the result in Rx.

<smallval> is an expression with a value in the range from 0 through 15. The expression cannot contain forward or external labels.

EXCEPTIONS

If integer overflow occurs, the least significant 32 bits of the result are placed in Rx, the most significant bits are discarded, and the integer overflow trap is taken if enabled.

INSTRUCTION FORMAT



AND
Logical AND

SYNTAX

```
AND     Rx, Ry
AND     Rx, <smallval>
```

FUNCTION

```
Rx := Rx AND Ry
```

or

```
Rx := Rx AND smallval
```

DESCRIPTION

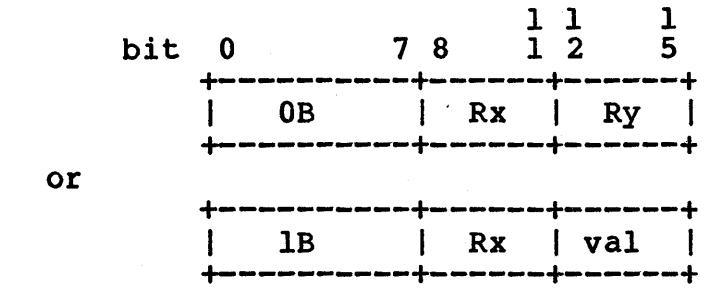
Perform logical AND with the contents of Rx and Ry, or Rx and <smallval>, and put the result in Rx.

<smallval> is an expression with a value in the range from 0 through 15. The expression cannot contain forward or external labels.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



ASL
Arithmetic Shift Left

SYNTAX

```
ASL     Rx, Ry
ASL     Rx, <smallval>
```

FUNCTION

```
hold := Rx[0];
shift contents of Rx to left by n bits;
Rx[0] := hold;
Rx[31-(n-1)..31] := 0

where n = <smallval> or contents of Ry mod 32
```

DESCRIPTION

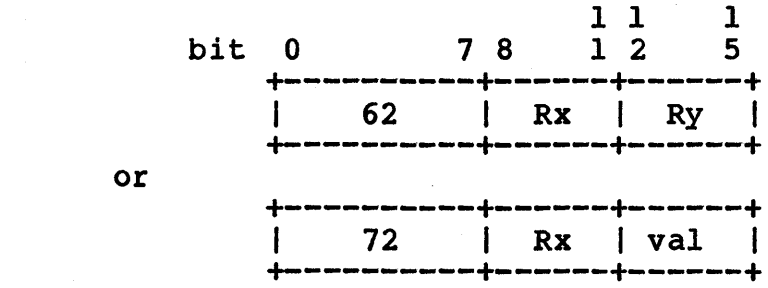
ASL shifts the bits in Rx to the left by the number of bits specified in Ry or by <smallval>. ASL inserts 0 at the right, but preserves the sign bit.

<smallval> is an expression with a value in the range from 0 through 15. The expression cannot contain forward or external labels.

EXCEPTIONS

If the initial Rx[0] does not equal the final Rx[0] (the sign bit), an integer overflow trap is taken if enabled.

INSTRUCTION FORMAT



ASR
Arithmetic Shift Right

SYNTAX

```
ASR     Rx, Ry
ASR     Rx, <smallval>
```

FUNCTION

shift contents of Rx to right by n bits;
Rx[1..n] := Rx[0]

where n = <smallval> or contents of Ry mod 32

DESCRIPTION

ASR shifts the bits in Rx to the right by the number of bits specified in Ry or by <smallval>.

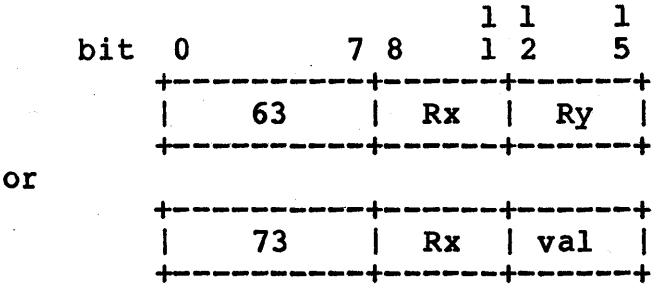
ASR shifts all bits to the right, filling the left with duplicates of the sign bit.

<smallval> is an expression with a value in the range from 0 through 15. The expression cannot contain forward or external labels.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



BR
Branch

SYNTAX

```
BR      <label> [,L]
BR      Rx <relop> Ry, <label> [!] [,L]
BR      Rx <relop> <smallval>, <label> [!] [,L]
```

FUNCTION

```
        PC := label;
or
        IF Rx <relop> Ry is true THEN PC := label
or
        IF Rx <relop> smallval is true THEN PC := label
```

DESCRIPTION

BR may conditionally or unconditionally transfer execution to the code identified by the <label>.

If only a label appears in the operand field, BR branches unconditionally to the code at that label.

If a relational operator (relop) appears with two register names, or one register and a small value, BR makes a comparison and branches to the label if true.

<smallval> is an expression with a value in the range from 0 through 15. The expression cannot contain forward or external labels.

<label> can contain label names, and must evaluate to a code-relative value. <label> may be short (16 bits) or long (32 bits). A long label must be flagged with [,L].

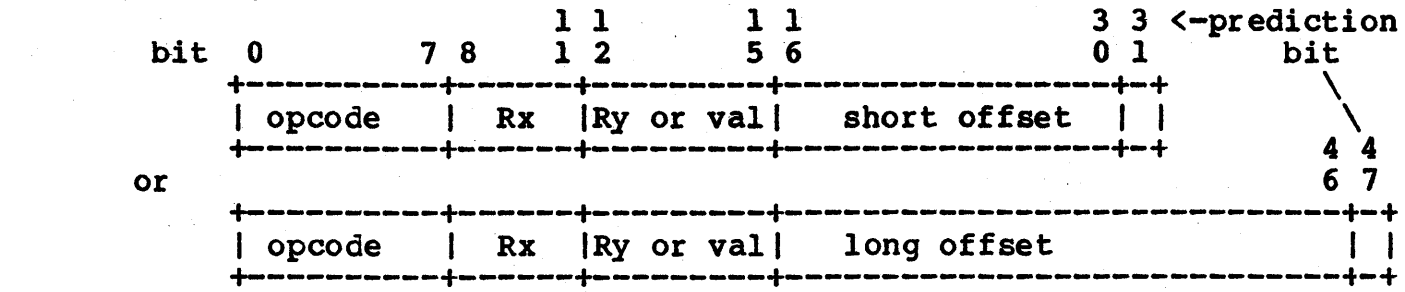
<relop> is any of the relational operators <, <=, =, >, >=, or <>.

[!] sets the branch prediction bit in the instruction.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



BR instruction	OPCODE short	long
BR offset	8B	9B
BR Rx > Ry, offset	80	90
On "Rx<Ry" , "Ry>Rx" is assembled		
BR Rx = Ry, offset	82	92
BR Rx <= Ry, offset	88	98
On "Rx>=Ry", "Ry<=Rx" is assembled		
BR Rx <> Ry, offset	8A	9A
BR Rx > val, offset	84	94
BR Rx < val, offset	85	95
BR Rx = val, offset	86	96
BR Rx <= val, offset	8C	9C
BR Rx >= val, offset	8D	9D
BR Rx <> val, offset	8E	9E

Use of the least significant bit as the prediction bit does not affect the interpretation of the offset bits because the offset is always a multiple of 2. The least significant bit is not considered in calculating the offset.

The offset is calculated to be the offset of the target from the current PC, not an absolute address of the target. A short offset is sign-extended.

CALL
Call Subroutine

SYNTAX

CALL Rx, <label> [,L]

FUNCTION

Rx := PC + instruction size;
PC := label

DESCRIPTION

CALL transfers control to the location at a specific offset from the current PC, saving the address of the next instruction. CALL puts the address of the next instruction in Rx and branches to the code at the label. Rx can be used as a return point from the subroutine.

<label> must evaluate to a code-relative value.

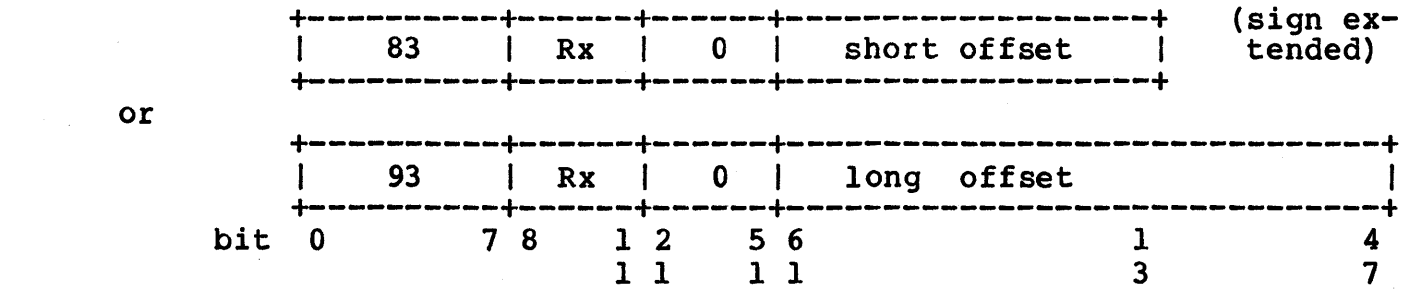
[,L] indicates that <label> evaluates to a 32-bit offset. If not specified, a short offset is assembled.

The offset is calculated to be the offset of the target from the current PC, not an absolute address of the target.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



CALLR
Call Subroutine Register

SYNTAX

```
CALLR    Rx, Ry
```

FUNCTION

```
Rx := PC + 2;
PC := PC + Ry
```

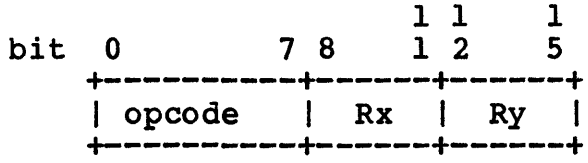
DESCRIPTION

CALLR puts the address of the next instruction in Rx and branches to the code at PC + Ry. Rx can be used as a return point from the subroutine.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



CBIT
Clear Bit

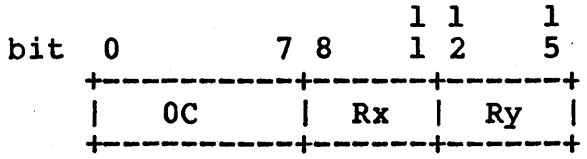
SYNTAX
```
     CBIT     Rx, Ry
```

FUNCTION
In RPx, clear the bit specified in Ry.

DESCRIPTION
CBIT clears the bit in RPx that is specified in Ry mod 64.

EXCEPTIONS
There are no exceptions.

INSTRUCTION FORMAT



CSL
Circular Shift Left

SYNTAX

```
CSL     Rx, Ry
CSL     Rx, <smallval>
```

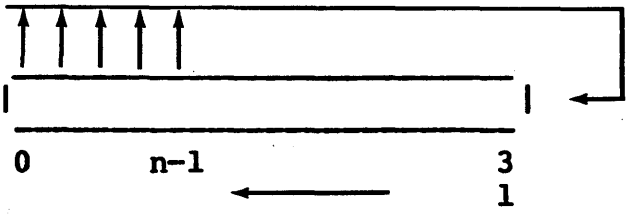
FUNCTION

```
hold := Rx[0..n-1];                    {for n>0}
shift contents of Rx to left by n bits;
Rx[31-(n-1)..31] := hold;              {for n>0}
```

where n = <smallval> or contents of Ry mod 32

DESCRIPTION

The first "n" bits of the word are carried around to bit 31, then all bits are shifted left by "n" bit positions, where "n" = <smallval> or the contents of Ry mod 32. No bits are lost in the process.

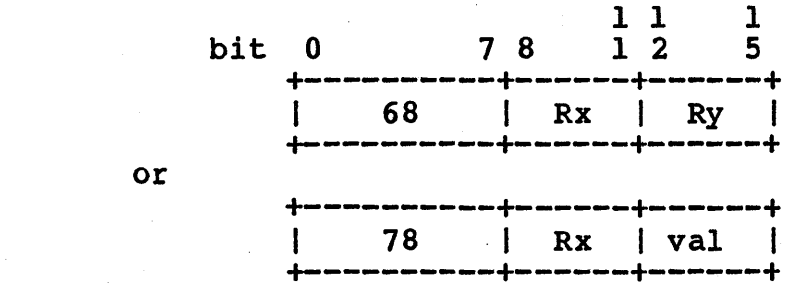


<smallval> is an expression with a value in the range from 0 through 15. The expression cannot contain forward or external labels.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



DCOMP
Double Register Compare

SYNTAX

```
DCOMP    Rx, Ry
```

FUNCTION

```
IF RPx < RPy THEN Rx := -1;
IF RPx = RPy THEN Rx := 0;
IF RPx > RPy THEN Rx := 1
```

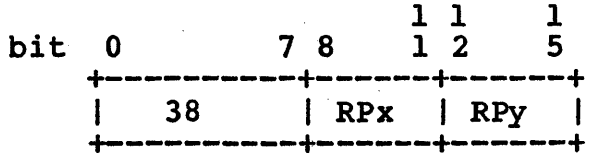
DESCRIPTION

DCOMP compares the 64-bit integers in RPx and RPy. If RPx is less than RPy, Rx is set to -1. If RPx equals RPy, Rx is set to 0. If RPx is greater than RPy, Rx is set to 1.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



DFIXR
Double Real Round to Integer

SYNTAX

```
DFIXR    Rx, Ry
```

FUNCTION

```
Rx := int (RPy)
```

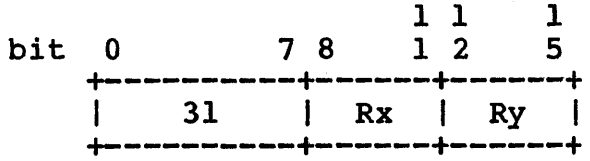
DESCRIPTION

DFIXR converts the double real number in RPy into an integer in Rx. Fractions of .5 or larger are rounded up to the next higher absolute value integer.

EXCEPTIONS

If integer overflow occurs, Rx is unmodified and an integer overflow trap is taken if enabled.

INSTRUCTION FORMAT



DFIXT
Double Real Truncate to Integer

SYNTAX

```
DFIXT    Rx, Ry
```

FUNCTION

```
Rx := trunc (RPy)
```

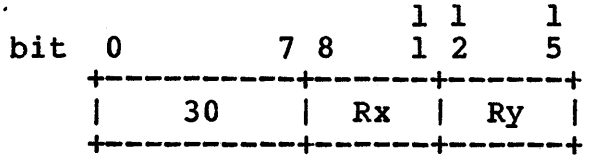
DESCRIPTION

DFIXT converts the double precision real number in RPy into an integer in Rx. All bits to the right of the decimal point are lost.

EXCEPTIONS

If integer overflow occurs, Rx is unmodified and an integer overflow trap is taken if enabled.

INSTRUCTION FORMAT



DFLOAT
Integer to Double Real

SYNTAX

```
DFLOAT   Rx, Ry
```

FUNCTION

```
RPx := float (Ry)
```

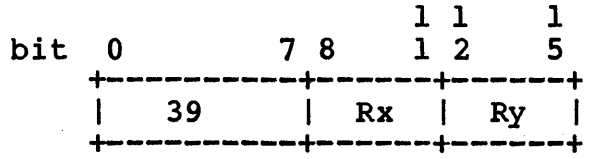
DESCRIPTION

DFLOAT converts the integer in Ry into a double precision real number in RPx.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



DIV
Integer Divide

SYNTAX

```
DIV     Rx, Ry
```

FUNCTION

```
Rx := Rx/Ry
```

DESCRIPTION

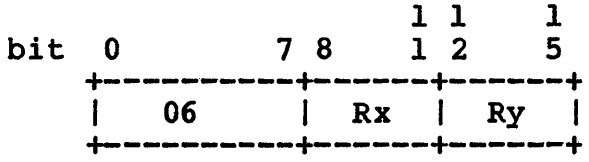
DIV divides the two's-complement integer in Rx by the two's-complement integer in Ry and puts the result in Rx.

EXCEPTIONS

Integer overflow can occur when the largest negative integer is divided by -1. In this case, Rx is not modified, and an integer overflow trap is taken if enabled.

An attempt to divide by 0 leaves Rx unmodified, and a divide-by-zero trap is taken if enabled.

INSTRUCTION FORMAT



DLSL
Double Logical Shift Left

SYNTAX

```
DLSL    Rx, Ry
DLSL    Rx, <smallval>
```

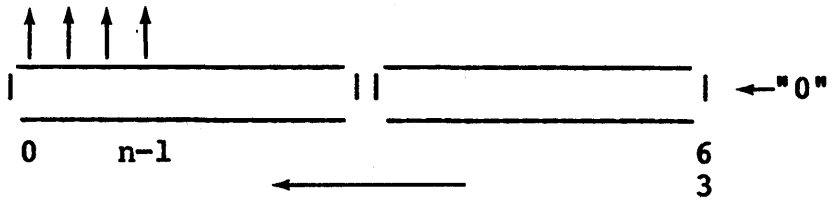
FUNCTION

shift contents of RPx to left by n bits;
RPx[63-(n-1)..63] := 0

where n = <smallval> or the contents of Ry mod 64

DESCRIPTION

DLSL shifts all bits to the left, inserting "0" on the right of R(x+1), truncating bits on the left of Rx, by the number of bits specified in Ry or <smallval>.

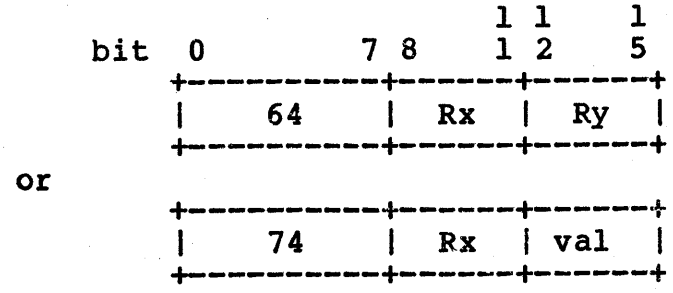


<smallval> is an expression with a value in the range from 0 through 15. The expression cannot contain forward or external labels.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



DLSR
Double Logical Shift Right

SYNTAX

```
     DLSR     Rx, Ry
     DLSR     Rx, <smallval>
```

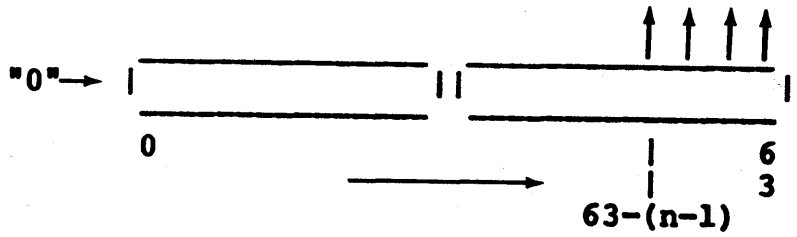
FUNCTION

shift contents of RPx to right by n bits;
RPx[0..n-1] := 0

where n = <smallval> or the contents of Ry mod 64

DESCRIPTION

DLSR shifts all bits to the right, inserting "0" on the left of Rx, and truncating bits on the right of R(x+1), by the number of positions contained in Ry or <smallval>.

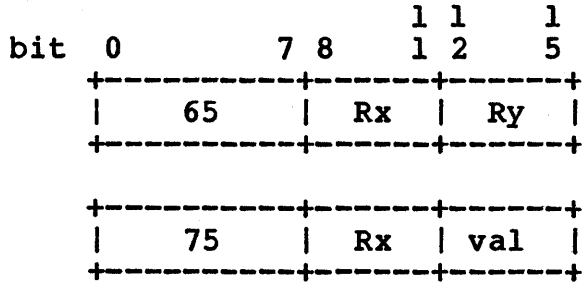


<smallval> is an expression with a value in the range from 0 through 15. The expression cannot contain forward or external labels.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



DRADD
Double Real Add

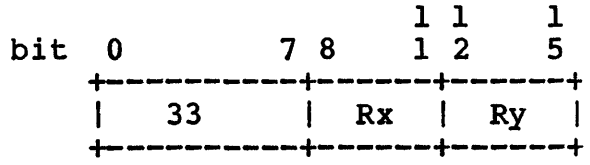
SYNTAX
```
DRADD    Rx, Ry
```

FUNCTION
```
RPx := RPx + RPy
```

DESCRIPTION
DRADD adds the double-real numbers in RPx and RPy and puts the sum in RPx.

EXCEPTIONS
On overflow, RPx is set to the largest real number with the appropriate sign bit. On real underflow, RPx is set to 0. In either case, an under- or overflow trap is taken if enabled.

INSTRUCTION FORMAT



DRCOMP
Double Register Compare

SYNTAX

```
DRCOMP   Rx, Ry
```

FUNCTION

```
IF RPx < RPy then Rx := -1;
IF RPx = RPy then Rx := 0;
IF RPx > RPy then Rx := 1
```

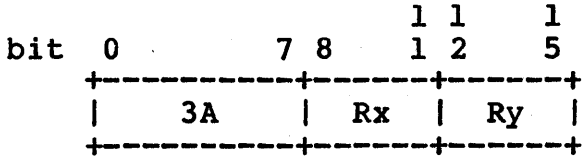
DESCRIPTION

DRCOMP compares the double-precision real numbers in RPx and RPy, and sets Rx to -1, 0, or 1 depending on whether RPx is less than, equal to, or greater than RPy, respectively.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



DRDIV
Double Precision Real Divide

SYNTAX

```
DRDIV    Rx, Ry
```

FUNCTION

```
RPx := RPx/RPy
```

DESCRIPTION

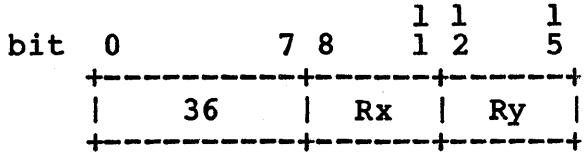
DRDIV divides the double-precision real in RPx by the double-precision real in RPy and puts the result in the RPx.

EXCEPTIONS

On overflow, RPx is set to the largest real number with the appropriate sign bit. On real underflow, RPx is set to 0. If RPy is zero, RPx is set to the largest real number with the appropriate sign bit. In any case, a real underflow, overflow, or divide by zero trap is taken if enabled.

An attempt to divide by zero leaves RPx unmodified, and a real divide by zero trap is taken if enabled.

INSTRUCTION FORMAT



DRMPY
Double Precision Real Multiply

SYNTAX

```
DRMPY    Rx, Ry
```

FUNCTION

```
RPx := RPx * RPy
```

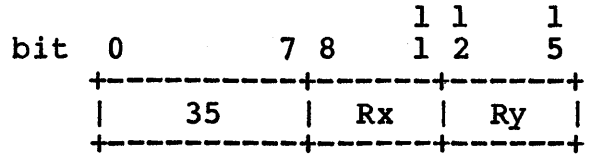
DESCRIPTION

DRMPY multiplies the double-precision real number in RPx by the double-precision real in RPy and puts the product in the RPx.

EXCEPTIONS

On overflow, RPx is set to the highest double real number with the appropriate sign bit. On underflow, RPx is set to 0. In either case, a real under- or overflow trap is taken if enabled.

INSTRUCTION FORMAT



DRNEG
Double Precision Real Negate

SYNTAX

```
DRNEG    Rx, Ry
```

FUNCTION

RPx := - RPy

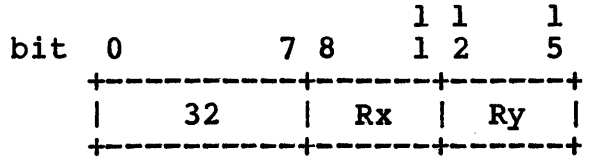
DESCRIPTION

DRNEG negates the double-precision real number in RPy and puts the result in the RPx.

EXCEPTIONS

On overflow, RPx is set to the largest real number with the appropriate sign bit. On real underflow, RPx is set to 0. In either case, an under- or overflow trap is taken if enabled.

INSTRUCTION FORMAT



DRSUB
Double Precision Real Subtract

SYNTAX

```
DRSUB    Rx, Ry
```

FUNCTION

RPx := RPx - RPy

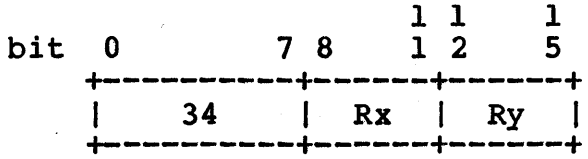
DESCRIPTION

DRSUB subtracts the double-precision real number in RPx from the double-precision real in RPy and puts the difference in RPx.

EXCEPTIONS

On overflow, RPx is set to the largest real number with the appropriate sign bit. On real underflow, RPx is set to 0. In either case, an under- or overflow trap is taken if enabled.

INSTRUCTION FORMAT



EADD
Extended Integer Add

SYNTAX

```
EADD     Rx, Ry
```

FUNCTION

```
Rx := Rx + Ry + R0[31];
R0[0..29] := 0;
R0[30] := overflow;
R0[31] := carry
```

DESCRIPTION

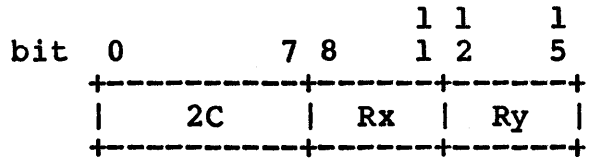
EADD adds the two's-complement integers in Rx and Ry, and at the same time adds the carry-in from R0 bit 31, and puts the low 32-bits of the sum in Rx. The carryout (most significant) bit is put in R0 bit 31. On overflow, R0 bit 30 is set. The upper 30 bits of R0 are set to zero.

EADD can be used to implement multiple-word arithmetic. Normally, this is done by setting R0[31] to zero, EADDing the least significant words, then the next least significant words, and so on, until the most significant words have been EADDed. Overflow can then be checked after the last EADD.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



EDIV
Extended Divide

SYNTAX

```
EDIV     Rx, Ry
```

FUNCTION

```
Rx := RPx/Ry ;
Ry := the remainder
```

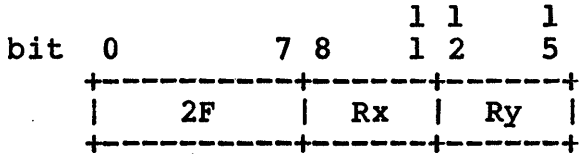
DESCRIPTION

EDIV divides the unsigned 64-bit contents of RPx by the unsigned 32-bit contents of Ry, and puts the integer quotient in Rx and the remainder in Ry.

EXCEPTIONS

If integer overflow occurs (the result is greater than 32 bits), RPx is not modified and an integer overflow trap is taken if enabled. If division by 0 is attempted, Rx is not modified and a divide-by-zero trap is taken if enabled.

INSTRUCTION FORMAT



EMPY
Extended Multiply

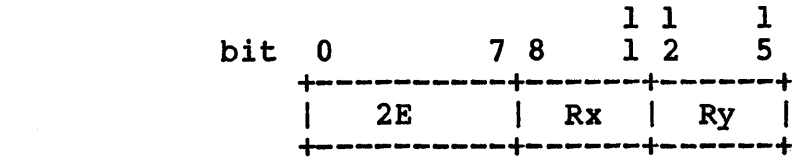
SYNTAX
```
     EMPY      Rx, Ry
```

FUNCTION
```
     RPx := Rx * Ry
```

DESCRIPTION
EMPY multiplies two unsigned 32-bit numbers and puts the unsigned 64-bit product in RPx.

EXCEPTIONS
There are no exceptions.

INSTRUCTION FORMAT



ESUB
Extended Subtract

SYNTAX

```
ESUB     Rx, Ry
```

FUNCTION

```
Rx := Rx - Ry + R0[31] ;     (one's complement subtraction)
R0[0..29] := 0;
R0[30] := overflow;
R0[31] := carry;
```

DESCRIPTION

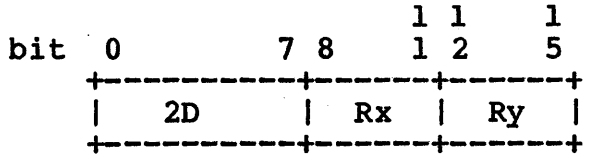
ESUB one's-complement-subtracts the two's-complement integer in Ry from the two's-complement integer in Rx, and at the same time adds the carry-in from R0[31], and puts the 32-bit two's-complement difference in Rx. The carryout (most significant) bit is put in R0[31]. Overflow is indicated in R0[30]. The upper 30 bits of R0 are set to zero.

ESUB can be used to implement multiple-word arithmetic. Normally, this is done by setting R0[31] to one, ESUBing the least significant words, then the next least significant words, and so on, until the most significant words have been ESUBed. Overflow can then be checked after the last ESUB.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



FIXR
Round to Integer

SYNTAX

```
FIXR     Rx, Ry
```

FUNCTION

```
Rx := round(Ry)
```

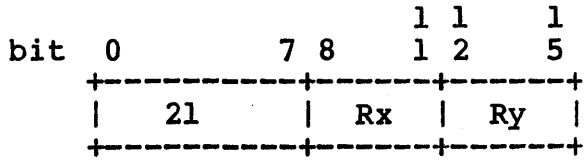
DESCRIPTION

FIXR converts the single-precision real contents of Ry into a two's-complement integer in Rx. Fractions of .5 or more are rounded up to the next higher absolute value integer.

EXCEPTIONS

If integer overflow occurs, Rx is not modified and an integer overflow trap is taken if enabled.

INSTRUCTION FORMAT



FIXT
Truncate to Integer

SYNTAX

```
FIXT     Rx, Ry
```

FUNCTION

```
Rx := trunc(Ry)
```

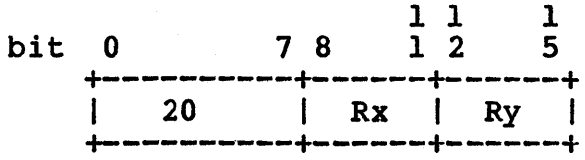
DESCRIPTION

FIXT converts the single-precision real number in Ry into a 32-bit integer in Rx. All bits to the right of the decimal point are lost.

EXCEPTIONS

If integer overflow occurs, Rx is not modified and an integer overflow trap is taken if enabled.

INSTRUCTION FORMAT



FLOAT
Convert Integer to Real

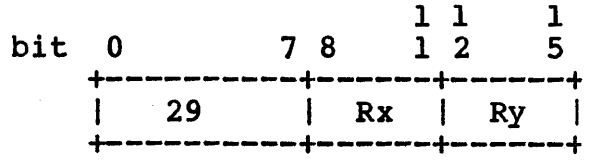
SYNTAX
```
     FLOAT    Rx, Ry
```

FUNCTION
```
     Rx := float(Ry)
```

DESCRIPTION
     FLOAT converts the integer in Ry into a real number in Rx.

EXCEPTIONS
     There are no exceptions.

INSTRUCTION FORMAT



LADDR
Load Data Segment Address

SYNTAX

```
LADDR    Rx,  Ry  [ ,<constant> [,L] ]
LADDR    Rx [,Ry]   ,<constant> [,L]
```

FUNCTION

```
    Rx := constant
or
    Rx := (contents of Ry) + constant
or
    Rx := PC + constant
or
    Rx := PC + Ry + constant
```

DESCRIPTION

The load address instruction stores the effective address into Rx. This instruction does not perform a memory reference, but instead loads a constant from the instruction stream into a register.

The LADDR instruction can be used to load two- or four-byte immediate values and, in indexed mode, can be used to add a constant to a register.

The operation of LADDR is varied by specifying Ry or a code-relative constant. If <constant> is data-relative, LADDR either loads register Rx with <constant> or loads register Rx with the sum of the contents of Ry and <constant>.

If the constant is code-relative, LADDR either loads register Rx with PC + <constant> or loads register Rx with the sum of the contents of Ry and PC + <constant>.

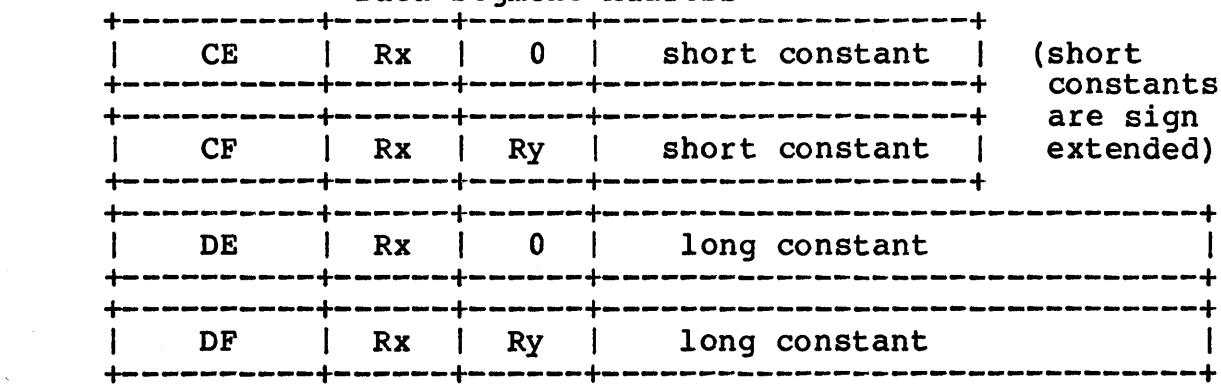
<constant> may be either a data-relative or a code-relative expression. [,L] indicates that <constant> is a 32-bit two's-complement integer. If no constant is specified, a short offset of 0 is assembled.

EXCEPTIONS

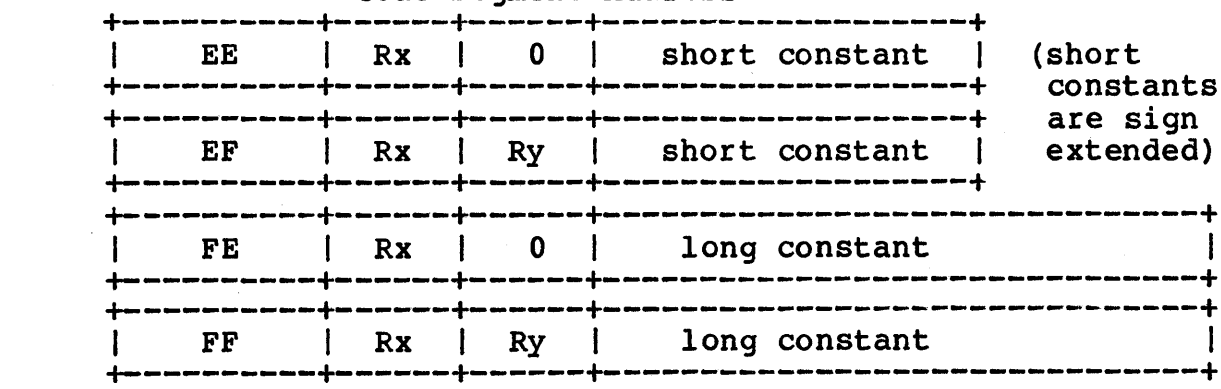
There are no exceptions.

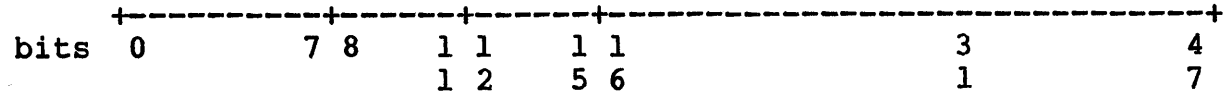
INSTRUCTION FORMAT

Data Segment Address



Code Segment Address





LCOMP
Logical Compare

SYNTAX

```
LCOMP    Rx, Ry
```

FUNCTION

```
IF Rx < Ry THEN Rx := -1;
IF Rx = Ry THEN Rx := 0;
IF Rx > Ry THEN Rx := 1
```

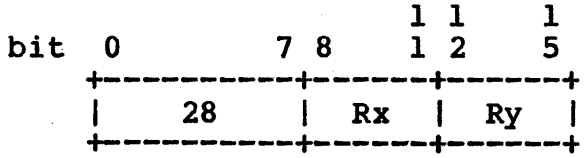
DESCRIPTION

LCOMP compares 32-bit unsigned integers in Rx and Ry and sets Rx to 1, 0, or -1 if Rx is greater than, equal to, or less than Ry, respectively.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



LOAD
Load Word from Data Segment

SYNTAX

```
LOAD    Rx [,Ry]  ,<address> [,L]
LOAD    Rx,  Ry   [,<address> [,L] ]
```

FUNCTION

```
   Rx := contents of address
or
   Rx := contents of (Ry + address)
or
   Rx := contents of (PC + address)
or
   Rx := contents of (PC + Ry + address)
```

DESCRIPTION

The register Rx is loaded with the word at the effective address. Each effective address for a memory reference instruction is explained below.

Displacement. The memory address is the displacement field from the instruction. All memory references are 32-bit virtual addresses. This form references data space.

Ry + displacement. The contents of register Ry are added to the displacement field. Memory is then read or written at this location.

PC + displacement. Instructions that reference code space do so relative to the program counter (PC). PC is added to the displacement field and memory is read from this location.

PC + Ry + displacement. PC is added to the displacement field, the result is added to the contents of Ry. Memory is then read at this location.

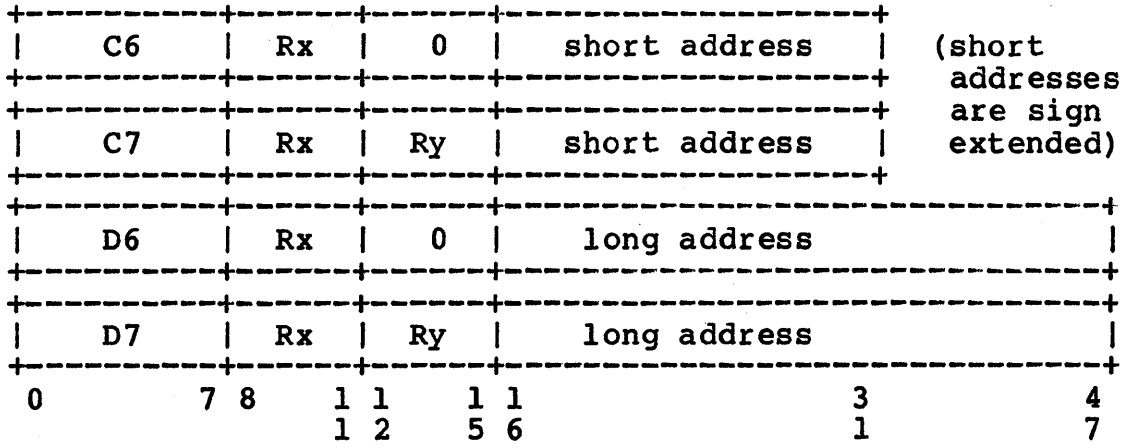
LOAD will generate a load from the code segment if the operand is code-relative (LOADP).

[,L] indicates that <address> is a 32-bit two's-complement integer.

EXCEPTIONS

A data alignment trap may result from attempting to LOAD with the effective address not on a word boundary.

INSTRUCTION FORMAT



LOADB
Load Byte from Data Segment

SYNTAX

```
LOADB    Rx [,Ry]  ,<address> [,L]
LOADB    Rx,  Ry  [,<address> [,L] ]
```

FUNCTION

```
   Rx := contents of address
or
   Rx := contents of (Ry + address)
or
   Rx := contents of (PC + address)
or
   Rx := contents of (PC + Ry + address)
```

DESCRIPTION

The register Rx is loaded with the byte at the effective address. Each effective address for a memory reference instruction is explained below. The byte is stored in bits 24 to 31 of Rx. Bits 0 to 23 are cleared.

Displacement. The memory address is the displacement field from the instruction. All memory references are 32-bit virtual addresses. This form references data space.

Ry + displacement. The contents of register Ry are added to the displacement field. Memory is then read or written at this location.

PC + displacement. Instructions that reference code space do so relative to the program counter (PC). PC is added to the displacement field and memory is read from this location.

PC + Ry + displacement. PC is added to the displacement field, the result is added to the contents of Ry. Memory is then read at this location.

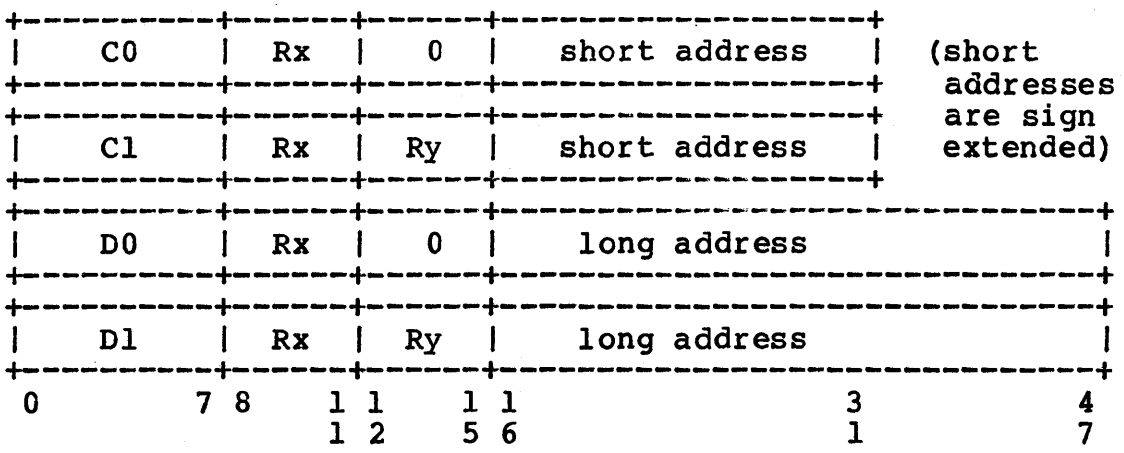
LOADB will generate a load from the code segment if the operand is code-relative (LOADBP).

[,L] indicates that <address> is a 32-bit two's-complement integer.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



LOADD
Load Double Word from Data Segment

SYNTAX

```
LOADD    Rx [,Ry]  ,<address> [,L]
LOADD    Rx,  Ry   [,<address> [,L] ]
```

FUNCTION

```
   Rx := contents of address
or
   Rx := contents of (Ry + address)
or
   Rx := contents of (PC + address)
or
   Rx := contents of (PC + Ry + address)
```

DESCRIPTION

The register Rx is loaded with the double word at the effective address. Each effective address for a memory reference instruction is explained below.

Displacement. The memory address is the displacement field from the instruction. All memory references are 32-bit virtual addresses. This form references data space.

Ry + displacement. The contents of register Ry are added to the displacement field. Memory is then read or written at this location.

PC + displacement. Instructions that reference code space do so relative to the program counter (PC). PC is added to the displacement field and memory is read from this location.

PC + Ry + displacement. PC is added to the displacement field, the result is added to the contents of Ry. Memory is then read at this location.

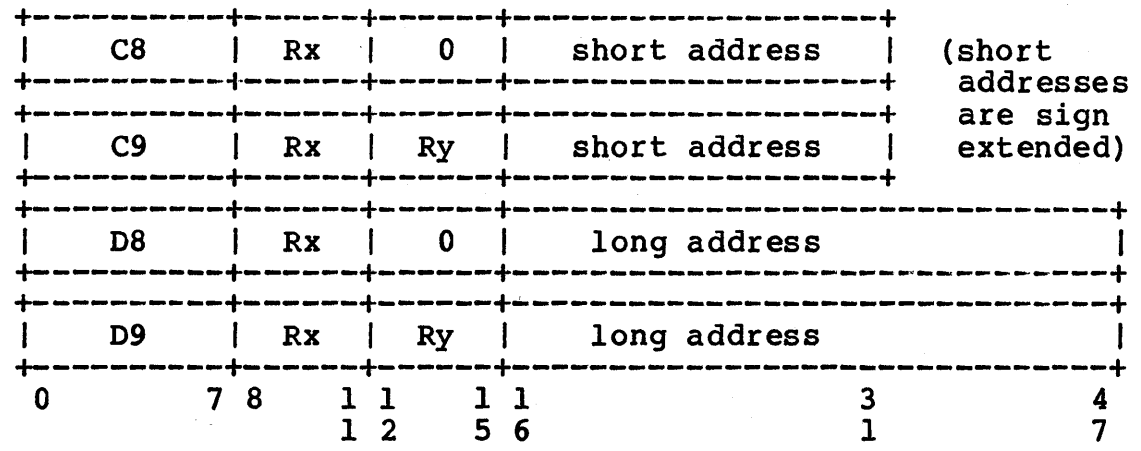
LOADD will generate a load from the code segment if the operand is code-relative (LOADDP).

[,L] indicates that <address> is a 32-bit two's-complement integer.

EXCEPTIONS

A data alignment trap may result from attempting to LOAD with the effective address not on a word boundary.

INSTRUCTION FORMAT



LOADH
Load Half Word from Data Segment

SYNTAX

```
LOADH    Rx [,Ry]  ,<address> [,L]
LOADH    Rx,  Ry  [,<address> [,L] ]
```

FUNCTION

Rx := contents of address

or

Rx := contents of (Ry + address)

or

Rx := contents of (PC + address)

or

Rx := contents of (PC + Ry + address)

DESCRIPTION

The register Rx is loaded with the half word at the effective address. Each effective address for a memory reference instruction is explained below. The half word is stored in bits 16 to 31 of Rx. Bits 0 to 15 are cleared.

Displacement. The memory address is the displacement field from the instruction. All memory references are 32-bit virtual addresses. This form references data space.

Ry + displacement. The contents of register Ry are added to the displacement field. Memory is then read or written at this location.

PC + displacement. Instructions that reference code space do so relative to the program counter (PC). PC is added to the displacement field and memory is read from this location.

PC + Ry + displacement. PC is added to the displacement field, the result is added to the contents of Ry. Memory is then read at this location.

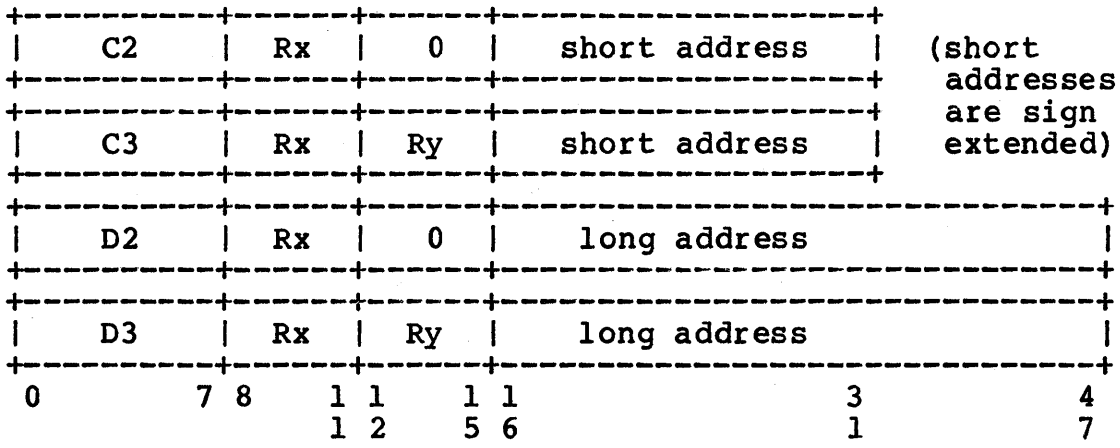
LOADH will generate a load from the code segment if the operand is code-relative (LOADHP).

[,L] indicates that <address> is a 32-bit two's-complement integer.

EXCEPTIONS

A data alignment trap may result from attempting to LOAD with the effective address not on a half word boundary.

INSTRUCTION FORMAT



LOOP
Increment and Branch

SYNTAX

```
LOOP     Rx, <smallval>, <label> [,L]
```

FUNCTION

```
Rx := Rx + smallval;
if Rx < 0 THEN branch to (PC + label)
```

DESCRIPTION

LOOP adds <smallval> to Rx, and if Rx is less than 0, branches to the code <label> from the current location.

<label> can be short (16 bits) or long (32 bits). The long form is indicated by following the label by [,L]. <label> must evaluate to a code-relative value.

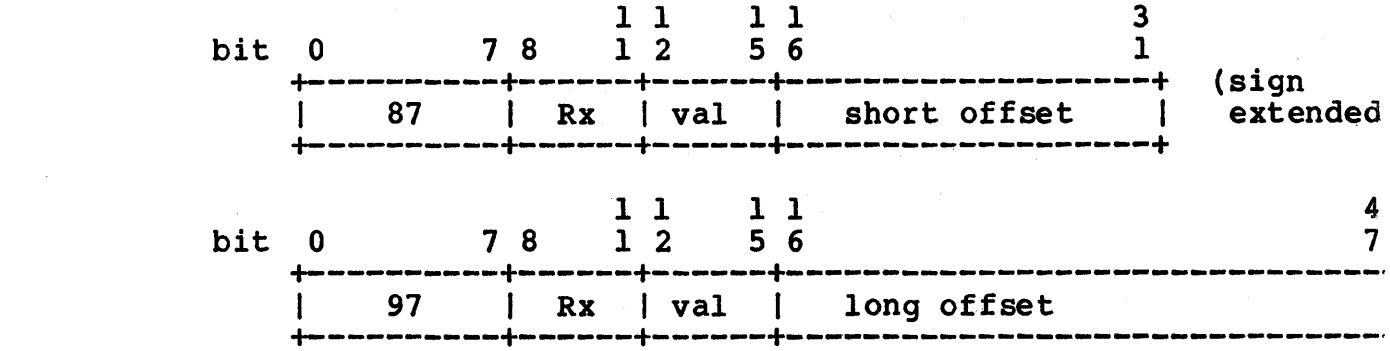
<smallval> is an expression with a value in the range from 0 through 15. The expression cannot contain forward or external labels.

The loop format is similar to the branch format except that no second register is allowed, only a value from 0 to 15, inclusive.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



LSL
Logical Shift Left

SYNTAX

```
LSL     Rx, Ry
LSL     Rx, <smallval>
```

FUNCTION

shift contents of Rx to left by n bits;
Rx[31-(n-1)..31] := 0

where n = <smallval> or contents of Ry mod 32

DESCRIPTION

LSL shifts all bits to the left, inserting 0 on the right, and truncating bits on the left.

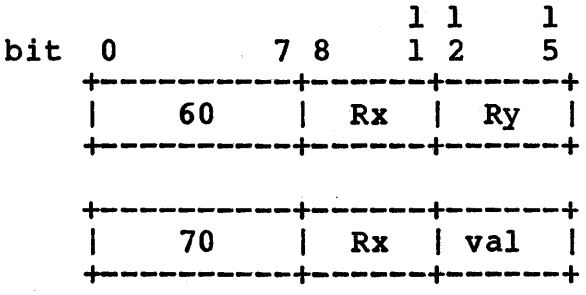
LSL expects two registers in the operand field, or one register and an expression <smallval>.

<smallval> is an expression with a value in the range from 0 through 15. The expression cannot contain forward or external labels.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



LSR
Logical Shift Right

SYNTAX

```
LSR     Rx, Ry
LSR     Rx, <smallval>
```

FUNCTION

shift contents of Rx to right by n bits;
Rx[0..n-1] := 0;

where n = <smallval> or the contents of Ry mod 32

DESCRIPTION

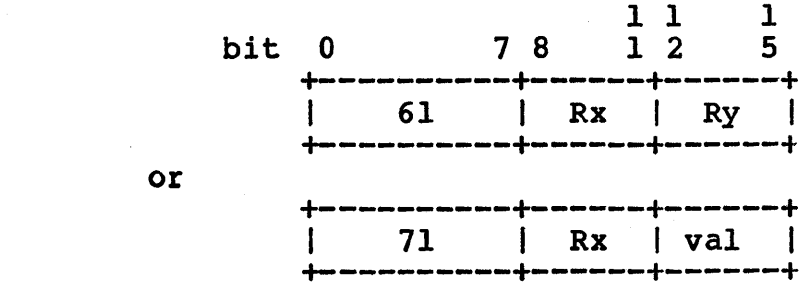
LSR shifts all bits in Rx to the right by the number of bit positions specified by <smallval> or contained in Ry. LSR inserts 0 on the left of Rx and truncates bits on the right.

<smallval> is an expression with a value in the range from 0 through 15. The expression cannot contain forward or external labels.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



MAKEDR
Round Double Real to Real

SYNTAX

```
MAKEDR   Rx, Ry
```

FUNCTION

```
Rx := real(RPy)
```

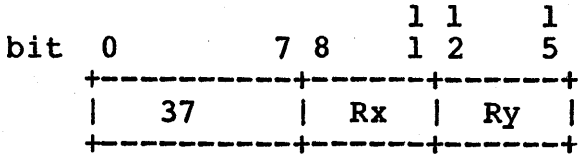
DESCRIPTION

MAKEDR rounds the double-real number in RPy into a single-precision real number in Rx.

EXCEPTIONS

If real overflow or underflow occurs, a real overflow or underflow trap is taken if enabled.

INSTRUCTION FORMAT



MAKERD
Convert Real to Long Real

SYNTAX

```
MAKERD   Rx, Ry
```

FUNCTION

```
RPx := long(Ry)
```

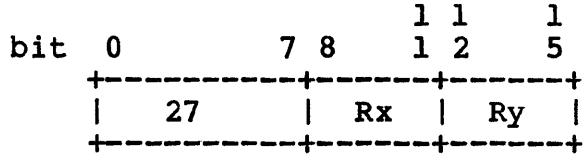
DESCRIPTION

MAKERD convert the real number in Ry into a long real number in register RPx.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



MOVE
Move Register

SYNTAX

```
MOVE    Rx, Ry
MOVE    Rx, <smallval>
MOVE    SRx, Ry
MOVE    Rx,  SRy
```

FUNCTION

```
  Rx := Ry
or
  Rx := <smallval>
or
  Special Register Rx := Ry
or
  Rx := Special Register Ry
```

DESCRIPTION

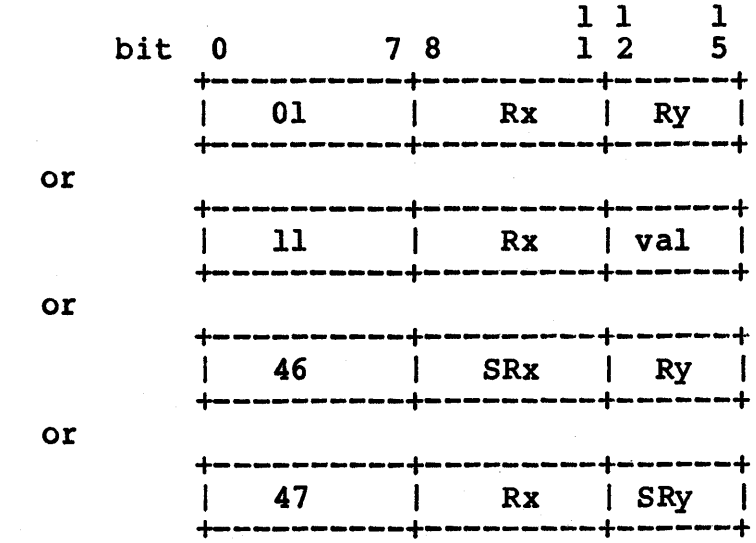
MOVE places a value in a register. There are four forms of this instruction. The first two forms copy either the contents of a register or a <smallval> into register Rx. The second two forms use one general register and one special register. Either a general register is copied to a special register, or a special register is copied to a general register.

<smallval> is an expression with a value in the range from 0 through 15. The expression cannot contain forward or external labels.

EXCEPTIONS

If MOVE is attempted with SRx or SRy and not in kernel mode, a kernel violation trap results.

INSTRUCTION FORMAT



MPY
Integer Multiply

SYNTAX

```
MPY     Rx, Ry
MPY     Rx, <smallval>
```

FUNCTION

```
  Rx := Rx * Ry
or
  Rx := Rx * smallval
```

DESCRIPTION

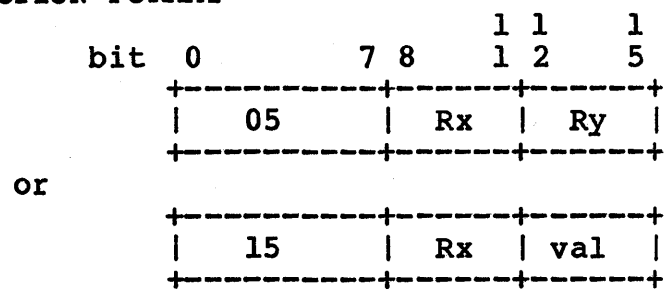
MPY multiplies the contents of Rx and Ry, or Rx and an expression <smallval>, and puts the result in Rx.

<smallval> is an expression with a value in the range from 0 through 15. The expression cannot contain forward or external labels.

EXCEPTIONS

If integer overflow occurs, the least-significant 32 bits of the result are placed in Rx, the most significant bits are discarded, and the integer overflow trap is taken if enabled.

INSTRUCTION FORMAT



NEG
Integer Negate

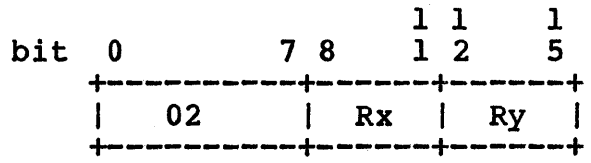
SYNTAX
```
     NEG       Rx, Ry
```

FUNCTION
```
     Rx := two's-complement(Ry)
```

DESCRIPTION
     NEG negates the contents of Ry using two's-complement arithmetic, and puts the result in Rx.

EXCEPTIONS
     NEG of -2**31 produces integer overflow.

INSTRUCTION FORMAT



NOT
Logical NOT

SYNTAX

```
NOT     Rx, Ry
NOT     Rx, <smallval>
```

FUNCTION

```
Rx := one's-complement(Ry)
```

or

```
Rx := one's-complement(smallval)
```

DESCRIPTION

NOT complements the bits in Ry or <smallval> and puts the result in Rx.

If <smallval> is specified, it is first loaded into bits 28 through 31 of Rx, and bits 0 through 27 are set to 0. If Ry is specified, its contents are first put in Rx.

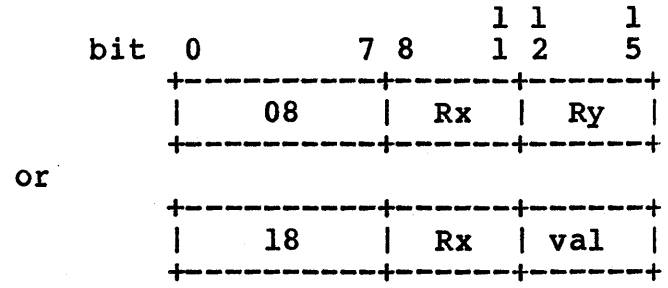
Then, NOT complements each bit in Rx and leaves the result in Rx.

<smallval> is an expression with a value in the range from 0 through 15. The expression cannot contain forward or external labels.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



OR
Logical OR

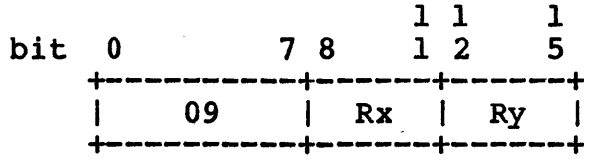
SYNTAX
```
     OR        Rx, Ry
```

FUNCTION
```
     Rx := Rx OR Ry
```

DESCRIPTION
OR performs logical OR on the contents of Rx and Ry and puts the result in Rx.

EXCEPTIONS
There are no exceptions.

INSTRUCTION FORMAT



RADD
Real Add

SYNTAX

```
RADD     Rx, Ry
```

FUNCTION

Rx := Rx + Ry

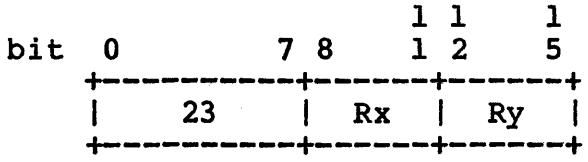
DESCRIPTION

RADD adds the 32-bit real numbers in Rx and Ry and puts their sum in Rx.

EXCEPTIONS

On overflow, Rx is set to the highest real number with the appropriate sign bit. On underflow, Rx is set to 0. In either case, a real under- or overflow trap is taken if enabled.

INSTRUCTION FORMAT



RCOMP
Real Compare

SYNTAX

```
RCOMP    Rx, Ry
```

FUNCTION

```
IF Rx < Ry then Rx := -1;
IF Rx = Ry then Rx := 0;
IF Rx > Ry then Rx := 1
```

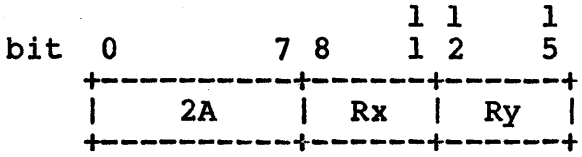
DESCRIPTION

RCOMP compares the real numbers in registers Rx and Ry using sign magnitude form, and sets Rx to -1, 0, or 1 depending on whether Rx is less then, equal to, or greater than Ry, respectively.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



RDIV
Real Divide

SYNTAX

```
RDIV     Rx, Ry
```

FUNCTION

Rx := Rx/Ry

DESCRIPTION

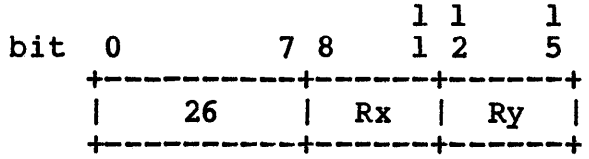
RDIV divides the 32-bit real number in Rx by the 32-bit real in Ry and puts the result in Rx.

EXCEPTIONS

On overflow, Rx is set to the largest real number with the appropriate sign bit. On underflow, Rx is set to 0. If Ry is zero, Rx is set to the largest real number with the appropriate sign bit set. In any case, a real underflow, overflow, or divide by zero trap is taken if enabled.

An attempt to divide by zero leaves Rx unmodified, and a real divide by zero trap is taken if enabled.

INSTRUCTION FORMAT



REM
Integer Remainder

SYNTAX

```
REM      Rx, Ry
```

FUNCTION

```
Rx := Rx - ((Rx/Ry) * Ry)
```

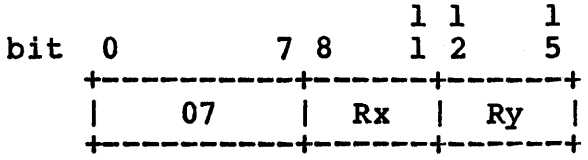
DESCRIPTION

REM divides the integer in Rx by the integer in Ry and puts the remainder in Rx.

EXCEPTIONS

Integer overflow can occur when the largest negative integer is divided by -1, and an integer overflow trap is taken if enabled. An attempt to divide by 0 leaves Rx unmodified, and a divide-by-zero trap is taken if enabled.

INSTRUCTION FORMAT



RET
Return

SYNTAX

```
RET   Rx, Ry
```

FUNCTION

```
Rx := PC + 2;
PC := Ry
```

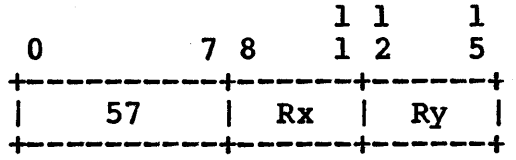
DESCRIPTION

RET puts the address of the next sequential instruction in Rx and branches to the address held in Ry. This is commonly used to return from a subroutine called by CALL or CALLR. RET can also call a subroutine if its absolute address is known.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



RMPY
Real Multiply

SYNTAX

```
RMPY     Rx, Ry
```

FUNCTION

```
Rx := Rx * Ry
```

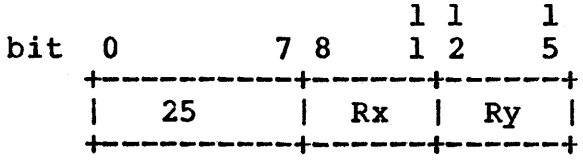
DESCRIPTION

RMPY multiplies the 32-bit real numbers in Rx and Ry and puts the product in Rx.

EXCEPTIONS

On overflow, Rx is set to the highest real number with the appropriate sign bit. On underflow, Rx is set to 0. In either case, a real under- or overflow trap is taken if enabled.

INSTRUCTION FORMAT



RNEG
Real Negate

SYNTAX

```
RNEG     Rx, Ry
```

FUNCTION

```
Rx := -Ry
```

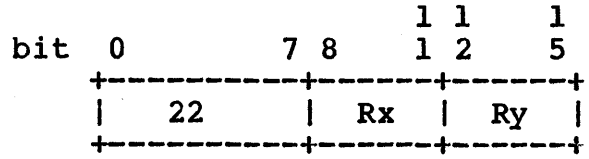
DESCRIPTION

RNEG negates the real number in Ry and puts the result in Rx.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



RSUB
Real Subtract

SYNTAX

```
RSUB     Rx, Ry
```

FUNCTION

```
Rx := Rx - Ry
```

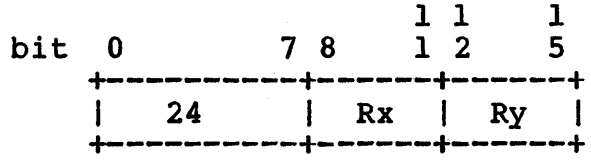
DESCRIPTION

RSUB subtracts the real number in Ry from the real in Rx and puts the difference in Rx.

EXCEPTIONS

If overflow occurs, Rx is set to the largest real number with the appropriate sign bit, and the real overflow trap is taken if enabled. If underflow occurs, Rx is set to 0, and the real underflow trap is taken if enabled.

INSTRUCTION FORMAT



SBIT
Set Bit

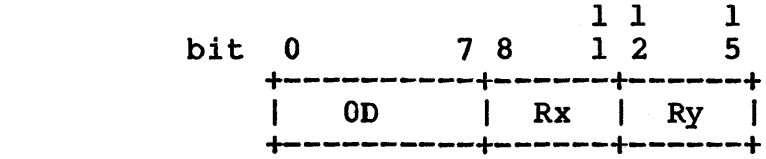
SYNTAX
```
     SBIT     Rx, Ry
```

FUNCTION
```
     RPx [bit Ry mod 64] := 1
```

DESCRIPTION
     SBIT sets the bit in RPx that is specified in Ry mod 64.

EXCEPTIONS
     There are no exceptions.

INSTRUCTION FORMAT



SEB
Sign Extend Byte

SYNTAX

```
     SEB       Rx, Ry
```

FUNCTION

```
     Rx [bits 0..23]  := Ry [bit 24] ;
     Rx [bits 24..31] := Ry [bits 24..31]
```

DESCRIPTION

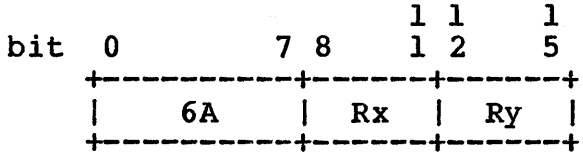
SEB converts a byte-size integer into a full-word integer.

SEB sets bits 0 to 23 of Rx to bit 24 of Ry, and copies bits 24 to 31 of Ry into bits 24 to 31 of Rx.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



SEH
Sign Extend Half Word

SYNTAX
```
     SEH       Rx, Ry
```

FUNCTION
```
     Rx [bits 0..15]  := Ry [bit 16] ;
     Rx [bits 16..31] := Ry [bits 16..31]
```

DESCRIPTION

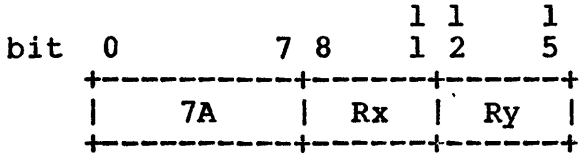
SEH converts a half-word integer into a full-word integer.

SEH sets bits 0 to 23 of Rx to bit 24 of Ry, and copies bits 24 to 31 of Ry into bits 24 to 31 of Rx.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



STORE
Store Word

SYNTAX

```
STORE    Rx,  <address> [,L]
STORE    Rx,  Ry  [ ,<address> [,L] ]
```

FUNCTION

store Rx at address

or

store Rx at (Ry + address)

DESCRIPTION

The register Rx is stored at the effective address. Each effective address for a memory reference instruction is explained below.

Displacement. The memory address is the displacement field from the instruciton. All memory references are 32-bit virtual addresses. This form references data space.

Ry + displacement. The contents of register Ry are added to the displacement field. Memory is then read or written at this location.

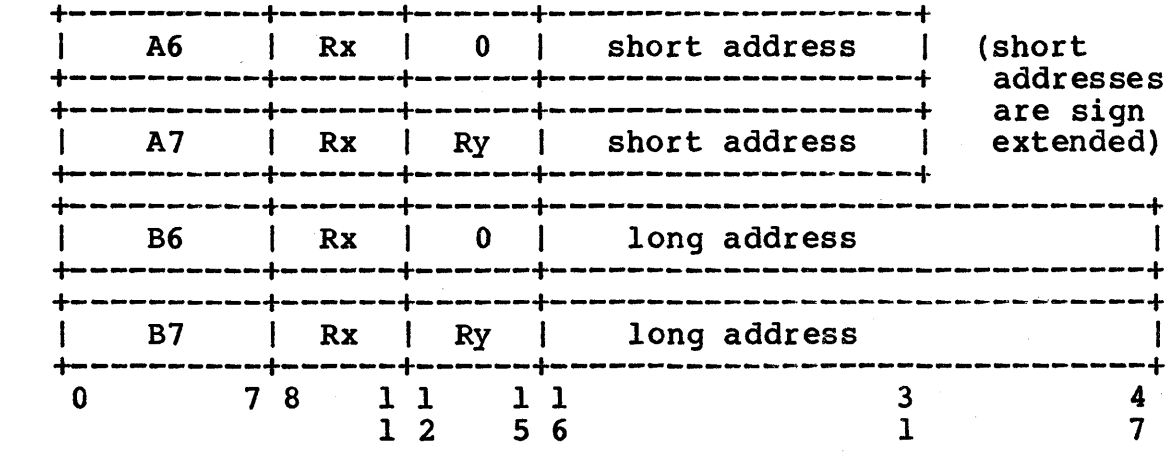
<address> is a data-relative expression.

[,L] indicates that <address> is a 32-bit integer.

EXCEPTIONS

A data alignment trap may result from attempting to STORE with the effective address not on a word boundary.

INSTRUCTION FORMAT



STOREB
Store Byte

SYNTAX

```
STOREB   Rx,  <address> [,L]
STOREB   Rx,  Ry  [ ,<address> [,L] ]
```

FUNCTION

store byte in Rx at address

or

store byte in Rx at (Ry + address)

DESCRIPTION

The byte in bits 24 to 31 of register Rx are stored at the effective address. Each effective address for a memory reference instruction is explained below.

Displacement. The memory address is the displacement field from the instruciton. All memory references are 32-bit virtual addresses. This form references data space.

Ry + displacement. The contents of register Ry are added to the displacement field. Memory is then read or written at this location.

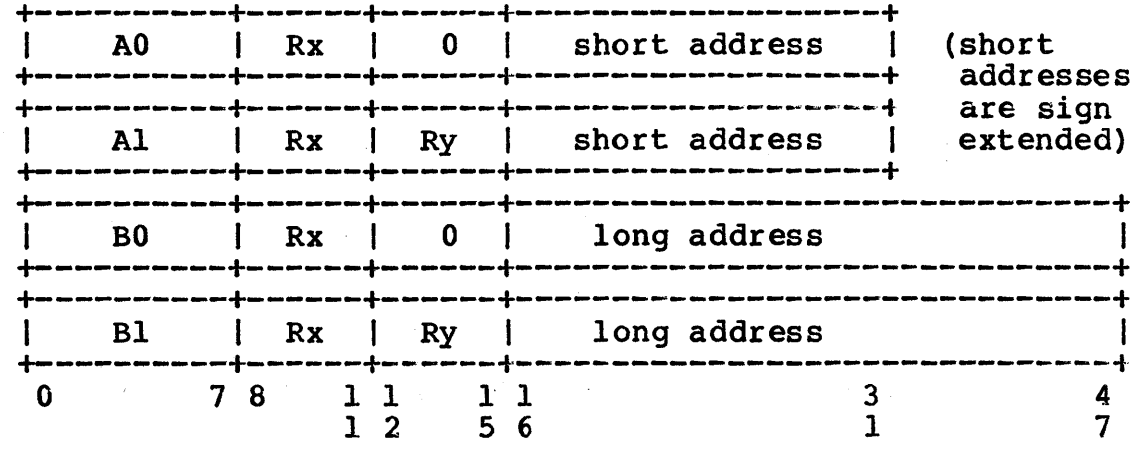
<address> is a data-relative expression.

[,L] indicates that <address> is a 32-bit integer.

EXCEPTIONS

A data alignment trap may result from attempting to STOREB with the effective address not on a word boundary.

INSTRUCTION FORMAT



STORED
Store Double Word

SYNTAX

```
STORED   Rx,  <address> [,L]
STORED   Rx,  Ry  [ ,<address> [,L] ]
```

FUNCTION

store double word in Rx at address

or

store double word in Rx at (Ry + address)

DESCRIPTION

The double word in Rx and R(x+1) is stored at the effective address. Each effective address for a memory reference instruction is explained below.

Displacement. The memory address is the displacement field from the instruciton. All memory references are 32-bit virtual addresses. This form references data space.

Ry + displacement. The contents of register Ry are added to the displacement field. Memory is then read or written at this location.

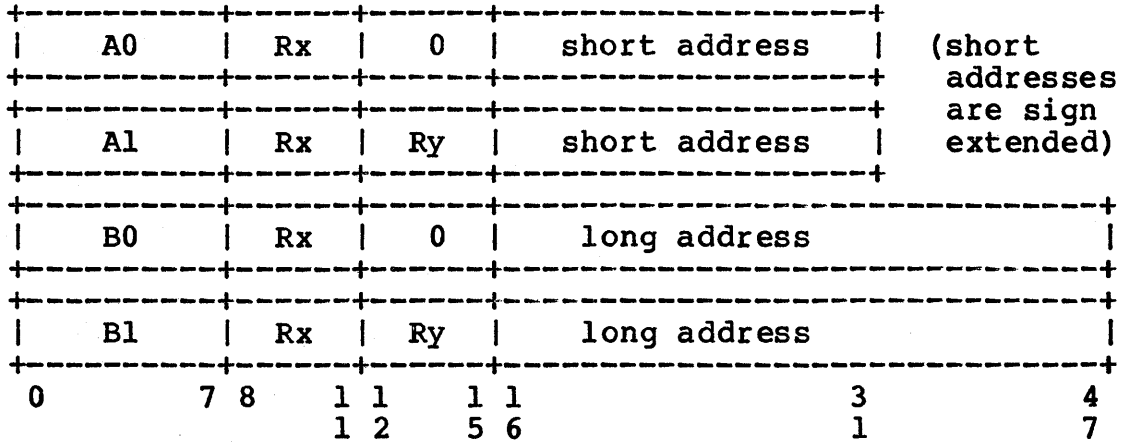
<address> is a data-relative expression.

[,L] indicates that <address> is a 32-bit integer.

EXCEPTIONS

A data alignment trap may result from attempting to STORED with the effective address not on a word boundary.

INSTRUCTION FORMAT



STOREH
Store Half Word

SYNTAX

```
STOREH   Rx,  <address> [,L]
STOREH   Rx,  Ry  [ ,<address> [,L] ]
```

FUNCTION

store half word in Rx at address

or

store half word in Rx at (Ry + address)

DESCRIPTION

The half word in bits 16 to 31 of register Rx are stored at the effective address. Each effective address for a memory reference instruction is explained below.

Displacement. The memory address is the displacement field from the instruciton. All memory references are 32-bit virtual addresses. This form references data space.

Ry + displacement. The contents of register Ry are added to the displacement field. Memory is then read or written at this location.

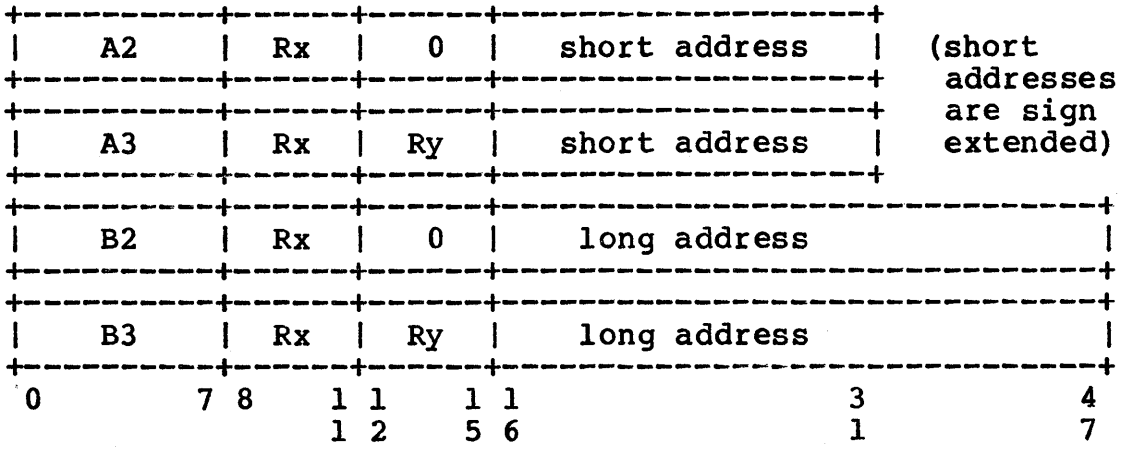
<address> is a data-relative expression.

[,L] indicates that <address> is a 32-bit integer.

EXCEPTIONS

A data alignment trap may result from attempting to STOREH with the effective address not on a word boundary.

INSTRUCTION FORMAT



SUB
Integer Subtract

SYNTAX

```
SUB     Rx, Ry
SUB     Rx, <smallval>
```

FUNCTION

```
    Rx := Rx - Ry
or
    Rx := Rx - smallval
```

DESCRIPTION

SUB subtracts the integer in Ry, or the integer in <smallval>, from the integer in Rx and puts the difference in Rx.

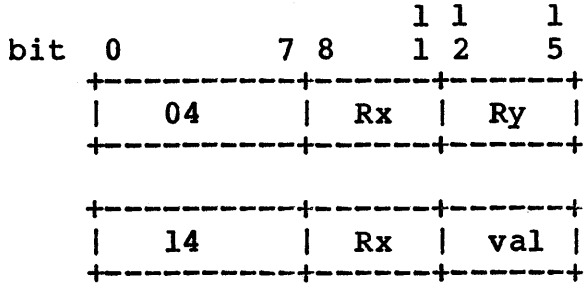
SUB expects two registers in the operand field, or one register and an expression <smallval>.

<smallval> is an expression with a value in the range from 0 through 15. The expression cannot contain forward or external labels.

EXCEPTIONS

If integer overflow occurs, the least significant 32 bits of the result are placed in Rx, the most significant bits are discarded, and the integer overflow trap is taken if enabled.

INSTRUCTION FORMAT



TBIT
Test Bit

SYNTAX

```
TBIT     Rx, Ry
```

FUNCTION

```
Rx[31] := RPx[Ry mod 64];
Rx[0..30] := 0
```

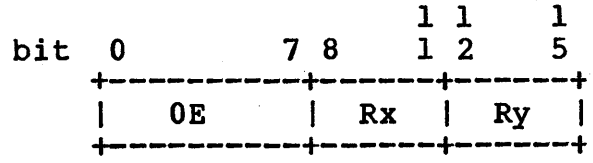
DESCRIPTION

TBIT sets bit 31 of Rx to the bit of RPx that is specified by Ry mod 64. Other bits of Rx are set to 0. Ry contains an integer from 0 to 63, inclusive.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



TEST
Test Values

SYNTAX

```
TEST    Rx <relop> Ry
TEST    Rx <relop> <smallval>
```

FUNCTION

tests the truth of expressions with relational operators

DESCRIPTION

TEST compares Rx with Ry, or Rx with a small value, and sets Rx to 1 (true) or 0 (false).

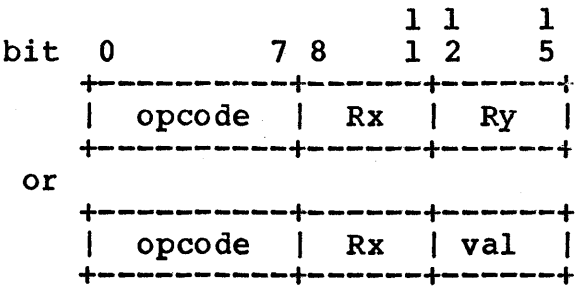
A <relop> is any of the relational operators <, <=, =, >, >=, or <>.

<smallval> is an expression with a value in the range from 0 through 15. The expression cannot contain forward or external labels.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



TEST instruction	opcode
TEST Rx > Ry	50
TEST Rx < Ry	51
TEST Rx = Ry	52
TEST Rx <= Ry	58
TEST Rx >= Ry	59
TEST Rx <> Ry	5A
TEST Rx > value	54
TEST Rx < value	55
TEST Rx = value	56
TEST Rx <= value	5C
TEST Rx >= value	5D
TEST Rx <> value	5E

XOR
Logical Exclusive OR

SYNTAX

```
XOR      Rx, Ry
```

FUNCTION

```
Rx := Rx XOR Ry
```

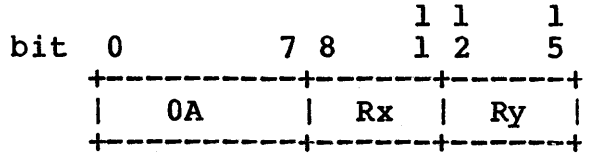
DESCRIPTION

XOR performs a logical exclusive OR between the 32 bits of Rx and Ry, and puts the result in Rx.

EXCEPTIONS

There are no exceptions.

INSTRUCTION FORMAT



Glossary

code-relative expression - an expression containing global or code segment labels.

code segment label - a label which is defined in a code section.

comment - a message in the program source code for organization and clarity. In an AS program, any characters after a ";" in a line are considered comments and are ignored by the assembler.

data-relative expression - an expression containing only global or data segment labels.

exception - either an interrupt or kernel trap which can occur when executing an instruction.

expression - a group of labels, operators, and/or numbers which can be arithmetically evaluated to a single value.

forward label - a label that is declared in the code at some point after the current location.

instruction - the assembled result of a mnemonic. An instruction includes an opcode, operands, and optional address fields, as shown in the instruction format diagrams.

label - one of three types: code, data, and absolute. A label identifies a line of code or data and can be referenced in an expression to calculate a location in a program. Labels consist of any sequence of alphabetic characters, numbers, "__", "$", and ">", but do not begin with a number.

least significant bits - the bits furthest from bit 0 in a word, shown at the right-hand end of word diagrams.

long value - a value which is represented in 32 bits, flagged in some instructions by ",L". Many assembler instruction formats have a long or short address field in addition to the opcode and operand fields.

mod - "X mod Y" is the integer remainder value after dividing X by Y. Examples: 10 mod 2 = 0; 15 mod 4 = 3; 17 mod 32 = 0; 65 mod 32 = 1; (Ry mod 32 = x mod 32, where x is the contents of Ry).

most significant bits - the bits closest to bit 0 in a word, shown at the left-hand end of word diagrams.

opcode - a two-digit hexadecimal code, occupying the most significant 8 bits of an instruction, which uniquely identifies the functionality of an instruction to the Ridge processor. The assembler derives opcodes from the mnemonic names.

overflow - an attempt to calculate a number which is too large to be represented.

program counter (PC) - the address of the instruction currently being executed by the processor. The PC is incremented by the processor in order to execute the instructions sequentially, and frequently manipulated by the branch instructions in order to commence execution in different parts of the code.

pseudo-instruction - an assembler "command" (used much like the mnemonics) which helps the user organize and define data and code, but does not generate opcodes.

RPx (register pair x) - The two registers Rx and R(x+1) mod 16. RP3 consists of R3 and R4; R15 consists of R15 and R0.

Rx, Ry - in syntax notation, Rx or Ry stand for one of general registers 0 through 15, like R4 or R9.

section - of two types: code or data. A section is a division of an assembly program which defines the runtime segment into which the instructions in it will be assembled.

segment - the portion of a linked program file which contains either code or data.

short value - a value which is represented in 16 bits. Many assembly instruction formats have a long or short address field in addition to the opcode and operand fields.

sign extended - when a value is sign-extended in an instruction format, its sign bit is duplicated throughout the most significant (lower numbered) bits of the constant field of the instruction.

small value (smallval) - in this AS manual, a small value is an expression which evaluates to an integer in the range 0 to 15. A small value is often assembled into the second 4-bit operand field of an instruction.

SRx - in syntax notation, SRx refers to one of special registers o to 15, like SR15 or SR6.

trap - When enabled, a trap detects the occurrence of a certain exceptional condition and transfers control to the kernel. "Trap bits" in SR10 determine which traps are enabled or disabled.

underflow (real) - an attempt to calculate a number which is too small to be represented.

Alphabetic Index

Functional Instruction List

MEMORY REFERENCE INSTRUCTIONS
  Load Instructions
    LOAD, LOADB, LOADD, LOADH

  Store Instructions
    STORE, STOREB, STORED, STOREH

  Load Address Instruction
    LADDR

INTEGER ARITHMETIC INSTRUCTIONS
  Single-Precision
    ADD, DIV, NEG, MPY, REM, SUB

  Extended-Precision
    EADD, EDIV, EMPY, ESUB

REAL ARITHMETIC INSTRUCTIONS
  Single-Precision
    RNEG, RADD, RSUB, RMPY, RDIV, FLOAT, FIXR, FIXT, MAKERD

  Double-Precision
    DFIXR, DFIXT, DFLOAT, DRADD, DRDIV, DRMPY, DRNEG, DRSUB, MAKEDR

BIT MANIPULATION INSTRUCTIONS
    ASL, ASR, CBIT, CSL, DLSL, DLSR, LSL, LSR, SBIT, TBIT

LOGICAL INSTRUCTIONS
    AND, NOT, OR, XOR

TEST INSTRUCTION
    TEST

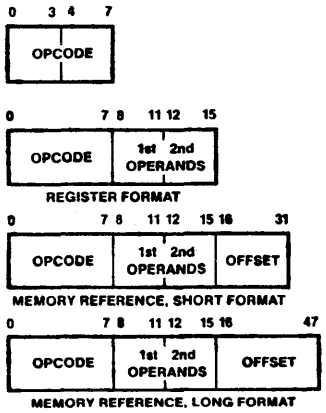
DATA MOVEMENT INSTRUCTION
    MOVE

COMPARISON INSTRUCTIONS
    DCOMP, DRCOMP, LCOMP, RCOMP

SIGN EXTEND INSTRUCTIONS
    SEB, SEH

BRANCH AND CALL INSTRUCTIONS
    BR, CALL, CALLR, LOOP, RET

INSTRUCTION FORMATS:



Register Format

Memory Reference Format

	Segment Referenced	Length
	Code	Short
	Code	Long
	Data	Short
	Data	Long
	Data	Short
	Data	Long
	Code	Short
	Code	Long

RIDGE OPCODE MAP

Least Significant Nibble (Hex), Opcode (4:7)

Most Significant Nibble (Hex), Opcode (0:3)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		MOVE	NEG	ADD	SUB	MPY	DIV	REM	NOT	OR	XOR	AND	CBIT	SBIT	TBIT	CHK
1	NOP	MOVE Immed		ADD Immed	SUB Immed	MPY Immed			NOT Immed			AND Immed				CHK Immed
2	FIXT	FIXR	RNEG	RADD	RSUB	RMPY	RDIV	MAKERD	LCOMP	FLOAT	RCOMP		EADD	ESUB	EMPY	EDIV
3	DFIXT	DFIXR	DRNEG	DRADD	DRSUB	DRMPY	DRDIV	MAKEDR	DCOMP	DFLOAT	DRCOMP	TRAP				
4	SUS	LUS	RUM	LDREGS	TRANS	DIRT	MOVE SR←R	MOVE R←SR					MAINT		READ	WRITE
5	>	TEST <	=	CALLR	TEST Immediate >	TEST Immediate <	TEST Immediate =	RET	< =	TEST > =	<>	KCALL	TEST Immediate < =	TEST Immediate > =	TEST Immediate <>	
6	LSL	LSR	ASL	ASR	DLSL	DLSR			CSL		SEB					
7	LSL Immed	LSR Immed	ASL Immed	ASR Immed	DLSL Immed	DLSR Immed			CSL Immed		SEH					
8	> BR		= BR	CALL	> BR Immediate	< BR Immediate	= BR Immediate	LOOP	< = BR		<> BR	BR	< = BR Immediate	> = BR Immediate	<> BR Immediate	
9	>		=		>	<	=		< =		<>		< =	> =	<>	
A	STOREB X		STOREH X				STORE X		STORED X							
B	X		X				X		X							
C		X		X				X		X					LADDR X	
D	LOADB X		LOADH X				LOAD X		LOADD X						X	
E		X		X				X		X					LADDRP X	
F		X		X				X		X					X	

X = Indexed (i.e., target address is further offset by a register named in the second operand field).
Immediate (Immed) = the second operand field contains a value.

Introduction to the Shell

This document is based on a paper from Bell Laboratories.

The Shell is the interactive and programmable command interpreter. This document covers the syntax of ROS commands and shell program scripts. The reader should have a a basic understanding of ROS and a high-level programming language.

1.1 Simple commands

A command consists of one or more words separated by blanks. The first word is the command name; other words are arguments (or parameters) for that command.

```
who
```

is a command that prints the names of logged-in users. The command

```
ls – l
```

prints a list of files in the user's directory. The argument – l tells *ls* to display special information about each file. Try entering **ls** without any arguments.

1.2 Background commands

Normally, the shell executes a command, waits for it to finish, then re-displays the command prompt "$" for additional user input. A command execution process may be run in the "background"; the system creates a separate "detached" process and immediately prompts the user for additinal input:

```
cc pgm.c &
```

calls the C compiler to compile the file **pgm.c** . The trailing & character tells the shell not to wait for the command to finish, but rather to create a background process for it and return immediately to the system prompt. When a command is run in the background, ROS displays its background process number for the user's information. A list of all currently active background processes is available through the *ps* command.

1.3 Input output redirection

Most commands send output to the standard output device, normally the user's terminal. Output may be redirected to a different device or file by the ">" character:

```
ls – l > file
```

The output of **ls – l** is sent directly to *file*. If *file* does not exist at that time, the shell creates it. If it exists, the shell overwrites it with the new output. The ">>" symbol tells the shell to append output to *file*, but do not overwrite it:

```
ls – l >> file
```

In the example above, *file* will end up with the **ls** output at the end of whatever may have been in it initially.

When command input is normally expected from the terminal, input may be respecified as being from a file:

```
wc < file
```

The *wc* ("wordcount") command reads its input from *file* and prints its output (the number of characters, words and lines found) on the terminal. Input and output can both be redirected at the same time:

```
wc < infile > outfile
```

1.4 Pipes and filters

A particular set of data may need to be processed through several commands. Rather than create several files, each containing permutations of the data in preparation for the next step, the output of one command may be "piped" directly to the input of another by means of the "|" pipe operator:

```
ls – l  wc
```

Two commands connected in this way constitute a *pipeline* and the overall effect is the same as

```
ls – l > file; wc < file
```

except that no *file* is used. See the sh(1) and pipe(2) pages of the ROS Reference Manual. Pipes are unidirectional. Both the **ls** and **wc** processes run in synchrony; **ls** is halted when the pipe is full and it resumes when **wc** is ready for more input.

A *filter* is a command that reads input, transforms it in some way, and outputs the result. One such filter is *grep,* which prints all lines of an input file(s) that contain a particular string. For example,

```
grep badword file52
```

prints all lines of **file52** that contain the word **badword.**

```
ls  grep badword
```

pipes the **ls** output (a list of the user's files) to **grep badword,** and prints all lines of all files that contain the word **badword.** In the case above, no files are modified, created, or destroyed; output is to the terminal.

Another useful filter is *sort.* For example,

```
who  sort
```

prints an alphabetically sorted list of logged in users.

A pipeline may consist of more than two commands, for example,

```
ls  grep old  wc – l
```

prints the number of file names in the current directory containing the string *old.*

1.5 File name generation

Many commands accept arguments which are file names. For example,

```
ls – l main.c
```

prints information relating to the file *main.c*.

The shell provides a mechanism for generating a list of file names that match a pattern. For example,

```
ls – l c
```

generates, as arguments to *ls,* all file names in the current directory that end in *.c*. The character is a pattern that will match any string including the null string. In general *patterns* are specified as follows.

- ***** Matches any string of characters including the null string.
- **?** Matches any single character.

[...] Matches any one of the characters enclosed. A pair of characters separated by a minus will match any character lexically between the pair.

For example,

```
[a-z]*
```

matches all names in the current directory beginning with one of the letters *a* through *z*.

```
/usr/fred/test/?
```

matches all names in the directory **/usr/fred/test** that consist of a single character. If no file name is found that matches the pattern then the pattern is passed, unchanged, as an argument.

This mechanism is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

```
echo /usr/fred/*/core
```

finds and prints the names of all *core* files in sub-directories of **/usr/fred.** (*echo* prints its arguments, separated by blanks.) This last feature can be expensive, requiring a scan of all sub-directories of **/usr/fred.**

There is one exception to the general rules given for patterns. The character '.' at the start of a file name must be explicitly matched.

```
echo *
```

will therefore echo all file names in the current directory not beginning with '.'.

```
echo .*
```

will echo all those file names that begin with '.'. This avoids inadvertent matching of the names '.' and '..' which mean 'the current directory' and 'the parent directory' respectively. (Notice that *ls* suppresses information for the files '.' and '..'.)

1.6 Quoting

Characters that have a special meaning to the shell, such as < > * ? &, are called metacharacters. A complete list of metacharacters is given in appendix B. Any character preceded by a \ is *quoted* and loses its special meaning, if any. The \ is elided so that

```
echo \?
```

will echo a single ? , and

```
echo \\
```

will echo a single \. To allow long strings to be continued over more than one line the sequence **newline** is ignored.

\ is convenient for quoting single characters. When more than one character needs quoting the above mechanism is clumsy and error prone. A string of characters may be quoted by enclosing the string between single quotes. For example,

```
echo xx'****'xx
```

will echo

```
xx****xx
```

The quoted string may not contain a single quote but may contain newlines, which are preserved. This quoting mechanism is the most simple and is recommended for casual use.

A third quoting mechanism using double quotes is also available that prevents interpretation of some but not all metacharacters. Discussion of the details is deferred to section 3.4.

1.7 Prompting

When the shell is used from a terminal it will issue a prompt before reading a command. By default this prompt is '$ '. It may be changed by saying, for example,

PS1=yesdear

that sets the prompt to be the string *yesdear*. If a newline is typed and further input is needed then the shell will issue the prompt '> '. Sometimes this can be caused by mistyping a quote mark. If it is unexpected then an interrupt (DEL) will return the shell to read another command. This prompt may be changed by saying, for example,

PS2=more

1.8 The shell and login

Following *login* (1) the shell is called to read and execute commands typed at the terminal. If the user's login directory contains the file **.profile** then it is assumed to contain commands and is read by the shell before reading any commands from the terminal.

1.9 Summary

- **ls**
  Print the names of files in the current directory.
- **ls >file**
  Put the output from *ls* into *file*.
- **ls wc – l**
  Print the number of files in the current directory.
- **ls grep old**
  Print those file names containing the string *old*.
- **ls grep old wc – l**
  Print the number of files whose name contains the string *old*.
- **cc pgm.c &**
  Run *cc* in the background.

2.0 Shell procedures

The shell may be used to read and execute commands contained in a file. For example,

```
sh file [ args ... ]
```

calls the shell to read commands from *file*. Such a file is called a *command procedure* or *shell procedure*. Arguments may be supplied with the call and are referred to in *file* using the positional parameters **$1, $2, ...** For example, if the file *wg* contains

```
who  grep $1
```

then

```
sh wg fred
```

is equivalent to

```
who  grep fred
```

All files have three access attributes, *read, write* and *execute*. Each attribute may or may not be set, thus allowing or disallowing the action on the file. The command *chmod* (1) may be used to set the execute status of a file and make it executable:

```
chmod +x wg
```

sets execute status for file *wg*. Following this, the command

```
wg fred
```

is equivalent to

```
sh wg fred
```

This allows shell procedures and programs to be used interchangeably. In either case a new process is created to run the command.

As well as providing names for the positional parameters, the number of positional parameters in the call is available as **$#**. The name of the file being executed is available as **$0**.

A special shell parameter $* is used to substitute for all positional parameters except **$0**. A typical use of this is to provide some default arguments, as in,

```
nroff -T450 -ms $*
```

which simply prepends some arguments to those already given.

2.1 Control flow - for

A frequent use of shell procedures is to loop through the arguments (**$1, $2, ...**) executing commands once for each argument. An example of such a procedure is *tel* that searches the file **/usr/lib/telnos** that contains lines of the form

```
...
fred mh0123
bert mh0789
...
```

The text of *tel* is

```
for i
do grep $i /usr/lib/telnos; done
```

The command

```
tel fred
```

prints those lines in **/usr/lib/telnos** that contain the string *fred*.

```
tel fred bert
```

prints those lines containing *fred* followed by those for *bert.*

The **for** loop notation is recognized by the shell and has the general form

```
for name in w1 w2 ...
do command-list
done
```

A *command-list* is a sequence of one or more simple commands separated or terminated by a newline or semicolon. Furthermore, reserved words like **do** and **done** are only recognized following a newline or semicolon. *name* is a shell variable that is set to the words *w1 w2 ...* in turn each time the *command-list* following **do** is executed. If **in** *w1 w2 ...* is omitted then the loop is executed once for each positional parameter; that is, **in** $* is assumed.

Another example of the use of the **for** loop is the *create* command whose text is

```
for i do >$i; done
```

The command

```
create alpha beta
```

ensures that two empty files *alpha* and *beta* exist and are empty. The notation *>file* may be used on its own to create or clear the contents of a file. Notice also that a semicolon (or newline) is required before **done.**

2.2 Control flow - case

A multiple way branch is provided for by the **case** notation. For example,

```
case $# in
    1) cat >>$1 ;;
    2) cat >>$2 <$1 ;;
    *) echo 'usage: append [ from ] to' ;;
esac
```

is an *append* command. When called with one argument as

```
append file
```

$# is the string *1* and the standard input is copied onto the end of *file* using the *cat* command.

```
append file1 file2
```

appends the contents of *file1* onto *file2.* If the number of arguments supplied to *append* is other than 1 or 2 then a message is printed indicating proper usage.

The general form of the **case** command is

```
case word in
    pattern) command-list;;
    ...
esac
```

The shell attempts to match *word* with each *pattern,* in the order in which the patterns appear. If a match is found the associated *command-list* is executed and execution of the **case** is complete. Since * is the pattern that matches any string it can be used for the default case.

A word of caution: no check is made to ensure that only one pattern matches the case argument. The first match found defines the set of commands to be executed. In the example below the commands following the second * will never be executed.

```
case $# in
     *) ... ;;
     *) ... ;;
esac
```

Another example of the use of the **case** construction is to distinguish between different forms of an argument. The following example is a fragment of a *cc* command.

```
for i
do case $i in
   -[ocs])      ... ;;
   -*) echo 'unknown flag $i' ;;
   *.c) /lib/c0 $i ... ;;
   *)   echo 'unexpected argument $i' ;;
   esac
done
```

To allow the same commands to be associated with more than one pattern the **case** command provides for alternative patterns separated by a | . For example,

```
case $i in
     -x|-y)   ...
esac
```

is equivalent to

```
case $i in
     -[xy])   ...
esac
```

The usual quoting conventions apply so that

```
case $i in
     \?)      ...
```

will match the character ? .

2.3 Here documents

The shell procedure *tel* in section 2.1 uses the file **/usr/lib/telnos** to supply the data for *grep*. An alternative is to include this data within the shell procedure as a *here* document, as in,

```
for i
do grep $i <<!
   ...
   fred mh0123
   bert mh0789
   ...
!
done
```

In this example the shell takes the lines between <<! and ! as the standard input for *grep*. The string ! is arbitrary, the document being terminated by a line that consists of the string following << .

Parameters are substituted in the document before it is made available to *grep* as illustrated by the following procedure called *edg*.

```
ed $3 <<%
g/$1/s//$2/g
w
%
```

The call

```
edg string1 string2 file
```

is then equivalent to the command

```
ed file <<%
g/string1/s//string2/g
w
%
```

and changes all occurrences of *string1* in *file* to *string2*. Substitution can be prevented using \ to quote the special character $ as in

```
ed $3 <<+
1,\$s/$1/$2/g
w
+
```

(This version of *edg* is equivalent to the first except that *ed* will print a **?** if there are no occurrences of the string **$1.**) Substitution within a *here* document may be prevented entirely by quoting the terminating string, for example,

```
grep $i <<\#
...
#
```

The document is presented without modification to *grep*. If parameter substitution is not required in a *here* document this latter form is more efficient.

2.4 Shell variables

The shell provides string-valued variables. Variable names begin with a letter and consist of letters, digits and underscores. Variables may be given values by writing, for example,

```
user=fred box=m000 acct=mh0000
```

which assigns values to the variables **user, box** and **acct.** A variable may be set to the null string by saying, for example,

```
null=
```

The value of a variable is substituted by preceding its name with **$**; for example,

```
echo $user
```

will echo *fred.*

Variables may be used interactively to provide abbreviations for frequently used strings. For example,

```
b=/usr/fred/bin
mv pgm $b
```

will move the file *pgm* from the current directory to the directory **/usr/fred/bin.** A more general notation is available for parameter (or variable) substitution, as in,

```
echo ${user}
```

which is equivalent to

```
echo $user
```

and is used when the parameter name is followed by a letter or digit. For example,

```
tmp=/tmp/ps
ps a >${tmp}a
```

will direct the output of *ps* to the file **/tmp/psa,** whereas,

```
ps a >$tmpa
```

would cause the value of the variable **tmpa** to be substituted.

Except for $? the following are set initially by the shell. $? is set after executing each command.

$? The exit status (return code) of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully, otherwise a non-zero exit status is returned. Testing the value of return codes is dealt with later under **if** and **while** commands.

$# The number of positional parameters (in decimal). Used, for example, in the *append* command to check the number of parameters.

$$ The process number of this shell (in decimal). Since process numbers are unique among all existing processes, this string is frequently used to generate unique temporary file names. For example,

```
ps a >/tmp/ps$$
...
rm /tmp/ps$$
```

$! The process number of the last process run in the background (in decimal).

$- The current shell flags, such as **-x** and **-v**.

Some variables have a special meaning to the shell and should be avoided for general use.

$MAIL When used interactively the shell looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since it was last looked at the shell prints the message *you have mail* before prompting for the next command. This variable is typically set in the file **.profile,** in the user's login directory. For example,

```
MAIL=/usr/mail/fred
```

$HOME The default argument for the *cd* command. The current directory is used to resolve file name references that do not begin with a **/**, and is changed using the *cd* command. For example,

```
cd /usr/fred/bin
```

makes the current directory **/usr/fred/bin.**

```
cat wn
```

will print on the terminal the file *wn* in this directory. The command *cd* with no argument is equivalent to

```
cd $HOME
```

This variable is also typically set in the the user's login profile.

$PATH A list of directories that contain commands (the *search path*). Each time a command is executed by the shell a list of directories is searched for an executable

file. If $PATH is not set then the current directory, **/bin**, and **/usr/bin** are searched by default. Otherwise $PATH consists of directory names separated by **:**. For example,

PATH=:/usr/fred/bin:/bin:/usr/bin

specifies that the current directory (the null string before the first **:**), **/usr/fred/bin, /bin** and **/usr/bin** are to be searched in that order. In this way individual users can have their own 'private' commands that are accessible independently of the current directory. If the command name contains a / then this directory search is not used; a single attempt is made to execute the command.

$PS1 The primary shell prompt string, by default, '$ '.

$PS2 The shell prompt when further input is needed, by default, '> '.

$IFS The set of characters used by *blank interpretation* (see section 3.4).

2.5 The test command

The *test* command, although not part of the shell, is intended for use by shell programs. For example,

```
test – f file
```

returns zero exit status if *file* exists and non-zero exit status otherwise. In general *test* evaluates a predicate and returns the result as its exit status. Some of the more frequently used *test* arguments are given here, see *test* (1) for a complete specification.

test s	true if the argument *s* is not the null string
test – f file	true if *file* exists
test – r file	true if *file* is readable
test – w file	true if *file* is writable
test – d file	true if *file* is a directory

2.6 Control flow - while

The actions of the **for** loop and the **case** branch are determined by data available to the shell. A **while** or **until** loop and an **if then else** branch are also provided whose actions are determined by the exit status returned by commands. A **while** loop has the general form

while *command-list*$_1$
do *command-list*$_2$
done

The value tested by the **while** command is the exit status of the last simple command following **while.** Each time round the loop *command-list*$_1$ is executed; if a zero exit status is returned then *command-list*$_2$ is executed; otherwise, the loop terminates. For example,

```
while test $1
do ...
   shift
done
```

is equivalent to

```
for i
do ...
done
```

shift is a shell command that renames the positional parameters **$2, $3, ...** as **$1, $2, ...** and loses **$1.**

Another kind of use for the **while/until** loop is to wait until some external event occurs and then run some commands. In an **until** loop the termination condition is reversed. For example,

```
until test – f file
do sleep 300; done
commands
```

will loop until *file* exists. Each time round the loop it waits for 5 minutes before trying again. (Presumably another process will eventually create the file.)

2.7 Control flow - if

Also available is a general conditional branch of the form,

```
if command-list
then    command-list
else    command-list
fi
```

that tests the value returned by the last simple command following **if.**

The **if** command may be used in conjunction with the *test* command to test for the existence of a file as in

```
if test – f file
then    process file
else    do something else
fi
```

An example of the use of **if, case** and **for** constructions is given in section 2.10.

A multiple test **if** command of the form

```
if ...
then    ...
else    if ...
        then    ...
        else    if ...
                ...
                fi
        fi
fi
```

may be written using an extension of the **if** notation as,

```
if ...
then    ...
elif    ...
then    ...
elif    ...
...
fi
```

The following example is the *touch* command which changes the 'last modified' time for a list of files. The command may be used in conjunction with *make* (1) to force recompilation of a list of files.

```
flag=
for i
do case $i in
   -c) flag=N ;;
   *)  if test -f $i
       then    ln $i junk$$; rm junk$$
       elif test $flag
       then    echo file \$i\' does not exist
       else    >$i
       fi
    esac
done
```

The **-c** flag is used in this command to force subsequent files to be created if they do not already exist. Otherwise, if the file does not exist, an error message is printed. The shell variable *flag* is set to some non-null string if the **-c** argument is encountered. The commands

```
ln ...; rm ...
```

make a link to the file and then remove it thus causing the last modified date to be updated.

The sequence

```
if command1
then    command2
fi
```

may be written

```
command1 && command2
```

Conversely,

```
command1 || command2
```

executes *command2* only if *command1* fails. In each case the value returned is that of the last simple command executed.

2.8 Command grouping

Commands may be grouped in two ways,

```
{ command-list ; }
```

and

```
( command-list )
```

In the first *command-list* is simply executed. The second form executes *command-list* as a separate process. For example,

```
(cd x; rm junk )
```

executes *rm junk* in the directory **x** without changing the current directory of the invoking shell.

The commands

```
cd x; rm junk
```

have the same effect but leave the invoking shell in the directory **x.**

2.9 Debugging shell procedures

The shell provides two tracing mechanisms to help when debugging shell procedures. The first is invoked within the procedure as

```
set -v
```

(**v** for verbose) and causes lines of the procedure to be printed as they are read. It is useful to help isolate syntax errors. It may be invoked without modifying the procedure by saying

```
sh -v proc ...
```

where *proc* is the name of the shell procedure. This flag may be used in conjunction with the **-n** flag which prevents execution of subsequent commands. (Note that saying *set -n* at a terminal will render the terminal useless until an end-of-file is typed.)

The command

```
set -x
```

will produce an execution trace. Following parameter substitution each command is printed as it is executed. (Try these at the terminal to see what effect they have.) Both flags may be turned off by saying

```
set -
```

and the current setting of the shell flags is available as $- .

2.10 The man command

The *man* command prints sections of the ROS manual. It is called, for example, as

```
man sh
man -t ed
man 2 fork
```

In the first line, the manual section for *sh* is printed. Since no section is specified, section 1 is used. The second example will typeset (**-t** option) the manual section for *ed*. The last prints the *fork* manual page from section 2.

```
cd /usr/man

: 'colon is the comment command'
: 'default is nroff ($N), section 1 ($s)'
N=n s=1

for i
do case $i in
   [1-9]*)      s=$i ;;
   -t) N=t ;;
   -n) N=n ;;
   -*) echo unknown flag \$i\' ;;
   *)  if test -f man$s/$i.$s
       then    ${N}roff man0/${N}aa man$s/$i.$s
       else    : 'look through all manual sections'
               found=no
               for j in 1 2 3 4 5 6 7 8 9
               do if test -f man$j/$i.$j
                  then man $j $i
                       found=yes
                  fi
               done
               case $found in
                    no) echo '$i: manual page not found'
               esac
       fi
   esac
done
```

Figure 1. A version of the man command

3.0 Keyword parameters

Shell variables may be given values by assignment or when a shell procedure is invoked. An argument to a shell procedure of the form *name=value* that precedes the command name causes *value* to be assigned to *name* before execution of the procedure begins. The value of *name* in the invoking shell is not affected. For example,

```
user=fred command
```

will execute *command* with **user** set to *fred*. The **-k** flag causes arguments of the form *name=value* to be interpreted in this way anywhere in the argument list. Such *names* are sometimes called keyword parameters. If any arguments remain they are available as positional parameters **$1, $2,**

The *set* command may also be used to set positional parameters from within a procedure. For example,

```
set - *
```

will set **$1** to the first file name in the current directory, **$2** to the next, and so on. Note that the first argument, **-**, ensures correct treatment when the first file name begins with a **-** .

3.1 Parameter transmission

When a shell procedure is invoked both positional and keyword parameters may be supplied with the call. Keyword parameters are also made available implicitly to a shell procedure by specifying in advance that such parameters are to be exported. For example,

```
export user box
```

marks the variables **user** and **box** for export. When a shell procedure is invoked copies are made of all exportable variables for use within the invoked procedure. Modification of such variables within the procedure does not affect the values in the invoking shell. It is generally true of a shell procedure that it may not modify the state of its caller without explicit request on the part of the caller. (Shared file descriptors are an exception to this rule.)

Names whose value is intended to remain constant may be declared *readonly*. The form of this command is the same as that of the *export* command,

```
readonly name ...
```

Subsequent attempts to set readonly variables are illegal.

3.2 Parameter substitution

If a shell parameter is not set then the null string is substituted for it. For example, if the variable **d** is not set

```
echo $d
```

or

```
echo ${d}
```

will echo nothing. A default string may be given as in

```
echo ${d-.}
```

which will echo the value of the variable **d** if it is set and '.' otherwise. The default string is evaluated using the usual quoting conventions so that

```
echo ${d-'*'}
```

will echo * if the variable **d** is not set. Similarly

```
echo ${d-$1}
```

will echo the value of **d** if it is set and the value (if any) of **$1** otherwise. A variable may be assigned a default value using the notation

```
echo ${d=.}
```

which substitutes the same string as

```
echo ${d-.}
```

and if **d** were not previously set then it will be set to the string '.'. (The notation ${...=...} is not available for positional parameters.)

If there is no sensible default then the notation

```
echo ${d?message}
```

will echo the value of the variable **d** if it has one, otherwise *message* is printed by the shell and execution of the shell procedure is abandoned. If *message* is absent then a standard message is printed. A shell procedure that requires some parameters to be set might start as follows.

```
: ${user?} ${acct?} ${bin?}
...
```

Colon (:) is a command that is built in to the shell and does nothing once its arguments have been evaluated. If any of the variables **user, acct** or **bin** are not set then the shell will abandon execution of the procedure.

3.3 Command substitution

The standard output from a command can be substituted in a similar way to parameters. The command *pwd* prints on its standard output the name of the current directory. For example, if the current directory is **/usr/fred/bin** then the command

```
d=`pwd`
```

is equivalent to

```
d=/usr/fred/bin
```

The entire string between grave accents (`...`) is taken as the command to be executed and is replaced with the output from the command. The command is written using the usual quoting conventions except that a ` must be escaped using a \. For example,

```
ls `echo "$1"`
```

is equivalent to

```
ls $1
```

Command substitution occurs in all contexts where parameter substitution occurs (including "*here*" documents) and the treatment of the resulting text is the same in both cases. This mechanism allows string processing commands to be used within shell procedures. An example of such a command is *basename* which removes a specified suffix from a string. For example,

```
basename main.c .c
```

will print the string *main*. Its use is illustrated by the following fragment from a *cc* command.

```
case $A in
    ...
    *.c)    B=`basename $A .c`
    ...
esac
```

that sets **B** to the part of **$A** with the suffix **.c** stripped.

Here are some composite examples.

- **for i in `ls – t`; do ...**
  The variable **i** is set to the names of files in time order, most recent first.
- **set `date`; echo $6 $2 $3, $4**
  will print, e.g., *1977 Nov 1, 23:59:59*

3.4 Evaluation and quoting

The shell is a macro processor that provides parameter substitution, command substitution and file name generation for the arguments to commands. This section discusses the order in which these evaluations occur and the effects of the various quoting mechanisms.

Commands are parsed initially according to the grammar given in appendix A. Before a command is executed the following substitutions occur.

- parameter substitution, e.g. **$user**
- command substitution, e.g. **`pwd`**

  Only one evaluation occurs so that if, for example, the value of the variable **X** is the string *$y* then

  ```
  echo $X
  ```

  will echo *$y*.
- blank interpretation

  Following the above substitutions the resulting characters are broken into non-blank words (*blank interpretation*). For this purpose 'blanks' are the characters of the string $IFS. By default, this string consists of blank, tab and newline. The null string is not regarded as a word unless it is quoted. For example,

  ```
  echo ''
  ```

  will pass on the null string as the first argument to *echo*, whereas

  ```
  echo $null
  ```

  will call *echo* with no arguments if the variable **null** is not set or set to the null string.
- file name generation

  Each word is then scanned for the file pattern characters *, ? and [...] and an alphabetical list of file names is generated to replace the word. Each such file name is a separate argument.

The evaluations just described also occur in the list of words associated with a **for** loop. Only substitution occurs in the *word* used for a **case** branch.

As well as the quoting mechanisms described earlier using \ and '...' a third quoting mechanism is provided using double quotes. Within double quotes parameter and command substitution occurs but file name generation and the interpretation of blanks does not. The following characters have a special meaning within double quotes and may be quoted using \.

$	parameter substitution
`	command substitution
"	ends the quoted string
\	quotes the special characters $ ` " \

For example,

```
echo "$x"
```

will pass the value of the variable **x** as a single argument to *echo*. Similarly,

```
echo "$*"
```

will pass the positional parameters as a single argument and is equivalent to

```
echo "$1 $2 ..."
```

The notation $@ is the same as $* except when it is quoted.

```
echo "$@"
```

will pass the positional parameters, unevaluated, to *echo* and is equivalent to

```
echo "$1" "$2" ...
```

The following table gives, for each quoting mechanism, the shell metacharacters that are evaluated.

	metacharacter					
	\	$	*	`	"	'
'	n	n	n	n	n	t
`	y	n	n	t	n	n
"	y	y	n	y	t	n

t	terminator
y	interpreted
n	not interpreted

Figure 2. Quoting mechanisms

In cases where more than one evaluation of a string is required the built-in command *eval* may be used. For example, if the variable **X** has the value $*y*, and if **y** has the value *pqr* then

```
eval echo $X
```

will echo the string *pqr*.

In general the *eval* command evaluates its arguments (as do all commands) and treats the result as input to the shell. The input is read and the resulting command(s) executed. For example,

```
wg='eval who|grep'
$wg fred
```

is equivalent to

```
who|grep fred
```

In this example, *eval* is required since there is no interpretation of metacharacters, such as |, **following substitution.**

3.5 Error handling

The treatment of errors detected by the shell depends on the type of error and on whether the shell is being used interactively. An interactive shell is one whose input and output are connected to a terminal (as determined by *gtty* (2)). A shell invoked with the **-i** flag is also interactive.

Execution of a command (see also 3.7) may fail for any of the following reasons.

- Input output redirection may fail. For example, if a file does not exist or cannot be created.

- The command itself does not exist or cannot be executed.
- The command terminates abnormally, for example, with a "bus error" or "memory fault". See the signal(2) page of the ROS Reference Manual for a complete list of ROS signals.
- The command terminates normally but returns a non-zero exit status.

In all of these cases the shell will go on to execute the next command. Except for the last case an error message will be printed by the shell. All remaining errors cause the shell to exit from a command procedure. An interactive shell will return to read another command from the terminal. Such errors include the following.

- Syntax errors. e.g., if ... then ... done
- A signal such as interrupt. The shell waits for the current command, if any, to finish execution and then either exits or returns to the terminal.
- Failure of any of the built-in commands such as *cd*.

The shell flag – **e** causes the shell to terminate if any error is detected, such as:

1	hangup
2	interrupt
3	quit
4	illegal instruction
5	trace trap
12	bad argument to system call
13	write on a pipe with no one to read it
14	alarm clock

Figure 3. SAMPLE list of ROS signals.
See signal(2) for a full list.

Those signals marked with an asterisk produce a core dump if not caught. However, the shell itself ignores quit which is the only external signal that can cause a dump. The signals in this list of potential interest to shell programs are 1, 2, 3, 14 and 15.

3.6 Fault handling

Shell procedures normally terminate when an interrupt is received from the terminal. The *trap* command is used if some cleaning up is required, such as removing temporary files. For example,

```
trap 'rm /tmp/ps$$; exit' 2
```

sets a trap for signal 2 (terminal interrupt), and if this signal is received will execute the commands

```
rm /tmp/ps$$; exit
```

exit is another built-in command that terminates execution of a shell procedure. The *exit* is required; otherwise, after the trap has been taken, the shell will resume executing the procedure at the place where it was interrupted.

Signals can be handled in one of three ways. They can be ignored, in which case the signal is never sent to the process. They can be caught, in which case the process must decide what action to take when the signal is received. Lastly, they can be left to cause termination of the process without it having to take any further action. If a signal is being ignored on entry to the shell procedure, for example, by invoking it in the background (see 3.7) then *trap* commands (and the signal) are ignored.

The use of *trap* is illustrated by this modified version of the *touch* command (Figure 4). The cleanup action is to remove the file **junk$$**.

```
flag=
trap 'rm -f junk$$; exit' 1 2 3 15
for i
do case $i in
   -c) flag=N ;;
   *)  if test -f $i
       then    ln $i junk$$; rm junk$$
       elif test $flag
       then    echo file \'$i\' does not exist
       else    >$i
       fi
   esac
done
```

Figure 4. The touch command

The *trap* command appears before the creation of the temporary file; otherwise it would be possible for the process to die without removing the file.

Since there is no signal 0, it is used by the shell to indicate the commands to be executed on exit from the shell procedure.

A procedure may, itself, elect to ignore signals by specifying the null string as the argument to trap. The following fragment is taken from the *nohup* command.

```
trap '' 1 2 3 15
```

which causes *hangup, interrupt, quit* and *kill* to be ignored both by the procedure and by invoked commands.

Traps may be reset by saying

```
trap 2 3
```

which resets the traps for signals 2 and 3 to their default values. A list of the current values of traps may be obtained by writing

```
trap
```

The procedure *scan* (Figure 5) is an example of the use of *trap* where there is no exit in the trap command. *scan* takes each directory in the current directory, prompts with its name, and then executes commands typed at the terminal until an end of file or an interrupt is received. Interrupts are ignored while executing the requested commands but cause termination when *scan* is waiting for input.

```
d=`pwd`
for i in *
do if test -d $d/$i
   then cd $d/$i
        while echo "$i:"
              trap exit 2
              read x
        do trap : 2; eval $x; done
   fi
done
```

Figure 5. The scan command

read x is a built-in command that reads one line from the standard input and places the result in

the variable **x**. It returns a non-zero exit status if either an end-of-file is read or an interrupt is received.

3.7 Command execution

To run a command (other than a built-in) the shell first creates a new process using the system call *fork*. The execution environment for the command includes input, output and the states of signals, and is established in the child process before the command is executed. The built-in command *exec* is used in the rare cases when no fork is required and simply replaces the shell with a new command. For example, a simple version of the *nohup* command looks like

```
trap "" 1 2 3 15
exec $
```

The *trap* turns off the signals specified so that they are ignored by subsequently created commands and *exec* replaces the shell by the command specified.

Most forms of input output redirection have already been described. In the following *word* is only subject to parameter and command substitution. No file name generation or blank interpretation takes place so that, for example,

```
echo hello > *.c
```

writes the word "hello" into a file whose name is ***.c** .

Input/output specifications are evaluated left to right as they appear in the command.

> *word*	The standard output (file descriptor 1) is sent to the file *word* which is created if it does not already exist.
word	The standard output is sent to file *word*. If the file exists then output is appended (by seeking to the end); otherwise the file is created.
< *word*	The standard input (file descriptor 0) is taken from the file *word*.
word	The standard input is taken from the lines of shell input that follow up to but not including a line consisting only of *word*. If *word* is quoted then no interpretation of the document occurs. If *word* is not quoted, parameter and command substitution occur and \ is used to quote the characters \ $ ` and the first character of *word*. In the latter case **\newline** is ignored (c.f. quoted strings).
>& *digit*	The file descriptor *digit* is duplicated using the system call *dup* (2) and the result is used as the standard output.
<& *digit*	The standard input is duplicated from file descriptor *digit*.
<&–	The standard input is closed.
>&–	The standard output is closed.

Any of the above may be preceded by a digit in which case the file descriptor created is that specified by the digit instead of the default 0 or 1. For example,

```
2>file
```

runs a command with message output (file descriptor 2) directed to *file*.

```
2>&1
```

runs a command with its standard output and message output merged. (Strictly speaking file descriptor 2 is created by duplicating file descriptor 1 but the effect is usually to merge the two streams.)

The environment for a command run in the background such as

```
list .c  lpr &
```

is modified in two ways. Firstly, the default standard input for such a command is the empty

file **/dev/null**. This prevents two processes (the shell and the command), which are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case. For example,

```
ed file &
```

would allow both the editor and the shell to read from the same input at the same time.

The other modification to the environment of a background command is to turn off the QUIT and INTERRUPT signals so that they are ignored by the command. This allows these signals to be used at the terminal without causing background commands to terminate. For this reason, the ROS convention for a signal is that if it is set to 1 (ignored) then it is never changed even for a short time. Note that the shell command *trap* has no effect for an ignored signal.

3.8 Invoking the shell

The following flags are interpreted by the shell when it is invoked. If the first character of argument zero is a minus, then commands are read from the file **.profile**.

– c *string*
: If the **– c** flag is present then commands are read from *string*.

– s If the **– s** flag is present or if no arguments remain then commands are read from the standard input. Shell output is written to file descriptor 2.

– i If the **– i** flag is present or if the shell input and output are attached to a terminal (as told by *gtty*) then this shell is *interactive*. In this case TERMINATE is ignored (so that **kill 0** does not kill an interactive shell) and INTERRUPT is caught and ignored (so that **wait** is interruptable). In all cases QUIT is ignored by the shell.

$LIST$

Appendix A - Grammar

item: *word*
input-output
name = *value*

simple-command: *item*
simple-command item

command: *simple-command*
(*command-list*)
{ *command-list* }
for *name* **do** *command-list* **done**
for *name* **in** *word* **do** *command-list* **done**
while *command-list* **do** *command-list* **done**
until *command-list* **do** *command-list* **done**
case *word* **in** *case-part* **esac**
if *command-list* **then** *command-list else-part* **fi**

pipeline: *command*
pipeline command

andor: *pipeline*
andor && *pipeline*
andor pipeline

command-list: *andor*
command-list ;
command-list &
command-list ; *andor*
command-list & *andor*

input-output: > *file*
< *file*
word
word

file: *word*
& *digit*
& –

case-part: *pattern*) *command-list* ;;

pattern: *word*
pattern word

else-part: **elif** *command-list* **then** *command-list else-part*
else *command-list*
empty

empty:

word: a sequence of non-blank characters

name: a sequence of letters, digits or underscores starting with a letter

digit: **0 1 2 3 4 5 6 7 8 9**

Appendix B - Meta-characters and Reserved Words

a) syntactic

|	pipe symbol
&&	'andf' symbol
||	'orf' symbol
;	command separator
;;	case delimiter
&	background commands
( )	command grouping
<	input redirection
<<	input from a here document
>	output creation
>>	output append

b) patterns

*	match any character(s) including none
?	match any single character
[...]	match any of the enclosed characters

c) substitution

${...}	substitute shell variable
`...`	substitute command output

d) quoting

\	quote the next character
'...'	quote the enclosed characters except for '
"..."	quote the enclosed characters except for $ ` \ "

e) reserved words

if then else elif fi
case in esac
for while until do done
{ }

Ridge Debug

PREFACE

This manual describes the Ridge Debug process. The first section offers a short overall description of the process; the second section describes the commands; and the third section contains a tutorial.

Considerable familiarity with programming is assumed on the reader's part; however, the tutorial contains specific information on Ridge compilation and editing.

The reader is referred to the Ridge "Pascal Reference Manual" and "FORTRAN Service Note 1" for details on the Ridge compilers for these languages.

TABLE OF CONTENTS

This manual used to be part 9006.
It is now incorporated into the ROS Programmer's Guide (9050).

-blank-

SECTION 1

INTRODUCTION

OVERVIEW

The Debug process provides on-line debugging of runtime errors for both high level and assembly language programs. Debug capabilities include 16 specifiable breakpoints, examination of registers and data and code memory, and the ability to modify data. A help facility lists the debug commands.

In the Ridge Operating System, processes are software entities that perform computational tasks and which interact with other processes to provide application-level functions or system services. The Debug process is one of many processes managed by the User Monitor associated with each active user. The Debug process can be used interactively to examine and modify the state of all user processes associated with a User Monitor.

When user processes are active, the Debug process remains suspended. If a user process encounters an illegal instruction, invalid memory reference, or other exception, the user process is suspended and the Debug process is activated (indicated by the Debug prompt, ":").

The Debug process can also be activated from a command interpreter as part of starting a user program. This is done by typing "debug" followed by the program name and any arguments.

Or, if a program is executing, the Debug process can be activated by typing Control-\.

SECTION 2

COMMAND SET

INTRODUCTION

This section defines the debug command set (summarized in Table 1). In the debug process, values are output in hexadecimal and must be input in hexadecimal. The user must wait for the debug prompt, ":", before entering a command, and, after entering a command, must press RETURN.

Syntax Notation

In general, the first letter or two of a command's name is the mnemonic which must be used to invoke the command. Command mnemonics can be written in either upper or lower case letters. Required parameters are surrounded by "<" and ">". Options are surrounded by "[" and "]".

Table 1. Command Summary

SYNTAX	COMMAND NAME AND DESCRIPTION
B [address]	BREAK. Set a breakpoint or list all existing breakpoints.
CB <address>	CLEAR BREAKPOINT. Clears specified breakpoint.
D <address> [count]	DISPLAY DATA. Displays data at specified memory address(es).
DC <address> [count]	DISPLAY CODE. Displays code at specified memory address(es).

(continued on next page)

Table 1. Command Summary, Continued

SYNTAX	COMMAND NAME AND DESCRIPTION
DR	DISPLAY REGISTERS. Displays contents of registers R0 through R15 and PC.
E	EXIT. Exit from debug process.
H	HELP. Lists all debug commands.
K [pID]	KILL. Terminates current (or specified) process.
M <address>	MODIFY. Change or verify sequential data bytes.
P [pID]	PROCESS SET. Sets specified process as current, or lists all processes.

BREAKPOINT Command

Syntax: B [address]

This command sets a breakpoint at a virtual code location specified (in hex). When the designated address is reached, the debug process is activated and a banner with the current program counter (PC) is displayed. After a breakpoint has been reached, it is cleared.

A maximum of 16 breakpoints may be set. If an attempt is made to set more than 16 breakpoints, "break full" is displayed. If no address is specified, the Breakpoint command lists all currently set breakpoints.

CLEAR BREAKPOINT Command

Syntax: CB <address>

This command allows a specified breakpoint to be eliminated. The address must be in hex.

DISPLAY DATA Command

Syntax: D <address> [count]

This command allows the user to view the contents of data memory in hexadecimal and ASCII format. When only a memory address is specified, 16 bytes of data are displayed. When a count is also specified (in hex), that many bytes of memory are displayed starting from the initial address. For example:

```
:D 10 14
```

will cause 14H bytes of data, starting at location 10H, to be displayed. ASCII characters are also displayed to the right of each line; "." represents characters that are nondisplayable.

DISPLAY CODE Command

Syntax: DC <address> [count]

This command allows the user to see the contents of code memory in hexadecimal format. The assembly language for the code is also shown, as are ASCII characters. Nondisplayable characters are represented by ".". When only a memory address is specified, the single instruction (2 to 6 bytes) that starts at that address is displayed. When a count is also specified (in hex), that many bytes are displayed starting from the initial address.

DISPLAY REGISTERS Command

Syntax: DR

This command displays the status of the 16 registers for the current process. A banner shows the process identification number (pID) and the current program counter (PC). The next line provides a representation of the registers R0 through R7, from left to right; the succeeding line shows registers R8 through R15.

EXIT Command

Syntax: E

This command causes the debug process to be suspended and program execution to resume. It is used interactively with CTRL-\ to start and stop the debug process: CTRL-\ suspends program execution; any debug command(s) can then be used; Exit causes the resumption of program execution.

HELP Command

Syntax: H

HELP displays an alphabetical listing, including syntax, of the debug commands.

KILL Command

Syntax: K [pID]

This command kills the current (or specified) process and returns control to the Shell command interpreter if there are no more processes being debugged.

MODIFY Command

Syntax: M <address>

This command allows the user to alter data at a memory location that is specified in hexadecimal. The command displays the first byte of data from the starting address on the screen. The user can then either enter new values, or retain the existing values by pressing RETURN; in either case the debug process displays the next byte on the screen for the user to retain or change. This sequence continues indefinitely until the user breaks out of it by striking a nonhexadecimal character.

PROCESS SET Command

Syntax: P [pID]

This command sets the specified process pID as the current process, or, if no pID is given, displays all process IDs of processes being debugged.

SECTION 3

TUTORIAL

INTRODUCTION

This section illustrates the basics of compiling and debugging a sample Pascal program, "prime.s", that computes the first 1000 prime numbers. Once written, the following command file compiles, assembles, and links such a source program:

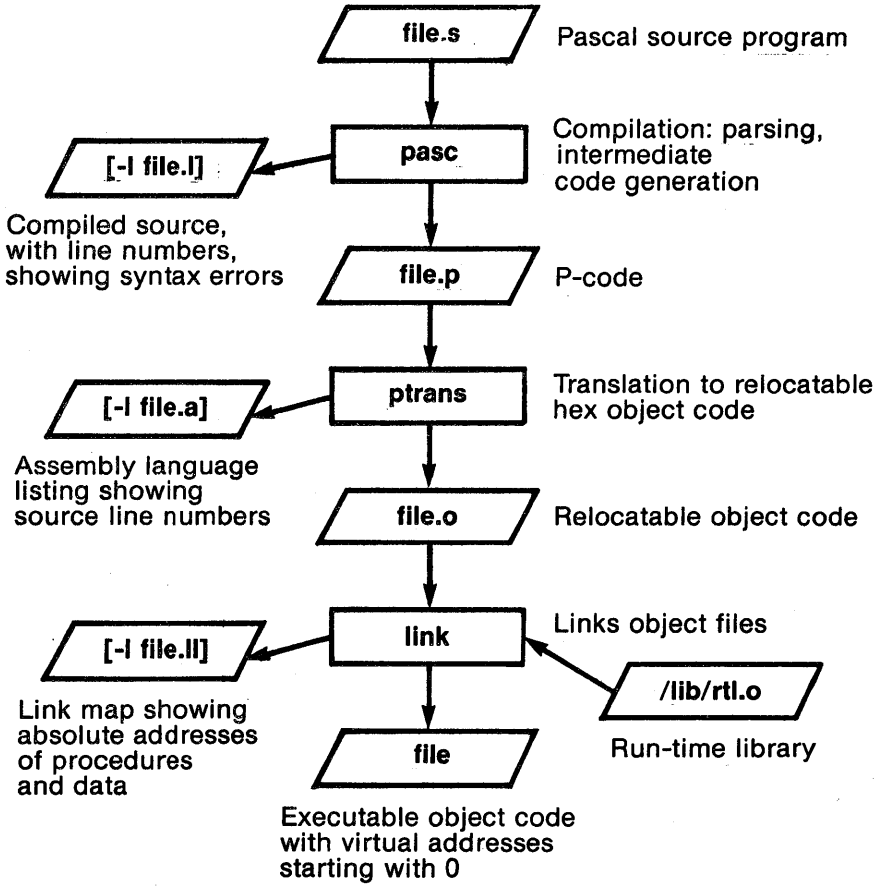
```
pasc -l prime.l prime.s
ptrans -l prime.a prime.p
link -l prime.map prime.o /lib/rtl.o
```

In the first line, the Pascal compiler ("pasc") compiles from the source code ("prime.s") two output files: one is an intermediate P-code file ("prime.p"); the second is an optional file invoked with "-l" and named "prime.l". "prime.l" is a source code listing that shows line numbers, syntax error messages, and the locations of program and data code (P/D LC).

In the second line, the P-code translator ("ptrans") translates and assembles the "prime.p" P-code file into a relocatable object code file ("prime.o"). Another optional file invoked by "-l", named "prime.a", provides an assembly language listing that shows source code line numbers.

In the third line, the linker ("link") combines the "prime.o" object file and the runtime library object file ("/lib/rtl.o") to produce the final executable code file, "prime". An optional file has also been invoked by "-l" to create a link map file ("prime.map") that shows the absolute addresses of procedures and data.

Figure 1 shows a schematic overview of this compiling, assembling, and linking process. The listings for "prime.l", "prime.a", and "prime.map" appear at the end of this tutorial.



NOTE: [ ] indicate an option. In this illustration, the optional link map file, referred to in text as "prime.map", is called "file.ll".

Figure 1. Compiling a Pascal Program

DEBUGGING A PROGRAM

To begin the debug process as part of starting "prime", type after the Shell prompt ($):

```
$ debug prime
```

A message is displayed on the screen, followed on the next line by ":", the debug prompt:

```
process 00001F60 suspended
:
```

The process identification number indicates the current process ID number of the suspended program.

To examine the registers before program execution (the PC is 0), use the Display Register command:

```
:dr
```

The contents of the registers and the PC are displayed, showing the initial values of the registers:

```
PID: ØØØØ1F6Ø  PC: ØØØØØØØØ
RØ        R1        R2        R3        R4        R5        R6        R7
ØØØØØØØØ  ØØØØØØØØ  ØØØØØØØØ  ØØØØØØØØ  FFF77FF7  FFF77FF7  FFF77FF7  FFF77FF7
R8        R9        R1Ø       R11       R12       R13       R14       R15
ØØØØØØØØ  ØØØØØØØØ  ØØØØØØØØ  ØØØØØØØØ  ØØØØØØØØ  E5DF57FF  1BBF6EFF  FEB7B17F
```

To check that variables in a source code line, for example, line 22:

```
X := 1; INC := 4; LIM := 1; SQUARE := 9;
```

are being properly initialized and the correct values loaded into the registers, first examine the "prime.l" listing. The listing for this line is as follows:

```
LINE #  P/D LC  LVL
------  ------  ---
    22      23   1)     X := 1; INC := 4; LIM := 1; SQUARE := 9;
```

"LINE" refers to the source code line numbers; "LVL" indicates nesting level; the "P/D LC" value represents the memory location of program or data code in decimal (found in the P-code listing, which isn't included in this tutorial).

Turning to the "prime.a" listing, we find the assembler listing

shows the code for line 22 to be stored starting at memory location 64H. In this listing, the leftmost column shows the memory location; the next, the machine code; the last two the assembler source code:

```
SOURCE LINE 22
    00000064   1111          MOVEI   R1,1
    00000066   A6101008      STORE   R1,4104
    0000006A   1124          MOVEI   R2,4
    0000006C   A620100C      STORE   R2,4108
    00000070   1131          MOVEI   R3,1
    00000072   A6301010      STORE   R3,4112
    00000076   1149          MOVEI   R4,9
    00000078   A6401014      STORE   R4,4116
```

According to the assembler, the decimal values 1, 4, 1, and 9 are in fact being placed in registers R1, R2, R3, and R4, respectively. To examine these registers, breakpoints can be placed at appropriate locations--such as immediately before and after a MOVEI.

If it is desirable to first verify that the code looks exactly like the listing, use the Display Code command in conjunction with the first memory address by typing in:

```
:dc 64
```

This causes the following line to be displayed:

```
00000064: 1111              MOVEI   R1,1        '..'
```

The meaning of the columns is the same as that previously explained for the "prime.a" listing, except for the last column on the right which shows any ASCII characters; those that are nondisplayable are represented by a ".".

The assembler listing can be used to determine where to set breakpoints within a program; for a procedure or function, however, examining the "prime.map" listing is useful. "prime.map" shows the locations for all procedures and functions, including the runtime library routines. The lefthand column lists the routines in alphabetical order with their assigned virtual addresses; the righthand column is arranged by order of increasing memory location.

To set a breakpoint before the MOVEI (which should move 1 into register 1):

```
:b 64
```

Then, to set a breakpoint after the MOVEI to see the new status of register 1, type:

```
:b 66
```

Additional breakpoints (up to a total of 16) can be set to check whether the other registers are being corrently loaded. Each breakpoint must be set separately. To list all breakpoints that have been set, type:

```
:b
```

To examine the registers, leave the debugger and start the "prime" program, using the Exit command:

```
:e
```

When a breakpoint is encountered, a message is displayed on the screen:

```
break at 00000064
```

To examine the register contents at this first breakpoint location, use the Display Register command ("dr"). This causes the registers to be displayed:

```
PID: ØØØØ1F6Ø  PC: ØØØØØØ64
RØ       R1       R2       R3       R4       R5       R6       R7
ØØØØØØØ2 ØØØØØØ33 ØØØØØØØ1 ØØ7FFEØØ ØØ7FØØØØ ØØØØØ1FC ØØØ7FFEØ ØØØØØ1FF
R8       R9       R1Ø      R11      R12      R13      R14      R15
ØØØØØØ61 ØØØØØØ77 ØØ2958ØØ ØØØØ4C7A ØØØØ115Ø ØØØØØØØ2 ØØ1115Ø  ØØØØØØØØ
```

Notice the 33 in register 1.

Use the Exit command again to go to the next breakpoint, and display the registers using the Display Register command:

```
PID: ØØØØ1F6Ø  PC:ØØØØØØ66
RØ       R1       R2       R3       R4       R5       R6       R7
ØØØØØØØ2 ØØØØØØØ1 ØØØØØØØ1 ØØ7FFEØØ ØØ7FØØØØ ØØØØØ1FC ØØØ7FFEØ ØØØØØ1FF
R8       R9       R1Ø      R11      R12      R13      R14      R15
ØØØØØØ61 ØØØØØØ77 ØØ2958ØØ ØØØØ4C7A ØØØØ115Ø ØØØØØØØ2 ØØ1115Ø  ØØØØØØØØ
```

Register 1 now contains 1.

The instruction at location 66H stores register 1 into data address 1008H (4104 decimal), which is the variable "X". To verify that the variable is actually being initialized, set a breakpoint at the next instruction, and then exit the debugger:

```
:b 6A
```

followed by:

```
:e
```

After the breakpoint is encountered, display the data location using the Display Data command:

```
:d 1008
```

The debugger displays the 16 bytes starting at location 1008H:

```
00001008: 00 00 00 01 7F 00 00 00 00 00 00 00 00 7F FF FF FF '................'
```

Notice that the 32-bit value 1 has been set into the word (4 bytes) at location 1008H. Further breakpoints could be set and memory examined with the Display Data command or modified with the Modify Data command.

Once the bugs in a program have been determined, the program's process can be terminated by the Kill command:

```
:k
```

Control of the terminal is then returned to the Shell.

PRIME.L LISTING

```
LINE #  P/D LC  LVL  < Ridge Pascal Compiler, Version of 29-Oct-82 >
------  ------  ---

     1           ) {$d-,A+}
     2           ) (* Program "primes" computes the first 1000 prime numbers, and *)
     3           ) (* writes them in a table with 20 numbers per line.  This takes 1347 *)
     4           ) (* msec. on the CDC 6400 (1061 msec. without the range checking)     *)
     5           )
     6           ) (* Modified to 10 numbers per line. *)
     7           )
     8           ) PROGRAM PRIMES(OUTPUT,input);
     9           ) CONST N = 1000;  N1 = 33;  (*N1 = SQUARE ROOT OF N*)
    10           ) VAR I,K,X,INC,LIM,SQUARE,L,HI,LOW, q, iterations: INTEGER;
    11    4140   1)    PRIM: BOOLEAN;
    12    4144   1)    P,V: ARRAY[1..N1] OF INTEGER;
    13    4408   1)
    14    4408   1)     SHITIME ,SLOWTIME, EHITIME, ELOWTIME: INTEGER;
    15    4424   1) function runtime : integer ; external;
    16           ) BEGIN
    17           )
    18           ) iterations := 50;
    19      12   1)
    20      12   1) for q := 1 to iterations do begin
    21      21   1)   {WRITE(2:6, 3:6);} L := 2;
    22      23   1)       X := 1;  INC := 4;  LIM := 1;  SQUARE := 9;
    23      31   1)       FOR I := 3 TO N DO
    24      40   1)       BEGIN  (*FIND NEXT PRIME*)
    25      40   1)          REPEAT X := X+INC;  INC := 6-INC;
    26      48   1)             IF SQUARE <=X THEN
    27      52   1)                BEGIN LIM := LIM +1;
    28      56   1)                   V[LIM] := SQUARE;  SQUARE := SQR(P[LIM+1])
    29      70   1)                END;
    30      71   1)             K := 2;  PRIM := TRUE;
    31      75   1)             WHILE PRIM AND (K<LIM) DO
    32      81   1)             BEGIN K := K+1;
    33      85   1)                IF V[K] < X THEN V[K] := V[K] + P[K]*2;
    34     111   1)                PRIM := X <> V[K]
    35     116   1)             END
    36     119   1)          UNTIL PRIM;
    37     122   1)          IF I <= N1 THEN P[I] := X;
    38     132   1)         {WRITE(X:6);}L := L+1;
    39     136   1)          IF L = 10 THEN
    40     140   1)             BEGIN {WRITELN;} L := 0;
    41     142   1)             END
    42     142   1)       END;
    43     149   1)      {WRITELN;}
    44     149   1)    end {for q};
    45     156   1)
    46     156   1) writeln('RUN TIME = ',runtime:1,' MILLISECONDS');
    47     174   1) end.

****        47  LINE(S) READ,    1 PROCEDURE(S) COMPILED,
****       178  P_INSTRUCTIONS GENERATED
****        NO  SYNTAX ERROR(S) DETECTED.
```

PRIME.A LISTING

```
SOURCE LINE 18SL=P, ABS_AD=T, $S=Ø

    ØØØØØØØØ  DEAØFFFFFFFF  LADDR   R1Ø,-1
    ØØØØØØØ6  A6AØØØ4Ø      STORE   R1Ø,64
    ØØØØØØØA  11EØ          MOVEI   R14,Ø
    ØØØØØØØC  11FØ          MOVEI   R15,Ø
    ØØØØØØØE  9BØØFFFFFFFF  BR      $MAINBLK
$MAINBLK:
    ØØØØØØ14  A7BEØØØØ      STORE   R11,R14,Ø
    ØØØØØØ18  A7FEØØØ8      STORE   R15,R14,8
    ØØØØØØ1C  Ø1FE          MOVE    R15,R14
    ØØØØØØ1E  DFEEFFFFFFFF  LADDR   R14,R14,-1
    ØØØØØØ24  93BØFFFFFFFF  CALL    R11,SYSENTRY
    ØØØØØØ2A  CEØØØØ84      LADDR   RØ,132
    ØØØØØØ2E  A7ØEØØ18      STORE   RØ,R14,24
    ØØØØØØ32  93BØFFFFFFFF  CALL    R11,FDF
    ØØØØØØ38  CE1ØØØ8C      LADDR   R1,14Ø
    ØØØØØØ3C  A71EØØ18      STORE   R1,R14,24
    ØØØØØØ4Ø  93BØFFFFFFF2  CALL    R11,FDF
    ØØØØØØ46  CE1ØØØ32      LADDR   R1,5Ø
    ØØØØØØ4A  A61Ø1Ø28      STORE   R1,4136
SOURCE LINE 2Ø
    ØØØØØØ4E  1121          MOVEI   R2,1
    ØØØØØØ5Ø  A62Ø1Ø24      STORE   R2,4132
    ØØØØØØ54  1311          ADDI    R1,1
    ØØØØØØ56  A61Ø1148      STORE   R1,4424
    ØØØØØØ5A  8812FFFF      BR      R1<=R2,L5
F4:
SOURCE LINE 21
    ØØØØØØ5E  11Ø2          MOVEI   RØ,2
    ØØØØØØ6Ø  A6ØØ1Ø18      STORE   RØ,412Ø
SOURCE LINE 22
    ØØØØØØ64  1111          MOVEI   R1,1
    ØØØØØØ66  A61Ø1ØØ8      STORE   R1,41Ø4
    ØØØØØØ6A  1124          MOVEI   R2,4
    ØØØØØØ6C  A62Ø1ØØC      STORE   R2,41Ø8
    ØØØØØØ7Ø  1131          MOVEI   R3,1
    ØØØØØØ72  A63Ø1Ø1Ø      STORE   R3,4112
    ØØØØØØ76  1149          MOVEI   R4,9
    ØØØØØØ78  A64Ø1Ø14      STORE   R4,4116
SOURCE LINE 23
    ØØØØØØ7C  1153          MOVEI   R5,3
    ØØØØØØ7E  A65Ø1ØØØ      STORE   R5,4Ø96
```

```
    00000082  CE6003E9       LADDR   R6,1001
    00000086  A6601140       STORE   R6,4428
    0000008A  8865FFFF       BR      R6<=R5,L7
    0000008E  01C1           MOVE    R12,R1
VARIABLE AT 1,4104 ASSIGNED TO REGISTER 12
    00000090  01D2           MOVE    R13,R2
VARIABLE AT 1,4108 ASSIGNED TO REGISTER 13
    00000092  01B6           MOVE    R11,R6
F6:
R8:
SOURCE LINE 25
    00000094  010C           MOVE    R0,R12
    00000096  030D           ADD     R0,R13
    00000098  01C0           MOVE    R12,R0
    0000009A  1116           MOVEI   R1,6
    0000009C  012D           MOVE    R2,R13
    0000009E  0412           SUB     R1,R2
    000000A0  01D1           MOVE    R13,R1
SOURCE LINE 26
    000000A2  C6301014       LOAD    R3,4116
    000000A6  8030FFFF       BR      R3>R0,E9
SOURCE LINE 27
    000000AA  C6401010       LOAD    R4,4112
    000000AE  1341           ADDI    R4,1
    000000B0  A6401010       STORE   R4,4112
SOURCE LINE 28
    000000B4  0154           MOVE    R5,R4
    000000B6  7042           LSLI    R4,2
    000000B8  A73410B0       STORE   R3,R4,4272
    000000BC  C7641030       LOAD    R6,R4,4144
    000000C0  0566           MPY     R6,R6
SOURCE LINE 29
    000000C2  A6601014       STORE   R6,4116
E9:
L10:
SOURCE LINE 30
    000000C6  1102           MOVEI   R0,2
    000000C8  A6001004       STORE   R0,4100
    000000CC  1111           MOVEI   R1,1
    000000CE  A0101020       STOREB  R1,4140
W11:
SOURCE LINE 31
    000000D2  C0001020       LOADB   R0,4140
    000000D6  C6101004       LOAD    R1,4100
    000000DA  C6201010       LOAD    R2,4112
    000000DE  5112           TESTLT  R1,R2
    000000E0  0B10           AND     R1,R0
    000000E2  8E11FFFF       BR      R1<>1,L12
SOURCE LINE 32
    000000E6  C6301004       LOAD    R3,4100
    000000EA  1331           ADDI    R3,1
    000000EC  A6301004       STORE   R3,4100
```

```
SOURCE LINE 32
    00000E6  C6301004       LOAD    R3,4100
    00000EA  1331           ADDI    R3,1
    00000EC  A6301004       STORE   R3,4100
SOURCE LINE 33
    00000F0  0143           MOVE    R4,R3
    00000F2  7032           LSLI    R3,2
    00000F4  C75310B0       LOAD    R5,R3,4272
    00000F8  016C           MOVE    R6,R12
    00000FA  8865FFFF       BR      R6<=R5,E13
    00000FE  C77310B0       LOAD    R7,R3,4272
    0000102  C723102C       LOAD    R2,R3,4140
    0000106  7021           LSLI    R2,1
    0000108  0327           ADD     R2,R7
    000010A  A72310B0       STORE   R2,R3,4272
E13:
L14:
SOURCE LINE 34
    000010E  010C           MOVE    R0,R12
    0000110  C6101004       LOAD    R1,4100
    0000114  0121           MOVE    R2,R1
    0000116  7012           LSLI    R1,2
    0000118  CF3110B0       LADDR   R3,R1,4272
SOURCE LINE 35
    000011C  C7430000       LOAD    R4,R3,0
    0000120  5A40           TESTNE  R4,R0
    0000122  A040102C       STOREB  R4,4140
SOURCE LINE 36
    0000126  8B00FFAD       BR      W11
L12:
    000012A  C000102C       LOADB   R0,4140
    000012E  8E01FF67       BR      R0<>1,R8
SOURCE LINE 37
    0000132  C6101000       LOAD    R1,4096
    0000136  CE200021       LADDR   R2,33
    000013A  8012FFFF       BR      R1>R2,E15
    000013E  0131           MOVE    R3,R1
    0000140  7012           LSLI    R1,2
    0000142  014C           MOVE    R4,R12
    0000144  A741102C       STORE   R4,R1,4140
E15:
L16:
SOURCE LINE 38
    0000148  C6001018       LOAD    R0,4120
    000014C  1301           ADDI    R0,1
    000014E  A6001018       STORE   R0,4120
SOURCE LINE 39
    0000152  8E0AFFFF       BR      R0<>10,E17
SOURCE LINE 40
    0000156  1110           MOVEI   R1,0
    0000158  A6101018       STORE   R1,4120
E17:
```

```
    ØØØØØ162  A6ØØ1ØØØ       STORE   RØ,4Ø96
    ØØØØØ166  8AØBFF2F       BR      RØ<>R11,F6
    ØØØØØ16A  A6CØ1ØØ8       STORE   R12,41Ø4
    ØØØØØ16E  A6DØ1ØØC       STORE   R13,41Ø8
L7:
SOURCE LINE 44
    ØØØØØ172  C6ØØ1Ø24       LOAD    RØ,4132
    ØØØØØ176  13Ø1           ADDI    RØ,1
    ØØØØØ178  A6ØØ1Ø24       STORE   RØ,4132
    ØØØØØ17C  C61Ø1148       LOAD    R1,4424
    ØØØØØ18Ø  8AØ1FEDF       BR      RØ<>R1,F4
L5:
SOURCE LINE 46
    ØØØØØ184  CEØØØØ8C       LADDR   RØ,14Ø
    ØØØØØ188  188B           NOTI    R8,11
    ØØØØØ18A  CE1ØFFF4       LADDR   R1,-12
    ØØØØØ18E  E798ØØ1E       LOADP   R9,R8,3Ø
    ØØØØØ192  B798ØØØØØØØØ  STORE   R9,R8,Ø
    ØØØØØ198  8784FFF7       LOOP    R8,4,*-1Ø
    ØØØØØ19C  8BØØØØ1Ø       BR
    ØØØØØ1AC  112B           MOVEI   R2,11
    ØØØØØ1AE  113B           MOVEI   R3,11
    ØØØØØ1BØ  A7ØE ØØ18      STORE   RØ,R14,24
    ØØØØØ1B4  A71E ØØ2Ø      STORE   R1,R14,32
    ØØØØØ1B8  A72E ØØ28      STORE   R2,R14,4Ø
    ØØØØØ1BC  A73E ØØ3Ø      STORE   R3,R14,48
    ØØØØØ1CØ  93BØFFFFFFFF   CALL    R11,WRS
    ØØØØØ1C6  C7ØE ØØ18      LOAD    RØ,R14,24
    ØØØØØ1CA  A7ØE ØØØØ      STORE   RØ,R14,Ø
    ØØØØØ1CE  13E8           ADDI    R14,8
    ØØØØØ1DØ  93BØFFFFFFFF   CALL    R11,RUNTIME
    ØØØØØ1D6  1111           MOVEI   R1,1
    ØØØØØ1D8  C72EFFF8       LOAD    R2,R14,-8
    ØØØØØ1DC  14E8           SUBI    R14,8
    ØØØØØ1DE  A72E ØØ18      STORE   R2,R14,24
    ØØØØØ1E2  A7ØE ØØ2Ø      STORE   RØ,R14,32
    ØØØØØ1E6  A71E ØØ28      STORE   R1,R14,4Ø
    ØØØØØ1EA  93BØFFFFFFFF   CALL    R11,WRI
    ØØØØØ1FØ  C7ØE ØØ18      LOAD    RØ,R14,24
    ØØØØØ1F4  188F           NOTI    R8,15
    ØØØØØ1F6  CE1ØFFE4       LADDR   R1,-28
    ØØØØØ1FA  E798ØØ22       LOADP   R9,R8,34
    ØØØØØ1FE  B798FFFFFFF4   STORE   R9,R8,-12
    ØØØØØ2Ø4  8784FFF7       LOOP    R8,4,*-1Ø
    ØØØØØ2Ø8  8BØØØØ14       BR
    ØØØØØ21C  112D           MOVEI   R2,13
    ØØØØØ21E  113D           MOVEI   R3,13
    ØØØØØ22Ø  A7ØE ØØ18      STORE   RØ,R14,24
    ØØØØØ224  A71E ØØ2Ø      STORE   R1,R14,32
    ØØØØØ228  A72E ØØ28      STORE   R2,R14,4Ø
    ØØØØØ22C  A73E ØØ3Ø      STORE   R3,R14,48
    ØØØØØ23Ø  93BØFFFFFF9Ø   CALL    R11,WRS
```

```
    00000236  C70E0018        LOAD    R0,R14,24
    0000023A  A70E0018        STORE   R0,R14,24
    0000023E  93B0FFFFFFFF    CALL    R11,WLN
SOURCE LINE 47
    00000244  1110            MOVEI   R1,0
    00000246  A71E0018        STORE   R1,R14,24
    0000024A  93B0FFFFFFFF    CALL    R11,SYSEXIT
    00000250  C7BF0000        LOAD    R11,R15,0
    00000254  01EF            MOVE    R14,R15
    00000256  C7FF0008        LOAD    R15,R15,8
    0000025A  57BB            RET     R11,R11
  NUMBER OF BYTES OF CODE GENERATED = 604
```

PRIME.MAP LISTING

```
Ridge Linker Version of 29-Dec-82
Symbol table
$HEAP                00020258 Code      $MAINBLK             00000014 Code
$HEAP                00022E58 Code      ALLOCINIT            00000274 Code
$HEAP                00025178 Code      DISPOSE              000002A4 Code
$MAINBLK             00000014 Code      ALLOC                00003AE  Code
$MAINBLK             00002E2E Code      NEWSTRING            00000626 Code
$MAINBLK             00004C7A Code      SUBSTRING            0000065C Code
$MAINBLK             00005764 Code      OVERLAYSTRING        000006E6 Code
ABORTCOMMAND         000010B0 Code      COPYOFSTRING         0000075C Code
ACQUIREDEVICE        00005124 Code      CONCATSTRING         000007DA Code
ACTIVATE             00004EE4 Code      EQUALSTRING          00008B6  Code
ALLOC                00003AE  Code      COPYSUBSTRING        0000094E Code
ALLOCINIT            00000274 Code      FILLSTRING           000009C0 Code
ARM                  00004D72 Code      SEARCHSTRING         00000A1A Code
ATN                  000058D0 Code      MSGINIT              00000A92 Code
CHANGEDIR            00001A0A Code      PUTMESSAGE           00000AD2 Code
CHANGEFILESIZE       0000161E Code      GETMESSAGE           00000B36 Code
CLOSE                00000E10 Code      COPYFIDTOINTS        00000BAA Code
CLOSEFILE            00002C60 Code      COPYINTSTOFID        00000BDA Code
CLOSELINK            00004F80 Code      FETCHARGS            00000C06 Code
CLOSEQUEUE           00004F66 Code      CREATE               00000C9C Code
CONCATSTRING         000007DA Code      OPEN                 00000D54 Code
CONVTTIMETOYMODH     0000523E Code      CLOSE                00000E10 Code
CONVYMODHMISTNTO     0000547C Code      CREATEEQUATE         00000E78 Code
COPYFIDTOINTS        00000BAA Code      DELETEEQUATE         00000F78 Code
COPYINTSTOFID        00000BDA Code      LOADCOMMAND          00001014 Code
COPYOFSTRING         0000075C Code      ABORTCOMMAND         000010B0 Code
COPYSUBSTRING        0000094E Code      STARTCOMMAND         00001118 Code
COS                  00005810 Code      CREATEPIPE           000011AC Code
CREATE               00000C9C Code      DELETE               000011C4 Code
CREATEEQUATE         00000E78 Code      READBLOCK            00001258 Code
CREATEPIPE           000011AC Code      WRITEBLOCK           0000136E Code
CREATEPROCESS        00004EA2 Code      READCHAR             0000147A Code
CREATESPECIAL        0000168E Code      WRITECHAR            0000154C Code
DAYOFWEEK            00005716 Code      CHANGEFILESIZE       0000161E Code
```

```
DAYSECSNSECSTOTI    ØØØØ57D2 Code
DELETE              ØØØØ11C4 Code
DELETEEQUATE        ØØØØØF78 Code
DELETEMESSAGE       ØØØØ5Ø9Ø Code
DELETEPAGE          ØØØØ4E3E Code
DISARM              ØØØØ4D84 Code
DISPOSE             ØØØØØ2A4 Code
EQUALSTRING         ØØØØØ8B6 Code
EXAMINEFRAME        ØØØØ4E58 Code
EXAMINEPAGE         ØØØØ4DA2 Code
EXP                 ØØØØ5A44 Code
EXPO                ØØØØ36A4 Code
FAULTCOMPLETE       ØØØØ4E9Ø Code
FDF                 ØØØØ4B34 Code
FETCHARGS           ØØØØØC Ø6 Code
FIB                 ØØØØ26A2 Code
FILECHANGESIZE      ØØØØ1FF2 Code
FILECLOSE           ØØØØ2588 Code
FILEDELETED         ØØØØ5ØA2 Code
FILEINIT            ØØØØ1C48 Code
FILEOPEN            ØØØØ218Ø Code
FILESEEK            ØØØØ1CCC Code
FILESTATUS          ØØØØ2C96 Code
FILLSTRING          ØØØØØØ9CØ Code
FIX                 ØØØØ5ØBC Code
FLB                 ØØØØ2892 Code
FLUSH               ØØØØ5Ø6E Code
FREE                ØØØØ5ØDE Code
GETARGS             ØØØØ1B14 Code
GETC                ØØØØ2B2Ø Code
GETCURRENTDIR       ØØØØ1A9E Code
GETMESSAGE          ØØØØØB36 Code
GETSPECIALPID       ØØØØ5156 Code
GETTERMINALDEV      ØØØØ1BE8 Code
GETYMD              ØØØØ56A4 Code
INITQSEG            ØØØØ4FA4 Code
INITTABLE           ØØØØ5F8Ø Code
INSERTPAGE          ØØØØ4EØ8 Code
ISLEAP              ØØØØ5198 Code
KILL                ØØØØ4FØ8 Code
LEAPSTO             ØØØØ51DA Code
LOADCOMMAND         ØØØØ1Ø14 Code
LOG                 ØØØØ5AD2 Code
LOOKUPNAME          ØØØØ1952 Code
MODIFYPAGE          ØØØØ4DDA Code
MSGINIT             ØØØØØA92 Code
MYID                ØØØØ4F9A Code
NEWSTRING           ØØØØØ626 Code
OPEN                ØØØØØD54 Code
OPENFILE            ØØØØ2C12 Code
OPENLINK            ØØØØ4F3C Code
OPENQUEUE           ØØØØ4F1A Code
WRI                 ØØØØ2FF8 Code
WRITEBLOCK          ØØØØ136E Code
WRITECHAR           ØØØØ154C Code
WRITEPROCESSSTAT    ØØØØ5112 Code
WRR                 ØØØØ3828 Code
WRS                 ØØØØ344C Code
CREATESPECIAL       ØØØØ168E Code
READDIRECTORY       ØØØØ1732 Code
READLABEL           ØØØØ1812 Code
SPACEINFO           ØØØØ18AE Code
LOOKUPNAME          ØØØØ1952 Code
CHANGEDIR           ØØØØ1AØA Code
GETCURRENTDIR       ØØØØ1A9E Code
GETARGS             ØØØØ1B14 Code
GETTERMINALDEV      ØØØØ1BE8 Code
FILEINIT            ØØØØ1C48 Code
FILESEEK            ØØØØ1CCC Code
FILECHANGESIZE      ØØØØ1FF2 Code
FILEOPEN            ØØØØ218Ø Code
FILECLOSE           ØØØØ2588 Code
FIB                 ØØØØ26A2 Code
FLB                 ØØØØ2892 Code
GETC                ØØØØ2B2Ø Code
PUTC                ØØØØ2B8C Code
OPENFILE            ØØØØ2C12 Code
CLOSEFILE           ØØØØ2C6Ø Code
FILESTATUS          ØØØØ2C96 Code
POSITIONFILE        ØØØØ2CF2 Code
SETFILESIZE         ØØØØ2D3C Code
SYSENTRY            ØØØØ2D7E Code
SYSEXIT             ØØØØ2DA2 Code
$MAINBLK            ØØØØ2E2E Code
WRC                 ØØØØ2E74 Code
RDC                 ØØØØ2F82 Code
WRI                 ØØØØ2FF8 Code
RDI                 ØØØØ3232 Code
WRS                 ØØØØ344C Code
RDS                 ØØØØ35A8 Code
EXPO                ØØØØ36A4 Code
WRR                 ØØØØ3828 Code
POWEROFTEN          ØØØØ3F94 Code
RDR                 ØØØØ4ØA4 Code
WRB                 ØØØØ467Ø Code
RDB                 ØØØØ4724 Code
RBW                 ØØØØ4358 Code
PAG                 ØØØØ498C Code
WLN                 ØØØØ4AØA Code
RLN                 ØØØØ4A88 Code
RES                 ØØØØ4B14 Code
REW                 ØØØØ4B24 Code
FDF                 ØØØØ4B34 Code
$MAINBLK            ØØØØ4C7A Code
SEND                ØØØØ4CBØ Code
RECEIVE             ØØØØ4CDE Code
SENDPAGE            ØØØØ4DØC Code
RECEIVEPAGE         ØØØØ4D3A Code
TEST                ØØØØ4D6Ø Code
ARM                 ØØØØ4D72 Code
EXP                 ØØØØ5A44 Code
LOG                 ØØØØ5AD2 Code
INITTABLE           ØØØØ5F8Ø Code
$HEAP               ØØØ2Ø258 Code
$HEAP               ØØØ22E58 Code
$HEAP               ØØØ25178 Code
Link Completed. Size of program: 24452. No errors detected.
Output file is runnable object file
```

Ridge Computers
Corporate Headquarters

2451 Mission College Blvd.
Santa Clara, California 95054
Phone: (408) 986-8500
Telex: 176956