RUN/C Professional™

Age of Reason's C-Handling Environment and Repertoire

for MS-DOS or PC-DOS 2.0 and Greater

This manual describes Version 1
of RUN/C Professional

Software by Stephen Walton and Peter Brooks

Documentation by Todd Katz

Published by
Lifeboat Associates
1651 Third Ave.
New York, NY 10128

Tel: (212) 860-0300
Telex: 424490 (LBSOFT NYK)

Copyright (c) 1984, 1985
Age of Reason Co.
318 E. 6th St.
New York, NY 10003
Phone: (212) 473-2309

All Rights Reserved

Preface
Acknowledgments 3
Support 4

Section 1 Getting Started with RUN/C
Summary 1-1
Accessing RUN/C Programs 1-2
Before you begin 1-7
Product overview 1-8
Errors 1-11
Making safety copies 1-12
Testing RUN/C 1-14
Protecting your hard disk 1-15
CONFIG.SYS and device drivers 1-16
Command line options 1-18
Reserved words 1-22
Input/Output 1-23
Printer support 1-26
Math chip support 1-26

Section 2 Editing RUN/C Programs
Overview 2-1
About the RUN/C Editor 2-3
Cursor movement 2-5
Delete and Miscellaneous 2-11
Block operations 2-17
EDITOR and SHELL 2-21
Using SET EDITOR 2-23
SHELL command 2-24

Section 3 Numeric Expressions and Operators
Overview 3-1
Arithmetic operators 3-1
Assignment operators 3-2
Relational and logical operators 3-3
Increment and decrement operators 3-4
Bitwise operators 3-5
Assorted operators and expressions 3-6
Operator precedence 3-10

Section 4 Commands, Keywords and Functions
Summary 4-1
RUN/C terms by categories 4-2
Commands 4-3
Keywords 4-8
Functions 4-11
Miscellaneous 4-20
RUN/C terms in alphabetical order 4-21

Section 5 Using RUN/C Professional Loadable Libraries
Summary 5-1
LIB 5-4
LIBDIR 5-5
LIBLOAD 5-7
LIBSHOW 5-9
LIBUNLOAD 5-12
Using a loaded library function 5-13
Creating your own loadable library 5-16
Extracting functions from .LIB files 5-24
Accessing global variables in loaded libraries 5-24
Making RUN/C globals available to library functions 5-29
Assembly language requirements 5-30
CHECKRCL 5-33

Section 6 Debugging RUN/C Professional Programs
Summary 6-1
The runc() function 6-2
The BREAK and VAL commands 6-6

Section 7 Important Technical and Other Notes
Summary 7-1
Line vs. stream 7-2
A safe nest 7-6
Scope quirks 7-7
About the editor 7-9
About the manual 7-10
How close to a full C are we 7-10
Compiler issues 7-12

Appendix A RUN/C Screen Messages
Summary A-1
Numbered error messages A-1
Unnumbered error messages A-14
Other screen messages A-21

Appendix B Suggested Readings

Appendix C Some Common C Bugs

Appendix D The BASIC_C Graphics Library

Appendix E RUN/C product history

Addenda -- to be supplied as needed

Index

PREFACE

A Programming Environment for C

A C programming language development system needs three features to be fully accessible to the serious programmer: interpretation, accessibility to commercially-developed routines in that interpreted system and source level debugging.

RUN/C Professional provides these features. The interpreter combines many of the best features of two popular computer languages: BASIC and C. Like BASIC, RUN/C is interpreted, meaning that you can quickly write a program, run it, change it and run it again -- all within a single consistent environment closely integrated with a powerful full-screen editor. As an implementation of the C programming language, RUN/C interprets what is known as K&R* C. A library of more than 100 C functions is built into RUN/C.

Because more and more program development in C is dependent upon function libraries supplied by so-called third-party vendors, a C interpreter must be able to access these experts' systems. RUN/C's Loadable Libraries(tm) system provides such access. Libraries can be created rapidly and can be loaded, unloaded and combined with other libraries. These libraries can be called in by the executing program at any time.

RUN/C Professional also provides a rich set of source code debugging features: multiple breakpoints, tracing of executing program statements or variable values, single stepping and a built-in program profiler. The ability to halt program execution and determine and/or change the current value of any program variable is also quite useful. All relevant interpreter commands can be called by the executing program. These features -- rich in and of themselves -- can be combined in any manner. Thus, for example, it is possible to simultaneously trace variable and program statements and "break" when a particular variable has a negative value.

* * *

---

* K&R is the 'handle' used throughout the industry to refer to Brian Kernighan and Dennis Ritchie, the creators of C and authors of its definition, The C Programming Language (Prentice-Hall, 1978).

RUN/C Professional was designed and written by Stephen Walton and Peter Brooks. Brooks, a C specialist with a love for mathematics, created the full screen editor and Loadable Library system and designed the RUN/C Professional debugging functions. In addition, Brooks served as 'language lawyer' and debugging specialist for the project.

Brooks is also Executive VP of The Instructor Set, a company which teaches seminars on C and UNIX throughout the U.S. In addition to RUN/C, he is the author of LB-MAKE and LB-PROFILER, both available from Lifeboat Associates.

Steve Walton is a journalist by trade and a software author by choice. His software publications include: SKETCHMODE, a graphics manager, and STARCLASH, a strategy game (Hayden, 1980); PROSET, a system for coding files for typesetting (Cromwell Graphics Corp., 1982); THE ELECTRIC DRAGON, a computerized version of the ancient Chinese oracle the I Ching, STARCLASH II, and FORCE 400, another original strategy game, (Baen Software and Simon & Schuster Electronic Press, 1984). RUN/C -- The C Interpreter (Lifeboat Associates, 1985). THE EXPERT(tm), an expert system system, awaits publication.

Walton's well-received novel, No Transfer (Vanguard Press, 1967), was published while the author was in college.

RUN/C is based on a concept designed by John Reaves.

Todd Katz
President
Age of Reason Co.

New York December, 1985

SECTION 1. GETTING STARTED WITH RUN/C PROFESSIONAL

Summary

This section isn't required reading. It provides general information on RUN/C Professional for users with widely divergent experiences and interests.

We will first provide a sample run of a program under RUN/C Pro -- please don't try to follow along unless you have made a back-up copy of the distribution disk.

Then we will tell you how to make a "back-up" or safety copy of RUN/C. Next we will go over some of the typographical conventions we have used in this manual. Finally, some important but miscellaneous terms are described in a subsection on input and output.

As we are feeling a bit guilty about it, we want to make it clear that this manual does not intend or pretend to teach C or even to provide a comprehensive overview of its rules and uses. Particularly, information on arrays and pointers, two very important features of the language, is absent. For such you are respectfully referred to professional courses on C currently being taught in colleges, universities and special computer interest groups. In addition, many of the available books on C are quite good. Some of these are named in Appendix B.

Let's Run C

Before getting involved in the important details of getting you up and running, let's make certain that it will all be worthwhile by making a "dry run" of FTOC.C, a famous C program contained in K&R's The C Programming Language.

Two steps are involved in running a C program within RUN/C. The first is getting the program copied from the diskette into the memory of your computer. This is known as loading the program. The second step is the actual execution of the program's instructions. This is known as running the program.

Actually, there are two ways to accomplish these feats and we're going to show you both. The first is to **LOAD** and **RUN** the program in a single action from the command line in the same way you might a previously developed BASIC program. Secondly, we will load FTOC.C, modify it and run it again.

Accessing RUN/C Programs

DOS command line LOAD

To load RUN/C and run FTOC.C simultaneously, you should be logged-on to the *drive/directory* which contains RCP.EXE. Secondly, you should either know the location of FTOC.C or be sure that it is on your default drive/directory. Assuming that both programs are on the A> default drive/directory, FTOC can be run with the command:

```
A>RCP FTOC<enter>*
```

In a few moments your computer's screen will clear and the following will appear:

```
  0   -17.8
 20    -6.7
 40     4.4
 60    15.6
 80    26.7
100    37.8
120    48.9
140    60.0
160    71.1
180    82.2
200    93.3
220   104.4
240   115.6
260   126.7
280   137.8
300   148.9

A>
```

Notice that you have returned to the DOS command line. When you invoke the interpreter with a program from the command line, RUN/C will return you from whence you came if the program runs correctly. If there is an error of any kind, the appropriate message will be displayed and you will remain in RUN/C.**

---

* All commands must be followed by the <enter> key. In subsequent examples, the <enter> will not be shown.

** Why return to DOS? This allows the creation of a program that can do some useful work (such as modifying communication port settings) and then return automatically to the operating system level.

The results of our program are, you might agree, a little plain. We can spruce it up by inserting some simple formatted text before the data is printed out.

Separate LOAD and RUN
This time let's **LOAD** RUN/C without FTOC.C.

```
A>RCP
```

If RUN/C is on your default drive/directory, after a few moments the following sign-on message will appear:

```
RUN/C Professional, Version n.nn
by Stephen Walton and Peter Brooks
Copyright (c) 1984, 1985 by Age of Reason Co. All rights reserved.
nnnnnn bytes free

Ok
```

The screen prompt:

```
Ok
_
```

signifies that RUN/C is at the Command level, waiting your bidding. The number of bytes free varies with the amount of memory available on your computer.

Now let's load our program into memory with:

```
LOAD FTOC.C
```

and get a listing with:

```
LIST
```

After a few moments, the lines of our program will appear:

```
 1:/*
 2:print Fahrenheit-to-Celsius table
 3:  (This program is from p. 8 of the Kernighan and
     Ritchie text)
 4:*/
 5:main()
 6:{
 7:    int lower, upper, step;
 8:    float fahr, celsius;
 9:
      10:   lower = 0;      /* lower limit */
      11:   upper = 300;    /* upper limit */
 12:   step = 20;           /* step size */
 13:
 14:   fahr = lower;
 15:   while (fahr <= upper) {
 16:       celsius = (5.0/9.0) * (fahr-32.0);
 17:       printf("%4.0f %6.1f\n", fahr, celsius);
 18:       fahr = fahr + step;
 19:     }
 20:}
```

Ok

Examination of the program indicates that the block which prints out the conversion table begins on line 15. Let's edit our program with the goal of making the proper formatting changes just before line 15 with the command:

EDIT 15

(Continued next page)

```
ftoc.c                                    INSERT  Line: 1  Col: 1
                          (Exit edit: ESC or F10)  (Help: ^J or F1)

/*
print Fahrenheit-to-Celsius table
(This program is from p. 8 of the Kernighan and Ritchie text)
*/
main()
{
  int lower, upper, step;
  float fahr, celsius;

  lower = 0;        /* lower limit of temperature table */
  upper = 300;      /* upper limit */
  step = 20;                /* step size */

  fahr = lower;
  while (fahr <= upper) {
      celsius = (5.0/9.0) * (fahr-32.0);
          printf("%4.0f %6.1f\n", fahr, celsius);
          fahr = fahr + step;
  }
}
```

Notice the cursor symbol on line 15 (the **while** statement). We press <enter> and then insert the following two lines:

```
puts("\tCopyright(c) 1979 by Bell Labs.");
puts("\t\t\tFahrenheit\tCelsius");
<Esc>
```

We also should make a change in the line which prints the f-to-c table. Simply move the cursor down to the line:

```
printf("%4.0f %6.1f\n", fahr, celsius);
```

Using the cursor control keys and editing commands (fully described in Section 2), we change line 19 to read:

```
19:printf("\t\t\t%4.0f \t\t%6.1f\n", fahr, celsius);
```

When satisfied with the line, we press <enter> to copy it to the appropriate line in memory and then follow with an <Esc> to

return to the RUN/C Command level.

Using the same editing system, we might modify the variable **upper** to equal 250 instead of 300 and the variable **step** to equal 15 instead of 20. Now when the program is called with the command:

```
RUN
```

we will see:

```
K&R's F-to-C Program

Copyright(c) 1984 by Bell Labs.
      Fahrenheit Centigrade
      0              -17.8
      15              -9.4
      30              -1.1
      45               7.2
      60              15.6
      75              23.9
      90              32.2
      105             40.6
      120             48.9
      135             57.2
      150             65.6
      165             73.9
      180             82.2
      195             90.6
      210             98.9
      225            107.2
      240            115.6

           Ok
```

Now we can SAVE our new program with:

```
SAVE NEW-FTOC.C
```

and return again to the operating system with:

```
SYSTEM
```

Combined LOAD and RUN

Once inside RUN/C Professional it is possible combine the **LOAD** and **RUN** operations into a single step by using the **EXEC** (for 'execute') command:

```
EXEC FTOC
```

will **LOAD** and **RUN** our current program, even if it isn't currently the program in memory.

Interrupting program execution
At this point we should also introduce an important RUN/C feature -- the Control-c program interrupt. Once you have begun running a program, you may decide to terminate the **RUN** before it has completed. In some cases you may need to do this, since it is not impossible to create a C program which only ends when you reboot your computer. In RUN/C, you can interrupt the program and return to the Command level by pressing <Ctrl-c> or <Ctrl-Break>. All further references regarding program interruption from the keyboard will be to <Ctrl-c>.

Upon pressing <Ctrl-c> you will be presented with our "mini-menu" which in RUN/C Professional currently looks like:

```
Break in line n
  Continue, Dump, Ldump, Interactive, Help or End (C,D,L,I,H,E)?
```

Each of these options is described in detail in Section 4 and further discussed in Section 6.

Before you begin

Piracy of the High Cs
For your convenience, RUN/C is not copy protected. For our protection RUN/C's manual and software are copyrighted under U.S. and international copyright laws. Many months have been spent to try and provide you with the best possible program and a special effort has been made to make RUN/C affordable for everyone interested in the language.

A somewhat simplified rendering of our user license reads:

As an owner of a license for
RUN/C Professional
you may make as many back-up
copies of RUN/C for your use only
as you wish.

* * *

You may use RUN/C
on more than one computer
-- but only on one computer at a time.

* * *

Your can transfer your RUN/C license
to another person
-- but after doing so you must have
no copies of the RUN/C software or manual left.

Product overview

RUN/C is designed for:

1) creating and testing small- to mid-sized C programs.
2) to help newcomers to C learn the language faster and more completely.
3) to serve as a debugger for the development of functions which will eventually be included in large C programs.

Hardware requirements

RUN/C is designed to run on the IBM PC, PC "clones" and MS-DOS systems with standard implementations of ANSI.SYS. A minimum of 320k of memory and a double-sided disk drive to accommodate the 320k RUN/C program disk is also needed. Recommended memory is 512k.

Love that memory

All IBM PC's currently in production have at least 256k of RAM. If your PC was build before IBM made 256k the industry standard, you may need to add a memory board or multi-function board to your system in order to run RUN/C. The PC Jr., which comes factory-equipped with only 128k of RAM is a case in point; IBM and other manufacturers market memory expansion units taking Jr. to 256k and beyond. The best way to tell how much memory you have is to use the DOS program CHKDSK.COM. See your DOS manual for details.

Software Requirements

RUN/C is designed to be run only under PC DOS or MS-DOS version 2.0 or greater up through the most recent release, DOS 3.1, as of this writing. If you are uncertain about what version of

MS-DOS your system is running, try the built-in DOS command:

```
A>VER
```

You should see a message similar to:

```
IBM Personal Computer DOS Version 2.10
```

No other software is necessary for the successful operation of RUN/C. However, if you wish to compile your program, you will need a C compiler. In this respect RUN/C is similar to IBM's BASIC interpreter which requires a BASIC compiler in order to create an independently executable program from BASIC source code.

Distribution disk contents
The contents of the RUN/C distribution disks are listed in the RCP-READ.ME file on the disk. You should be able to display the contents of this file with the command:

```
A>TYPE RCP-READ.ME
```

These distribution disks contains a copy of RCP.EXE, the RCP-READ.ME file and about 100 sample programs, each of which has the extension ".C". The great majority of these programs are taken from the example programs in Section 4. Important note: C programs on the distribution disk are not the source code for RUN/C functions. **You do not need any of these sample programs for RUN/C to operate properly.**

Compiler Compatibility
RUN/C is currently most compatible with Lattice C. Although C compilers offer different implementations of the C programming language, there are many common features and functions. Thus many programs written with RUN/C will work without modification on C compilers which adher to the K&R definition of the language. These include:

* Aztec C
* Computer Innovation's C86
* DeSmet C88
* Digital Research C
* IBM C
* Lattice C
* Manx Software's Aztec C86
* Mark Williams C (Let's C)
* Microsoft C
* Wizard C

In a few cases we supply source code for special functions we

have built into RUN/C such as **locate** and **cls**. These are available in the files RCIBMLIB.C and RCANSLIB.C, the latter for systems other than IBM PCs and compatibles which use the ANSI.SYS device driver, discussed below.

Most, if not all, of the programs developed with RUN/C will be readily transportable to any standard C compiler. If you experience difficulties compiling programs with your C compiler which work with RUN/C, please let us know so that we can include specific instructions to other users of your compiler.

Documentation Conventions
When describing commands you will issue to the computer or the responses you will see on the screen we have used a combination of underline, **boldface**, *italics* and CAPITAL LETTERS. The rule we have tried to follow is that underlined indicates input from you, the user, while CAPITALIZED TERMS often indicates the computer's response. In our descriptive text **boldfaced** type is used to indicated input/output to the computer. Underline also allows us to emphasize a point here and there while *italics* refers to previously mysterious terms we plan to put in our as yet unwritten glossary.

We use a Courier monospace font to indicate a line or two which would appear on the screen like:

```
hello, world
Ok
```

If larger amounts of text need to be displayed we go to a more condensed font with a grey background such as:

```
Break in line n
  Continue, Dump, Ldump, Interactive, Help or End (C,D,L,I,H,E)?
```

The only rule -- which remains an unrealized ideal -- is to present the information in a manner which can be readily comprehended.

Control codes, escape codes and other non-standard keys such as the <tab key>, <Esc>, <Alt> and <enter> key are placed within angle brackets, as shown.

The RUN/C Professional program on disk is RCP.EXE and it is invoked with:

```
A>RCP
```

If you would prefer to use another name, RCP.EXE can be renamed with the command:

```
A>REN RCP.EXE C.EXE
```

Use any name you like. See your DOS manual for complete details on using the REN command.

Errors

When a syntax error is discovered, a message briefly describing the error and giving the error number, if any, will be displayed. Most errors are recognized either during the pre-run checking of your program or, more likely, during program execution -- after you have issued the instruction:

```
RUN
```

If an error occurs correct it and attempt to RUN the program again. We don't go in for cascading error messages, so usually only one error at a time will be reported.

Errors and other messages are listed in Appendix A of this manual.

Correcting errors

If the interpreter determines that an error condition exists, a numbered or unnumbered error message will be displayed on the screen and the **RUN** will be terminated. For example the program:

```
main()

printf("hello, world")

}
```

has a few problems. The opening curly brace ({) under **main()** is missing and the only statement in the program (line 3) lacks the semi-colon at the end of the line which would signal that the statement is completed. When this program is run the error message:

```
Right braces exceed left braces in line 5
 *** Error: unmatched braces ***
```

is displayed. (Notice that the second problem, namely the missing semi-colon in line 3, is not recognized and will not be until the primary error is corrected.) We can correct this error using the built-in editor by inserting an opening curly brace before the

printf statement.* Then, when the corrected program is run, the error message will be changed to:

```
*** Error # 191 at line 5
Illegal termination of expression
```

since no appropriate end to the expression containing

```
printf("hello, world")
```

is found.

For a complete list of messages see Appendix A.

Making safety copies

The RUN/C Professional distribution disk(s) you received will be labeled with the name of the publisher and a serial number which has been assigned to the licensee. RCP.EXE may be copied to another disk, to a hard (fixed) disk or (temporarily) to a RAM disk. We recommend making two floppy disk copies: one for day-to-day use and one as a back-up in case your day-to-day copy is lost, destroyed or goes bad. The distribution disk(s) should be kept in a safe place, preferably away from your day-to-day disks.

Your computer's manual is the best place to learn about making a back-up copy of a disk. However, if your system manual isn't handy, the instructions below may suffice.

Installing RUN/C

On one edge of your disk you will notice a plastic tab. This is the write protect tab. It means that you can read the contents of the disk, but you cannot write to the disk or erase anything on it. **Never remove the write-protect tab from distribution disks.**

The procedure for copying diskettes varies somewhat from machine to machine. We are going to present two common copying scenarios:

Making Safety Copies on a Double Disk System

1. Get four new double-sided, double-density diskettes.*

---

* For detailed examples of the editing commands see Section 2.

* Although discussions in this manual are keyed to the 5.25-inch double-sided, double-density disk format, RUN/C is available for most formats which use the MS-DOS operating system.

2. Insert your back-up DOS diskette into the A> disk drive and place your disk to be formatted into the B> drive. Type:

```
A>FORMAT B:
```

The message:

```
Insert new diskette for drive B:
and strike any key when ready
```

will appear. Insert the disk to be formatted into the B> drive and press any key. When the formatting is complete the message:

```
Formatting...Format complete
362,496 bytes total disk space
362,496 bytes available on disk
Format another (Y/N)?
```

will appear. Insert the next disk to be formatted and answer Y. After you have formatted four disks, answer N to end the formatting program.

3. To make an actual copy of the contents of a distribution disk place it in the A> drive and type:

```
A>COPY A:*.* B:/V*
```

The names of the files on the disk will be listed as they are successfully copied to the B> drive.

4. After you have copied one disk, insert the second formatted disk in the B> drive and repeat step 3.

5. Repeat the procedure for the socond distribution disk.

You now have two back-up copies of the distribution disks. Be sure And mark your disks properly.

Making Safety Copies on a Single Disk System

1) Get four double-sided, double-density diskettes.

2) Insert your DOS diskette into the A> disk drive. This time we will use the DOS program DISKCOPY.COM which both formats and copies the contents of one disk onto another. To begin type:

```
A>DISKCOPY
```

---

* /V for verify.

```
Insert source disk, drive A:
Strike any key when ready
```

will initiate a routine which will call upon you to insert the source and target disk several times, exactly how many depends on how on how much memory your system has. After several passes everything on your distribution disk will have been successfully copied and the message:

```
Copy complete
Copy another (Y/N)
```

will appear. To make a second safety copy repeat steps one and two. Then do the same thing for the second distribution disk.

There are several variations on the copying theme including copying the DOS system files onto your RUN/C safety disk, copying RUN/C onto a hard disk and using a 9-sector format rather than the distributed double-sided 8-sector format. For more information on these issues see sections on COPY, FORMAT, DISKCOPY and SYS in your DOS manual.

Testing RUN/C

Since there is no installation procedure for RUN/C, testing it is very simple. Just follow these steps:

1) Turn on the computer and insert the DOS boot disk into your A> disk drive (if you are using a PC XT, or an enhanced PC or AT with a hard disk you may automatically boot from the hard disk -- in which case this second portion of this step can be skipped). Wait until the system has booted up and the A> prompt has appeared.

2) Place your day-to-day copy of RUN/C in drive A> and enter the command:

```
A>RCP
```

The RUN/C sign-on message and **Ok** prompt should appear. If not, there are two possibilities:

1) You are attempting to use the program in an inappropriate environment or,

2) The software you received has somehow been damaged or improperly manufactured.

If you cannot resolve the problem through hints in this section

and, less likely, Section 7, contact your dealer, distributor or authors.

Protecting your hard disk

As noted above, RUN/C is not copy protected and there is no impediment to copying and using the program from a hard (or "fixed" disk). However, in developing programs in a powerful, low-level language such as C there exists the remote possibility that a stray pointer and/or ill-constructed array could "escape" from the program and data space and enter into areas of memory which should not be tampered with.*

Normally the worst that might be expected would be the necessity of a reboot of the system -- warm or even cold -- but under current versions of MS-DOS the File Allocation Table (FAT) that keeps track of the names and locations of files, directories and sub-directories is also fair game for errant pointers passing garbage data.

Therefore it is a good, prudent and recommended practice that the default drive be a floppy disk even if you load RUN/C from a hard disk. One easy way to accomplish this is to do a:

```
SHELL A:
```

at the beginning of your session or to have a header file which contains the statement:

```
runc("SHELL A:")
```

After you have tested your program sufficiently to be satisfied with its stability, you can copy it back onto the hard disk either from within RUN/C or from the operating system level.

To return to DOS

You may return to DOS any time after invoking RUN/C by first pressing <Esc> (to return to Command Mode, described below) followed by the word **SYSTEM** as in:

```
Ok
SYSTEM
```

---

* This is true of not only RUN/C, but of any C program whether compiled or interpreted. Unlike most C compilers, RUN/C performs both array bounds and pointer range checking (using techniques and subject to limitation described in Section 7), but it may not always be completely successful.

The only time this won't work is if a program is running in the interpreter. In that case press <Ctrl-c> to enter RUN/C's mini-menu (described in Section 4 under such topics as E(End) and then choose the E option to exit the program. Then type SYSTEM.

If your RUN/C disk is logged into the system (default) drive and your disk does not have a valid copy of the DOS operating system on it, you will probably get a message similar to:

```
Insert COMMAND.COM disk in drive A:
and strike any key when ready
```

Insert your DOS disk, press any key, and the A> prompt will reappear.

CONFIG.SYS and Device Drivers

The CONFIG.SYS file is special because every time the MS-DOS operating system "boots up" it looks to see if a file named CONFIG.SYS is on the default drive/directory. If so, its contents will be read before anything else. CONFIG.SYS can contain names of devices drivers and various system settings. Quite possibly you are already using CONFIG.SYS for a RAM disk, the ANSI.SYS screen driver, various mice and other devices.

RUN/C will essentially ignore the ANSI.SYS device driver even if it is named in the CONFIG.SYS file on IBM PC and close compatibles. This too is discussed in below.

We recommend placing two other lines in your CONFIG.SYS file. These are:

```
FILES=20
BUFFERS=16
```

The first setting allows up to 20 files to remain open simultaneously. The term "files" can be confusing, since various I/O devices such as the screen and printer are considered "files" by DOS. RUN/C currently opens as many a five "files" without opening any data files. Your program can open many more. The IBM PC has a default **FILES** setting at eight, although the maximum allowable is 99. We believe 20 should suffice for most applications.

The buffers setting is optional. Setting the number above the default level to 16 will, however, speed up some system processing, especially file loading and saving.

PC DOS or MS-DOS

RUN/C automatically adapts to the two most common variations of the MS-DOS operating system: either the IBM PC native mode (PC DOS) or Microsoft's ANSI.SYS screen addressing mode, which uses industry standard escape sequences to address the computer screen.

Each time RUN/C is started, a software check is made to determine two pieces of information: is the system an IBM computer or a complete compatible? and if not is an ANSI.SYS device driver active?

If your system appears to our software to be an IBM PC, XT, Jr. or AT the PC's *native mode* will be used even if the ANSI.SYS device driver is referenced in your CONFIG.SYS file. Only if RUN/C determines that your machine is not an IBM computer will it look for the ANSI.SYS device driver. If it is found, it will be used, otherwise RUN/C will assume that your machine uses the same video interface as the IBM PC in its native (non-ANSI) mode.

It will be easy for you to tell whether RUN/C is using the PC native mode or ANSI.SYS mode. Just watch to see if the cursor changes shape when you press the <Ins> key while in the RUN/C editor. If it does, you are using native PC mode. If it does not, the word:

INSERT

should appear near the bottom of your screen. Another good sign is if the function key assignments are displayed on the 25th line of your terminal. If they are, PC DOS mode is probably active.

If the cursor doesn't change shape and toggling the <Ins> key or <Ctrl-v> does not cause **INSERT** to appear, then your MS-DOS system is not currently using the ANSI.SYS device driver as we understand it should be employed on an MS-DOS system. Check with your dealer or computer manufacturer to see if it is available for your system.

What is ANSI.SYS?

As noted, the ANSI.SYS file is a device driver. When in use it resides in a small area of memory and converts ANSI standard escape sequences to the PC's video RAM and keyboard environment. A version of ANSI.SYS is shipped with most computers running Microsoft's MS-DOS 2.n operating system, including the IBM PC. Your DOS manual should be seen for complete information on CONFIG.SYS (described below) and ANSI.SYS.

ANSI.SYS opens to mixed reviews
To date our experience with ANSI.SYS on various MS-DOS systems has been somewhat disappointing. We have found that the ANSI standard is not much of a standard, since manufacturers implement various sub- or super-sets of ANSI.SYS to meet their own needs. Some ANSI.SYS implementations, for example, will allow our 25th line menu prompts. Other, more irritating, problems have also occured. The result is that we are developing an installation program for MS-DOS systems which we hope to have ready shortly.

Installing ANSI.SYS
If the RUN/C screen does not appear to operate properly on your MS-DOS system and if you have an ANSI.SYS file supplied with your computer, you might install it to see if this resolves the problem. This can easily be done by inserting the line:

```
device = ANSI.SYS
```

in your CONFIG.SYS file. If no CONFIG.SYS file is currently in use with your system, create one and insert the above line as the only line in the file and save it on your system disk.

Now reboot the system and re-test RCP.EXE. (Remember, if you are using an IBM PC or compatible no change will be noted.)

Alternatively, we have found that RUN/C works on some MS-DOS systems only after ANSI.SYS has been removed from the CONFIG.SYS file. Experiment.

Command Line Options

When RUN/C is invoked there are several *command line options* which can be specified. Certainly the most familiar command line option is one which specifies the name of a program file to be immediately loaded and run. For example, if RUN/C is invoked with the command:

```
A>RCP HELLO
```

the current default drive/directory will be searched for a file named HELLO.C. If it is found, the program will be loaded and run. If there is no such file, RUN/C will display the message:

```
Program hello.c not found
Ok
```

Other command line options include changing the size of the program stack (described below), the maximum number of lines

that RUN/C can interpret and the maximum number of **#defines** that can occur in an interpreted program.

While these switches allow you to increase the processing capabilities of RUN/C, their primary purpose is to provide for situations where memory is in short supply. If the amount of memory available to your systems is insufficient for RUN/C and your program to comfortable reside together (for example, if you are using DOS 3.0 on a 320k system), you can gain additional space for your programs by reducing the maximum number of lines in the program, the maximum number of **#defines** and the stack size to, say, 5,000-10,000 bytes.

Let's discuss each command line options:

Modifying the maximum number of program lines
The number of program lines can be changed by invoking RUN/C with a command like:

RCP -L*nnnn*

This changes the maximum number of lines to *nnnn*. As of this writing the default is 2,000, and the specified number must be in the 100 to 9,999 range.

Modifying the maximum number of #defines
You can modify the maximum number of **#define**'d terms with:

RCP -D*nnn*

This sets the maximum number of **#defines** in a program to *nnn*. The default is 50, but the maximum number of #define's can be between 10 and 999.

(The reason for the maximums is that a small amount of memory is preallocated for each possible line or **#define** in the running program -- nearly 200 bytes for each **#define** and about 8 bytes for each line.)

Changing the size of the program stack
The stack size or working space for your program is automatically set at 25,000 bytes, which is more than sufficient for most programs designed to be interpreted by RUN/C. It is possible, however, to adjust the stack size upward or downward with a command line option. For example if RUN/C is invoked with the command:

A>RCP =5000

The stack will be set to 5,000 bytes. On the other hand, the stack

could be adjusted upward to a maximum of approximately 65,000 bytes, provided that your system has that much additional memory available.

There is an inverse relationship between the size of the stack and the amount of space available for your program. Thus increasing the stack from the standard 25k to 64k would reduce the available user space by approximately 37k. Happily this is no problem for systems with plenty of memory (greater than 320k), since there will already be more than 64k of working space available for the user's program. In fact, all other things being equal, increasing the stack to the maximum is recommended, particularly if your program does much *recursion*, a programming technique in which a function repeatedly calls itself.

Why would someone want to adjust the stack downward? For one thing if the stack were smaller, it would be possible to use RUN/C on a system with somewhat less than 320k of RAM. Another possibility would be if the program used many arrays -- which take up system memory but not memory from the stack. The danger of adjusting the stack size too low is that if a program runs out of stack space the message:

```
STACK OVERFLOW ERROR
```

will be generated and control will return to DOS. If such a situation occurs the RUN/C program in memory will be lost. In addition, there is no guarantee that opened files will be closed properly. This can be very inconvenient if you have failed to save a copy of your program and frustrating even if you have. Thus the size of the stack should be adjusted downward with care. If you attempt to create a stack larger than 65,000 bytes the message:

```
Invalid stack size
```

will be generated and RUN/C will not load.

Permanently changing the defaults

If you wish to change these maximums on a continuing basis, consider creating a batch file containing something like:

```
RCP =10000 -1250 -d10
```

If you invoke RUN/C with these command line arguments the maximum number of lines in the program will be set at 250, the maximum number of **#define**'s at 10 and the stack at 10,000 bytes -- smaller, but still sufficient for most programs. On our system with these command line flags set we gained an additional 24,840 bytes for our programs.

The command line changes will remain in effect as long as RUN/C remains loaded into memory. For your convenience this batch program is on the distribution disk under the name RUN.BAT.

Redirected Screen Output

The final command line option is use of DOS's redirected I/O. An **>outfile** option which names a file or device to which the screen output is to be assigned can be used. The file or device name must be immediately preceded by a > ("greater than") sign. Valid device names include:

Device name	Device
CON:	Console
PRN:, LST:, LPT:, LPT1	Printer
AUX:, COM:, RDR:, PUN	Auxiliary Port
NUL:, NULL:	Null

An example of invoking RUN/C with this option is:

```
A>RCP HELLO >CAPTURE
```

This will direct all screen I/O into the disk file named CAPTURE. The only disadvantage to this technique is that none of the screen output (including the keystrokes you type) will be visible to you, since it is all being redirected. Therefore you will be "flying blind". If you do try this and lose track of where you are, here is a bit of "radar" which will get you back to DOS:

Press <Ctrl-c> twice, 'E' once, <Esc> twice and type:

```
SYSTEM
```

You should return to DOS and all screen I/O will be saved to CAPTURE.

Immediate Mode

RUN/C offers an Immediate mode for using built-in functions. For example the command:

```
Ok
@ printf("hi joe\n");
```

will print:

```
hi joe
```

on the screen.

The '@' sign is the signal to use the Immediate mode. The

Immediate mode is a useful way of testing out calling sequences for many of the functions built into RUN/C. For example the function **atoi** can be exercised with this Immediate mode line:

```
@ printf("integer number is %d", atoi("-1234TEST"));
```

which prints:

```
integer number is -1234
```

Although **atoi** is technically looking for a character pointer, a literal string will suffice both for Immediate mode and in C program statements.

The Immediate mode is described in Section 4 under the symbol '@'. It currently is the third key term described in that section. In addition see **VAL** in Section 4 and Section 6 for more suggestions regarding debugging techniques.

Reserved words

The following C keywords are reserved in RUN/C for use according to their standard C definitions:

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

RUN/C command names are not keywords. Moreover, RUN/C command names (with the exception of **UNLINK**) cannot normally be used within C programs with the significant exception of the commands which can be called from programs using the RUN/C Professional debugging function **runc()**. Commands are listed in **CLIST** in Section 4. The most current listing is on-line under the **CLIST** help command.

Built-in C functions are also reserved keywords. These words (such as **printf**) cannot be used within RUN/C programs except to invoke the functions they name. All the built-in functions are listed in Section 4 under **FLIST**. However, the most current

---

* Reserved but not implemented as of this writing.

listing is on-line under the **FLIST** help command.

In addition, **FILE** is a predefined **typedef** that references available file I/O structure. Any attempt to redefine any of key terms will likely result in an error message.

Input/Output

This section describes input/output (I/O) in RUN/C.

Files

A file is a collection of data which is either kept on a floppy disk, a fixed (hard) disk, or temporarily in a special set-aside section of random access memory, frequently called as a RAM disk. For a RUN/C program you have written to use data in a file, you must properly open the file before doing anything else.

File pointers

RUN/C allows read and write access to files using both character stream (primitive) I/O, known as *Level 1* I/O, and buffered I/O, known as *Level 2* I/O. These latter operations are performed using an object called a file pointer (a pointer to an object of type **FILE**). This file pointer is calculated by the function **fopen**, declared by the user, and then can be used with functions like **getc, putc, fgets**, etc. See **FILE** in "Commands, Keywords and Functions," Section 4.

File names

The physical file is described by its file specification, (*filespec* for short). This is a string expression of the form:

drive:filename.ext

The optional drive name (always followed by a colon) tells RUN/C where to look for the file. You do not need the drive name if the file you wish to access is located on your *default drive.*

The file name is composed of two parts. A *primary name*, which may be up to eight characters in length and a optional secondary name (known as the *extension)* which may contain up to three characters. The primary and secondary names are always separated by a period. The filename may contain a single period separator and any combination of alphanumeric characters (A through Z; 0 through 9). No distinction is made between upper and lower case characters. In addition, the following symbols may be used in file or pathnames:

@ $ & # @ ! % ' ` ( ) { } - _ ^ ~

Other non-alphanumeric characters such as the greater than (>), less than (<), asterisk (*), slash (/), backslash (\), question-mark (?) and vertical bar (|) are reserved by DOS and cannot be used in a file or directory name.

As mentioned, many RUN/C commands automatically open files. For example, the command:

```
Ok
LOAD B:HELLO.C
```

will load the file HELLO.C into memory if it can be found on the B> drive. There are actually several equivalent forms of this command:

```
LOAD "B:HELLO.C
```

or

```
LOAD "B:HELLO.C"
```

or

```
LOAD B:HELLO
```

The last example works because the extension .C is assumed unless otherwise specified. Note: If you wanted to access a file named simply HELLO the command:

```
LOAD B:HELLO.
```

would be used.

As you can see, the quotation marks are optional, although the first quotation mark must be preceded by a space. Also see **LOAD** in Section 4.

Pathnames

Under DOS 2.0 and greater, a *pathname* can be placed in front of the name of the file. This optional capability is very useful; it allows you to chart a path to a file through a series of directories and subdirectories. A *path* consists of a list of directory names separated by backslashes (\). The path is a string expressed in the form:

drive:path\subpath

If the file you seek is in the current or default drive, it is not necessary for you to specify the drive. Similarly, if the file you

seek access to is in the currently logged-on directory or subdirectory, it is unnecessary to specify the path.

Generally speaking, RUN/C provides full pathname support. Pathnames can be used any time a file is accessed or referenced either from the RUN/C command line or within a program being interpreted. If a filename is included, it must also be separated from the last directory name by a backslash. If a path begins with a backslash, RUN/C starts its search from the root directory; otherwise, the search begins at the current directory. Thus if you are in the directory CFILES and want to access a file in B-ENTRY subdirectory, the command:

```
LOAD B-ENTRY\FILE1
```

would be valid.

Currently it is not possible to view the names of existing subdirectories with RUN/C's **FILES** command. An alternative means of doing this is to use the **SHELL** command:

```
SHELL DIR\
```

to see all the files, including subdirectory entries on the root drive/directory.

Current Drive/Directory
The term *drive/directory* is used throughout this manual to refer to the current -- or default -- drive and directory. This is the drive directory combination which RUN/C (and other DOS programs) will search if you enter a filename without specifying a drive, directory or subdirectory. The current drive/directory can be changed by issuing a **SHELL** command such as:

```
Ok
SHELL CD\RECORDS\JOE
```

Device names
In addition to allowing keyboard input and screen output, RUN/C also makes available three special file pointers **stderr, stdout** and **stdin.** These are described in Section 4 under their names.

A printer attached to LPT1: can be accessed via the **LLIST, lprintf** and other printer-related commands and functions. **SETCOM** function may be used to reset the **aput** and **aget** asynchronous communication functions and the DOS MODE.COM command can be used to direct output from LPT1: to a serial printer or another parallel output device.

Using the Screen

Currently RUN/C supports both the color and monochrome 80-column text modes offered by the IBM PC and MS-DOS systems. The PC's special character set (decimal 128 and above) may be accessed via standard C functions such as:

```
putchar(156);
```

which will print a British pound sign on the IBM PC.

Printer support

If a printer is not properly attached to your computer and available (on-line), various types of error messages will be generated if you try and print something from within RUN/C. If you attempt to print through one of RUN/C's commands such as **LLIST** or by activating the <Ctrl-p> (or on PC compatibles <Ctrl-PtScr>) printer echo the error message:

```
Printer not ready
```

will be generated. However any attempt to access a printer which is not online with a RUN/C built-in function such as **lprintf** will call forth a DOS error message such as:

```
Write fault error writing device PRN
Abort, Retry, Ignore
```

If the condition can be corrected, you can use the **R**(Retry) option to continue with your program. Otherwise the operation must be aborted and you will be returned to DOS.

Math chip support

The 8087 (or 80287 if you have an AT-type system) math chip is automatically sensed by RUN/C if present. Accordingly, programs using transcendental math functions run faster if the 8087 is present. If you are using a math chip keep in mind that many PC's and XT's (if not all) and other MS-DOS computers have an internal hardware switch indicating whether or not a math chip is present. We have found that if the switch is not properly set, the math chip will be found by some programs, but not by others. Check your operations manual, or call your dealer for more information.

SECTION 2. EDITING RUN/C PROGRAMS

Overview

RUN/C has three primary levels of operation. The first is the Command level. A second, accessible only from the Command level, is the Edit level, RUN/C's built-in screen editor. The third level, also accessible from the Command level, is the **SHELL**, which allows access from RUN/C to programs and commands that are usually run from the DOS command line.

We shall first discuss the Edit and Command levels. Then we will consider the use of the external **EDITOR** command and the more general **SHELL** command for sophisticated text editing and other tasks.

Important notice
Commands, keywords and functions briefly mentioned in Sections 1-3 are generally described in detail in Section 4.

Command and Edit Levels
When RUN/C is first invoked, it is in the Command level. An example of calling RUN/C is given at the beginning of Section 1.

The screen prompt:

```
Ok
 *
_
```

signifies that RUN/C is waiting for a command. The Command and Edit levels are quite distinct; you will have no trouble telling which is active. In Command level the **Ok** prompt will always be just above the cursor. In Edit level, the **Ok** prompt will be replaced by an entirely new set of visual cues: a editor command area occupying the top three lines of the screen separated from the rest of the screen by a horizontal double line.

The Command and Edit levels are mutually exclusive. From the Command level you can give commands to the interpreter, but you cannot directly edit or insert text into your program. In Edit

---

* The underline in this and subsequent examples symbolizes the blinking screen cursor.

level you cannot issue commands to the interpreter or run your program.

Going back to our command prompt, let's put RUN/C through some paces. If you have a copy ready to run, you may want to follow along:

```
Ok
LOAD HELLO*

Loading line 7

Ok
LIST

        1:main()
        2:
        3:{
        4:
        5:   printf("Hello, world!\n");
        6:
        7:}

Ok

EDIT
```

```
hello.c                                INSERT  Line: 1  Col: 1
                        (Exit edit: ESC or F10)  (Help: ^J or F1)

main()

{
        printf("Hello, world!\n");
}
```

First we loaded a program from the disk named "HELLO.C". We didn't need to use the ".C" extension (for more information on the naming of files see Section 1) because it is the default extension and is automatically assumed if you do not specify an extension name.

---

* We have tried to consistently use capital letters for commands, but RUN/C will accept lower-case commands with equal enthusiasm.

Then we listed the program using the LIST command.

Finally, we began editing the program with the EDIT command. The EDIT command or, simply, E issued at the command line transfers program control to RUN/C's full screen editor and places the cursor at the beginning of the line.

Of course you can begin editing anywhere in your program by selecting another line number such as line 4:

```
E 4
```

which will place the cursor at the beginning of line 4. If you want to begin editing at the end of your program simply type:

```
E END
```

If a message has appeared which references a particular line number such as:

```
Error #21 at line 7
Variable or function not found
```

you can begin editing at that particular line by invoking the editor with:

```
EDIT ERROR
```

See the discussion of the EDIT command in Section 4 for more information on this subject.

About the RUN/C Editor

It's a cliche to say that editors are very personal -- or at least they are if you personally dislike the one you are using. From the onset it was clear that we couldn't create everyone's favorite editor (although we've made some provisions for that with our EDITOR command) so we tried to pick and choose some familiar and time-tested features.

What we ended up with is a cross between a Wordstar(tm) front end and a powerful full-screen editor resembling Phoenix Software's PMATE with a little bit of the BASIC editor thrown in for good measure. If you are used to the PC's cursor control keys and/or the WordStar control key system you will feel right at home with our editor. If you are used to an entirely different system, our optionally obtrusive on-screen help will make mastering the RUN/C editor a 20-minute or so project.

After loading hello.c and going into the editor, the top three screen lines will be something like:

```
hello.c                                INSERT  Line: 1  Col: 1
                    (Exit edit: ESC or F10)  (Help: ^J or F1)
```

You will always be able to get information about the file you are editing: its name, if any, whether you are in INSERT or OVERWRITE mode, the line and column location of your cursor and the reminder that if you press the <Esc> key (or, alternatively, <F10> if you are using an IBM PC or a close compatible) you will leave the editor and return to the RUN/C command mode. Another screen note indicates that there is help at hand if you press <Ctrl-j> (or <F1> on a PC).

Help Screen #1 -- Cursor Movement

Let's take a look at the first level of screen help by pressing <Ctrl-j>. (Note: We have written this text so that you can follow along on your computer. Hopefully it won't be too unbearable to read even if your PC isn't present.)

```
hello.c                    INSERT   Line: 1 Col: 1
                    (Exit edit: ESC or F10)  (Help: ^J or F1)

Help #1                 -- Cursor movement --  (more HELP: ^J or F1)
<- or ^S char left          ^ or ^E line up        PgUp or ^R previous page
-> or ^D char right      v or ^X line down         PgDn or ^C next page
^<- or ^A word left      Home or ^QS line begin  ^PgUp or ^QR top of prog.
^-> or ^F word right     Home or ^QD line end    ^PgDn or ^QC end of prog.
```

A quick primer

Perhaps the easiest way to learn to use the editor and to understand its various functions is to edit our HELLO.C program referring when needed to the Help menu. First we will edit only one line, then the full page editing commands will be described.

Let's edit line 5, our most complex line by moving the cursor down to line 5:

```
hello.c                              INSERT   Line: 1  Col: 1
                          (Exit edit: ESC or F10)  (Help: ^J or F1)

main()
 (
        printf("Hello, world!\n");
 )
```

There are several elements in this sample to consider. The underline under the "p" in **printf**, as noted previously, represents the position of the screen cursor, the blinking box or underline that is visible on your computer's screen when RUN/C is running. Although the line number is not present as it is when a program is **LIST**'ed, the line number is displayed on the top line of the screen as shown.

Screen lines can extend up to 235 characters although only 80 characters can be shown at one time. If the line extends beyond

the last screen column a plus sign (+) will be shown in rightmost column. Lines do not "wrap" around.

Now let's edit the line. If the <Right Arrow> key is pressed twice, the line will appear*:

```
printf("Hello, world!");
```

If the <Left Arrow> is pressed once, the line will appear:

```
printf("Hello, world!");
```

If <Ctrl-Right Arrow> is pressed twice, the line will appear:

```
printf("Hello, world!");
```

If <Ctrl-Left Arrow> is pressed once, the line will appear:

```
printf("Hello, world!");
```

If <End> is pressed, the line will appear:

```
printf("Hello, world!");_
```

If <Home> is pressed, the line will appear:

```
printf("Hello, world!");
```

Changing a Line

There are many ways to make changes in a line. In all cases the first thing to do is simply to move your cursor to where you want the change to begin. Next decide whether you wish to insert text or to overwrite what is currently there.

If you wish to insert text at your current position simply begin typing. If you want type over what is currently there, press the <Ins> key or <Ctrl-v> keys to toggle Overwrite Mode. The mode you are in will be displayed at the top of the screen.

Here is our sample line yet again:

```
printf("Hello, world!");
```

Since the cursor is a blinking underline, we know we are in Overtype Mode. If you press <Ins> and type "hi " the line will appear:

---

* Although we are describing text editing using the PC's numeric keypad control keys, WordStar compatible control keys -- described below -- can be used equally well.

```
hi printf("Hello, world!");
```

If you move the cursor to the beginning of the program line by pressing <Home>, press the <Ins> key and type "bye "; the line now appears:

```
bye hi printf("Hello, world!");
```

Deleting Characters
If you wish to delete some characters from the line, move the cursor to the appropriate point on the line. For example, if you wish to remove the "bye " from the current line, an easy way to do it is to use the <Home> key followed by four taps on the <Del> key. Now your line appears:

```
hi printf("Hello, world!");
```

If you are typing along and realize that you have made a mistake, a quick fix is to press the <Backspace> key. For example, if you have inserted the word 'big' in your line, but spelled it wrong as in:

```
hi printf("Hello, bii world!");
```

you can delete the second "i" by pressing the <Backspace>. Then typing 'g' will give you:

```
hi printf("Hello, big world!");
```

To delete from the current cursor position to the end of the line, you can use the <Ctrl-qY> combination which in the case of your sample line would give you:

```
hi printf("Hello, big
```

To complete the massacre you can type <Ctrl-q<Del>> which will delete from the current cursor position to the beginning of the line, leaving us with a rather blank:

```

```

Similarly, you can move forward or backward through you text using the up and down arrow keys or <Ctrl-r> and <Ctrl-c>. The function of all the editing keys are described below.

Autoline Numbers
Whenever one or several lines are inserted or deleted, all subsequent lines in the program are automatically renumbered as shown dynamically through the line counter in the top line in the menu. Note that if you try to edit a nonexistent line you will

enter the editor and the cursor will be placed just after the last line if the file.

Unlike Microsoft and PC BASIC, RUN/C automatically assigns line numbers. Line numbers are never entered by the RUN/C user directly into the program. Nor is it possible to edit or delete the autoline number when editing a line. In fact, the only times you will see a line number referenced directly are when certain types of errors occur or when a **LIST, LLIST, NLIST** or **NLLIST** command is used.

Using Editing Keys

If you are actually editing this program, you will find that you can move the cursor to the beginning or end of a line, to the top or bottom of the screen and/or the entire text, forward or backward by character or word, and to insert, delete or overwrite text as you see fit, using the cursor control keys found on the IBM PC's keyboard.

As with most text editors and word processors, you must use control keys and/or specially marked function keys to make changes in the text. Function key assignments generally follow those of PC BASIC. RUN/C control key assignments are a subset of those used by WordStar(tm).

The following shows RUN/C function key and control key assignments. Function key assignments generally have control key counterparts for those who prefer using control keys, for those using MS-DOS with ANSI.SYS (discussed in the previous section) and for those using terminals without the IBM PC's specially marked function keys. The editing keys are active in both Command and Edit levels.

Key(s)	Function
Home **<Ctrl-qS> ^QS**	Moves cursor to the beginning of the logical program line.
End **<Ctrl-qD> ^QD**	Moves cursor to the end of the logical program line.
<up arrow> **<Ctrl-e> ^E**	Moves the cursor up one row. If the cursor is at the top of the program, this command will have no effect.
<down arrow> **<Ctrl-x> ^X**	Moves the cursor down one row. If the cursor is at the bottom of the program, this command will have no effect.

<left arrow> **<Ctrl-s> ^S**	Moves the cursor one position "backward" toward the beginning of the line. If the cursor is at the beginning of the line, it will move to the end of the previous line.
<right arrow> **<Ctrl-d> ^D**	Moves cursor one position "forward" toward the end of the line. If the cursor is currently at the end of the line, it will move to the first column of the next line.
Ctrl-<left arrow> **<Ctrl-a> (^A)**	Moves the cursor to the beginning of the word to the left of the current cursor position on the logical line. A word in this context is defined as a character or group of characters which begins with a letter or a number. Words are separated by blanks or special characters such as punctuation marks. Thus the effect of this command is to move the cursor backward past the preceding group of non-alphanumeric characters, if any, to the first character of the next group of alphanumeric characters. (See examples below.) If the cursor is currently at the beginning of the line, it will move to the beginning of the last word in the previous line.
Ctrl-<right arrow> **<Ctrl-f> (^F)**	Moves the cursor forward to the beginning of the next word to the right of the current cursor position on the logical line. If the cursor is already at the last word in the line, it will move to the beginning of the first word in the next line.
<Home> **<Ctrl-qS> (^QS)**	Moves the cursor to the beginning of the current line. If you are already at the beginning of the line this command will have no effect.
<End> **<Ctrl-qD> (^QD)**	Moves the cursor to the last column of the current line. If you are already at the end of the line this command will have no effect.

PgUp **<Ctrl-r> (^R)**	Moves the cursor to the previous screen page. If you are at the first screen the cursor will move to the top of the screen. If the cursor is already on the top line this command will have no effect.
<PgDn> **<Ctrl-c> (^C)**	Moves the cursor to the next screen page. If you are already at the bottom of the text in memory the cursor will move to the last line in the program. If the cursor is already on the bottom line of the file this command will have no effect.
Ctrl-<PgUp> **<Ctrl-qR> (^QR)**	Moves the cursor to the top line of the program. If you are already at the beginning of the program this command will have no effect.
Ctrl-<PgDn> **<Ctrl-qC> (^QC)**	Moves the cursor to the bottom line of the program. No effect if you are already at the end of the program.

HELP Screen #2 -- Delete and Miscellaneous

By pressing <Ctrl-j> or <F1> again we can get the second of three help screens:

```
hello.c                                    INSERT  Line: 1  Col: 1
                                (Exit edit:  ESC or F10)  (Help: ^J or F1)

Help #2                  -- Delete and Misc --                (More HELP: ^J or F1)
BkSp      Del char left    ^QDEL   Del to beg-of-ln   ^QF    Search
DEL or ^G Del char         ^QY     Del to end-of-ln   ^QA    Search & Replace
^T        Del word right   ^N      new line           ^L     Search/Rep Again
^Y        Del line         INS or ^V  Ins/Overwrte    ^U     Cancel op or msg
```

This menu describes a number of keyboard commands to delete text and to search for text. In addition, several miscellaneous commands are described.

<Backspace>
<Ctrl-h> (^H)

Deletes the character to the left of the cursor. All characters to the right of the deleted character and the cursor move left one position. The cursor also moves left one position. If the cursor is at the beginning of the line, then backspace will delete the carriage return between the current line and the previous line. This will cause the current line to append itself to the end of the previous line.

<Del>
<Ctrl-g> (^G)

Deletes the character at the current cursor position. All characters to the right of the deleted character move left one position. The cursor remains in its current position. If the cursor is at the end of the line, then <Del> or <Ctrl-g> will delete the carriage return between the current line and the next line. This will cause the next line to append itself to the end of the current line.

<Ctrl-t>

Deletes all characters from the current cursor position up to and including the next white space or non-alphanumeric character. This command will not delete text beyond the current line.

<Ctrl-y>

Deletes the entire line the cursor is currently on and places the cursor at the beginning of the next line.

<Ins>
<Ctrl-v> (^V)

Toggles the Insert and Overwrite modes. The word **INSERT** or **OVERWRITE** will appear at the top of the screen, indicating which writing mode is active.

Whenever you enter the RUN/C editor it will be in Insert Mode. In this mode any character which is typed will be inserted at the current cursor position. The character which had occupied that position, the cursor and all characters to the right of the inserted character are moved to the right, making room for the new character.

In Overwrite mode, any character typed will be placed in the line at the current cursor position. If a character is currently in that position, it will be overwritten.

The first time the <Ins> key or <Ctrl-V> is pressed Overwrite mode is selected.

<Ctrl-q>F (^QF)

This is the search or "find" command. When invoked with our sample program this top portion of the screen will appear:

```
hello.c                INSERT   Line: 1 Col: 1
                  (Exit edit: ESC or F10)  (Help: ^J or F1)
Find: Hello,
```

In this case we have supplied the search string: **Hello** followed by <enter>. If the string is found, the cursor will move to the beginning of the search string. If the string cannot be located, the cursor will remain in its current position and the last command line will be replaced with:

```
hello.c                    INSERT   Line: 1 Col: 1
                  (Exit edit: ESC or F10)  (Help: ^J or F1)
hello, not found  (Hit ^U to continue)
```

To abort a search or search-and-replace operation or remove the search string prompt from the command line press <Ctrl-u>.

There are several things to keep in mind when using the search or search and replace features (described below) of the editor.

* All searches (and replacements) are forward. That is, from the current cursor position toward the end of the file. If you want to search for something "behind" your cursor go to the top of the program with <Ctrl-PgUp> and commence the search.

* Searches (and replacements) are case sensitive. Thus if we had looked for "hello" instead of "Hello" the string would have been found.

* Searches (and replacements) must be completed on a single line. In other words, it is not possible to search for a carriage return or, in fact, any control or high-bit (above 127 decimal) character other than <Tab>.

* Search strings are limited to 24 characters and replace strings to 21 characters.

* Once a search (or replace) has begun the search string or search/replace string combination will remain on the command line until (1) a **string not found** message appears and you press <Ctrl-u>, (2) you press <Ctrl-u> to cancel the command, (3) another search sequence is initiated, or (4) you exit the editor.

<Ctrl-qA> (^QA) This is the search-and-replace command. It is quite similar to the "find" command, described above. However, after giving the search string you will be prompted to provide a replacement string. When the <Ctrl-qA> is invoked with our sample program this top portion of the screen will appear:

```
hello.c                    INSERT   Line: 1 Col: 1
                  (Exit edit: ESC or F10)  (Help: ^J or F1)
Find: Hello,       Replace: Good-bye, Confirm (Y/N) Y
```

If the string is not found a message to that affect will be displayed on the third command line.

If the string is located the cursor will move to the string and the query:

```
Confirm (Y/N) _
```

will appear on the right hand side of line 3. If you answer with a 'Y' the string will be replaced. If you answer with a 'N' you will be returned to edit mode at the current position. If you want to abort a find-and-replace operation press <Ctrl-u>.

In addition to the points noted under <Ctrl-qF>, above, the following applies to all find-and-replace operations using the RUN/C editor:

* Replacement strings are limited to 21 characters. Neither control codes (such as a <enter>) or high-bit characters (above 127 decimal) will be accepted. The only exception is the <Tab> character.

* The <Ctrl-l> command is available for the most recent find-and-replace operation even if the command line has been cleared with <Ctrl-u>.

<Ctrl-l> (^L)

This keyboard command repeats the current search or search-and-replace operation which is shown on the last command line. Using <Ctrl-l> you can work your way through the entire file until the:

```
string not found
```

message appears.

The <Ctrl-l> command is available for the most recent find-and-replace operation even if the command line has been cleared with <Ctrl-u>.

<Ctrl-u> (^U)

Use this command to cancel an operation such as a two-character command when you have not yet typed a second character, e.g. after typing <Ctrl-q> or <Ctrl-k>. <Ctrl-u> will also cancel a command that requests an answer such as search or replace or read and write file.

<Ctrl-n> (^N)

Inserts a blank line at the line the cursor is currently on. The current line and all subsequent lines are moved downward and the cursor does not move.

<Ctrl-q><Del>

Deletes all characters from the cursor position to the beginning of the line. (This is a <Ctrl-q> followed by pressing the <Del> key.) If the cursor is already at the beginning of the line this command will have no effect.

<Ctrl-q>y (<^Qy>)

Deletes all characters from the cursor position to the end of the line. If the cursor is already at the end of the line this command will have no effect.

<Esc>
<F10>

Returns to the Command mode. If AUTOSAVE is active and set to 2 (see the discussion of SAVE and AUTOSAVE is Section 4) the text will automatically be saved to disk under the name PROGRAM.C. Otherwise it is up to the user to save text after leaving Edit mode. (Alternatively, the entire program can saved by marking its first through last

line and writing the block to disk. This is described under <Ctrl-w> below.)

<enter>
<Ctrl-m> (^M)

In Move the cursor to the beginning of the next line. If you are in Insert mode and the cursor is in the middle of a line, all text after the cursor is inserted as a new line. Use this technique to break up a line into one or several parts.

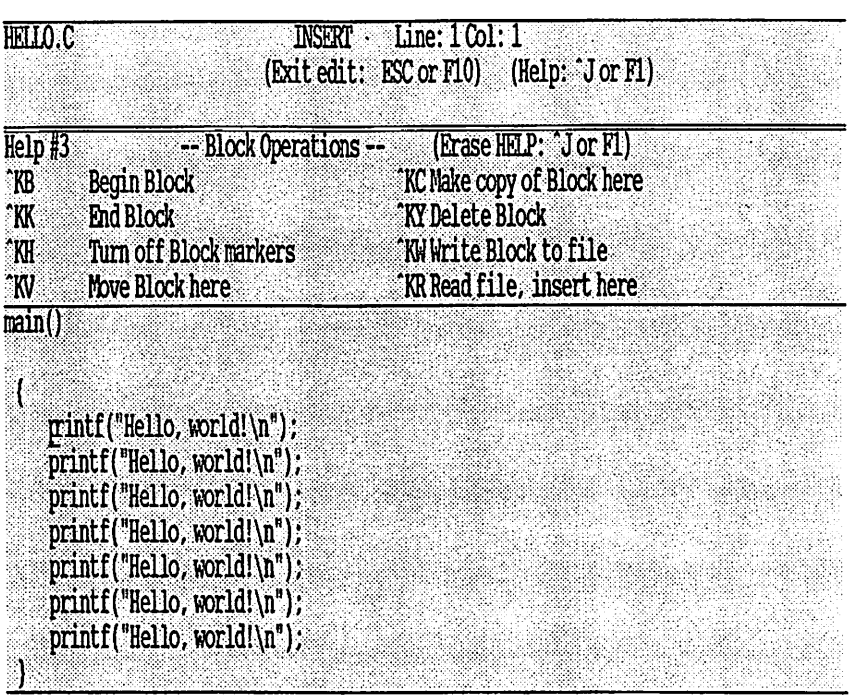
If the cursor is already at the end of the line, the cursor will be moved to the beginning of the next line which will be blank. <enter> is also used to terminate strings in search and replace operations.

<tab>
<Ctrl-i> (^I)

Moves the cursor to the next tab position. In Overtype mode the cursor is moved over any existing text. In Insert mode all the following text will be moved forward to the next tab position. Tabs are adjustable. The default setting is every five spaces beginning with column 1. Thus tab positions are columns 0, 5, 10, etc. See the TABS command, Section 4, for information on setting tabs.

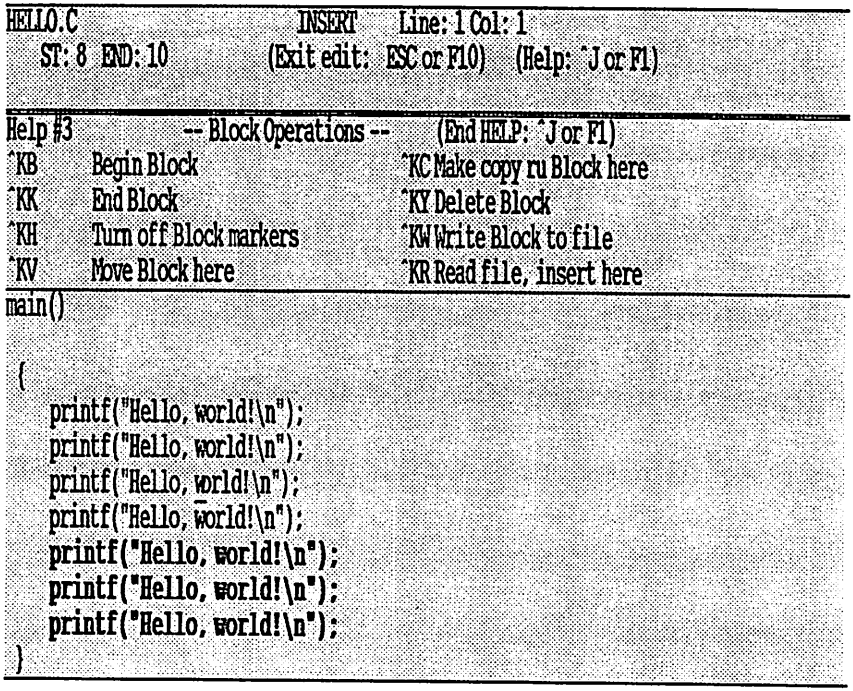
HELP Screen #3 -- Block Operations

Now let's look at help menu No. 3 which describes block operations:



Notice that the help message now suggests that pressing <F1> or ^J one more time will turn off the help menu -- something we're not ready to do yet. Notice also that we have changed our program, adding a large number of familiar lines.

We can begin marking our text by pressing the <Ctrl> key and the K simultaneously followed by a B. This is represented on our menu as **^KB.** (Actually you can press a capital B, a lowercase b or a ^B -- we aren't picky). Once you have marked the beginning of the line the only visible effect will be the **ST: 8** on the second command line indicating that the block starts at line 8. We move the cursor down two lines and to the middle of the line and type **^KK,** the end block command. Now **END: 10** shows up on the second statistics line. Thus the beginning and end lines of the block are reported, as shown in the screen below. In addition, on most systems highlighting will appear.



Notice that the blocks are entirely line oriented. Note: If you only want to mark a single line execute begin (<Ctrl-kB>) and end block (<Ctrl-kK>) operations without moving the cursor. Even if your cursor is in the middle of a line, the block will begin on the first column of that line and/or end on the last column.The only way to move part of a line is to first break it up using the <enter> key in Insert Mode.

Once a block has been defined it can be expanded or contracted by using the block commands either above, below in inside the defined block. Even if you leave the editor and return, the marked block will remain. The only thing you can't do is define a block end before a block beginning. If you try to do that only the last command you gave will be active.

To remove the block markings simply type <^Kh>.

There are a number of other useful block-related commands:

^KV Insert block just above the current cursor line

^KC Copy block just above the current cursor line

^KW Write block to a file

^KR Read a file from disk, inserting it just above the current cursor line

^KY Deletes block

Although a block can always be copied, it is not possible to move a block into itself (the cursor cannot be inside a block when an attempt is made to move it). Attempting to do so will result in the message:

```
Can't move block onto itself (hit ^U to continue)
```

Let's look at the final two block-oriented commands:

When <^Kw> is executed the following change will occur in the command menu:

```
HELLO.C                    INSERT   Line: 1 Col: 1
                  (Exit edit: ESC or F10)  (Help: ^J or F1)
File to write: myfile.c
```

After you have supplied the exact filename and pressed <enter>, the editor will write the block out to the named file and, if successful, report the name of the file and the number of lines written to it, as in:

```
test.c written (3 lines
```

If you attempt to write (<^KW>) with no block marked the error message:

```
Start or End block not set
      (Hit ^U to continue)
```

You must press <^U> (the editor reset command) to restart the editor.

You can also add or merge* text from a disk file into your text on

* For readers familiar with the RUN/C version 1.n this command replaces the MERGE command.

screen using the <^Kr> command. After the named file has been read the command area will appear as:

```
HELLO.C                    INSERT    Line: 1 Col: 1
                    (Exit edit: ESC or F10)  (Help: ^J or F1)
File to read:: myfile.c
```

where *nnn* represents the number of lines read by the editor. If the file you are trying to read cannot be found on the appropriate drive/directory, the message:

```
Cannot find the file: test.c (Hit ^U to continue)
```

EDITOR and SHELL

The commands **EDITOR** and **SHELL** allow you to perform functions and tasks which are usually only accessible from the operating system. The basic requirements in using EDITOR and SHELL are:

1. That you are using DOS 2.1 or higher.

2. That DOS's COMMAND.COM be on the drive directory pointed to by DOS's COMSPEC. Use DOS's **SET** command to determine the currently designated location of COMMAND.COM. Please refer to your DOS manual for information on resetting the COMSPEC if the current setting is not satisfactory.

3. That your system has sufficient memory for both RUN/C and your editor to be co-resident (we're talking greater than 384k in most cases).

Memory requirements

It is important to recognize that the amount of memory available to your system may limit the usefulness of **EDITOR** and **SHELL**. This is because when these commands are used RUN/C remains resident in memory, occupying more than 200k of RAM. This is no problem if your system has 512k bytes of RAM or more, but systems running less will have trouble with other large programs. The situation is further complicated by the recent proliferation of other co-resident utility programs such as Sidekick(tm), which can occupy 64k or more of memory.

A simple example: RUN/C Professional requires about 320k of RAM. If your editor requires, say, 128k, you will need nearly 512k to comfortably have the programs co-resident.

Memory problems

If there is insufficient memory available, efforts to use **EDITOR** or **SHELL** will not be successful. You may get a message telling you that there is insufficient memory for the operation you are trying to perform or something more spectacular may happen ...

If there is insufficient memory for both RUN/C and the program which EDITOR or SHELL is calling to be co-resident, but if there is sufficient memory to reload DOS's COMMAND.COM, the message:

```
Program too big to fit into memory
Press any key to return to RUN/C
```

will announce the failure of the attempt. When you press a key RUN/C will regain control of the system.

A more serious condition will occur* if there is insufficient memory to reload COMMAND.COM when executing **EDITOR** or **SHELL**. Generally the DOS message:

```
DOS ERROR 8
```

will appear. If such an error occurs you should reboot and avoid using **EDITOR** or **SHELL** until the amount of RAM available to your system has been increased. In addition to obtaining more memory (an increasingly affordable endeavor), consider reducing or eliminating the size of any RAM disks you may be using or eliminating other co-resident programs.

The EDITOR Command

Editors are quite personal and no matter how sophisticated an editor is, unless it is your editor, you are likely to consider it second best. Therefore RUN/C provides a "seamless web" (we have always wanted to use that term) which will automatically save your program, call up your favorite screen editor or word processing program, load your RUN/C program and allow you to edit it as you see fit.

When you have finished editing, you must save the program and leave your editor. RUN/C will then reappear and reload your newly edited program.

All that is needed to use this process is an editor or word processing program which has been (1) been renamed EDITOR.COM or EDITOR.EXE using standard DOS file renaming conventions or (2) specified with the **SET EDITOR** command or (3) called via an EDITOR.BAT file.

Renaming to EDITOR

In deciding whether to use the '.COM' or '.EXE' extension to **EDITOR**, check the name of your editor and use the same extension. For example, if the name of your program is WRITE.EXE you will rename it EDITOR.EXE with the command:

```
A>REN WRITE.EXE EDITOR.EXE
```

If the name of your program is WS.COM you can rename it

---

* Fortunately, it is not necessary to make this discovery by trial and error. Prior to using **SHELL** or **EDITOR**, try the **FRE** command. It will tell you how much memory is currently available on the system. If the number is within 5-10k of the size of your COMMAND.COM, no attempt should be made to use **SHELL** or **EDITOR**.

EDITOR.COM with the command:

```
A>REN WS.COM EDITOR.COM
```

That's all there is to it.

Using SET EDITOR

The **SET EDITOR** command allows you to use the editor or word processing program of your choice without having to create a double copy of your editor on your disk.

The command is invoked in the form:

SET EDITOR *filename*

where filename consists of the same filename you would use call up your editor from the DOS command line. The filename may include a drive or directory pathname.

An example might be:

```
SET EDITOR WS
```

which will call WS.COM* from your default drive/directory whenever you use the **EDITOR** command.

Another example is:

```
A>SET EDITOR EC
```

which would use the editor EC.EXE from the A: drive each time **EDITOR** is called.

Remember that **SET EDITOR** is set only from within the RUN/C command level -- not the full-screen editor. Currently **SET EDITOR** must be used each time RUN/C is invoked from DOS.

How EDITOR Works

When **EDITOR** is invoked as in:

```
Ok
EDITOR

Saving line n
```

---

* WordStar and other large word processing programs often have additional overlay files which also must be present on the default drive/directory.

the RUN/C program currently in memory is saved to the special file name:

```
PROGRAM.C
```

Program control then branches to **EDITOR.**

After you have edited your program, saved it (to the name PROGRAM.C), and exited your editor, the message:

```
Press any key to return to RUN/C ...
```

will appear. Go ahead, press any key and the message:

```
Loading line n
Back in RUN/C
Ok
```

will appear. Your newly edited program will be ready to **RUN** or for further editing.

Dealing with high bits
RUN/C reads only ASCII programs. Any "high bit" characters found in your program such as those automatically inserted in text created with WordStar's "document mode" are automatically reduced to the ASCII range 0-127 ("stripped") before RUN/C begins interpreting your program.

Other special codes which may be inserted by your editor or word processing problem may or may not present programs when RUN/C loads. (If you find any editors or word processing systems running under PC DOS or MS-DOS which are incompatible with RUN/C's **EDITOR** command please contact us so we can attempt to work around the problem or at least issue appropriate warnings.)

Managing PROGRAM.C
Since the PROGRAM.C convention is used by the **AUTOSAVE** command, as well as **EDITOR, SHELL, LOAD** and **SAVE**, it is very important to get into the habit of saving your program under the filename of your choice whenever you have a version which you want for posterity.

The SHELL Command
SHELL provides all the power of the **EDITOR** command but with fewer restrictions. **SHELL** can be used to unconditionally branch to the operating system or to call a particular program or DOS command and then, when the task is done, automatically return to RUN/C. Your program in memory will remain available when you return to RUN/C.

Unconditional Branching
When the SHELL command is issued by itself as in:

```
Ok
SHELL
```

The message:

```
Type 'exit' when done with external operation(s).
```

will appear just before control is passed to the operating system.

Once at the operating system level you can perform any operation you would normally perform: viewing the directory, changing the logged-in disk or directory, running other programs, even compiling a C program -- assuming that you have a C compiler and sufficient memory for the compilation process while RUN/C remains resident. (An estimated 512k of RAM is needed for most SHELL interpreter/compiler operations.) For an example of this technique see SHELL in Section 4.

After you have completed your command line operations, type EXIT as in:

```
A>EXIT
```

to return to RUN/C. The message:

```
Press any key to return to RUN/C

(Back in RUN/C)

Ok
```

will appear.

Branching to DOS and returning
When SHELL is invoked with a task, it will perform the assigned task (run the named program) and then return to RUN/C automatically.

Again, this might be as simple as:

```
Ok
SHELL COPY PROGRAM.C A:
```

which will copy PROGRAM.C to the A: drive.

Or the SHELL operation might mimic our EDITOR without having to rename your text editor or use SET EDITOR. Thus:

```
Ok
SHELL WS PROGRAM.C
```

will leave RUN/C and load the WordStar program with the file PROGRAM.C. The only difference is that when using SHELL, your program will not be automatically saved before calling the SHELL process.

When you have completed the operation and exit the editor, your will return to RUN/C with the message:

```
Press any key to return to RUN/C

(Returning to RUN/C)

Ok
```

You will then be returned to RUN/C where you may resume your programming.

SECTION 3. NUMERIC EXPRESSIONS and OPERATORS

Overview

As the name implies, the authors of C did not believe in using words where mere letters or symbols might do. The same is true of the types of symbols -- called *operators* -- required to perform common mathematics, uncommon mathematics and even some computer acrobatics!

In this section we will survey C's arithmetic operators, assignment operators, relational and logical operators, increment and decrement operators, bitwise operators and some miscellaneous operators and expressions including the indirection operator (*), address-of (&) operator, comma operator (,) conditional operators (? :) and structure and union operators (. and ->).

Lastly, a chart is provided showing the relative order in which operators are evaluated in the C programming language.

Arithmetic operators

Arithmetic operators -- also known as *binary operators* -- are symbols to tell the computer to perform the four common math functions: add (+), subtract (-), multiply (*) and divide (/). To this C (and most other computer languages) add the modulus operator (%), which yields the value of the remainder in a division operation involving whole numbers.

Thus we have:

Operator	Operation	Sample	Result
+	add	4 + 3	7
-	subtract	4 - 3	1
*	multiply	4 * 3	12
/	divide	4 / 3	1
%	modulus	4 % 3	1

In the divide example, the answer to the problem (4/3) is obviously 1 and 1/3. Using integer arithmetic, the quotent will hold only the whole number 1. However, the modulus operator -- being only interested in remainder values -- retains the value 1.

Precedence: an example

In C, the *, /, and % operators have a higher precedence than (and are evaluated before) the + and - operators. For example:

Operator	Operation	Sample	Result
n/a	n/a	3 * 2 - 1	5
n/a	n/a	3 - 1 * 2	1

However, parenthesis, which have precedence over all other operators can be used to reorder the way the expression is evaluated. Note how the results are changed:

Operator	Operation	Sample	Result
n/a	n/a	(3 * (2 - 1))	3
n/a	n/a	(3 - 1) * 2	4

Unary Minus

A minus sign placed next to a number indicates that the number is negative.

Operator	Operation	Sample	Result
-	unary minus	3 + -2	1
		3 - -2	5

Assignment Operators

There are several assignment operators. By far the most common is the single equals sign (=) which assigns values from right to left. The left side of the operation is known as the *lvalue* and the right side as the *rvalue*.

Operator	Operation	Sample	Result
=	assigns value	a = 3	a equals 3

In keeping with C's enthusiasm for "computer shorthand", the concept of assignment operators also includes operations such as:

```
i += 3;
```

which is exactly the same as:

```
i = i + 3;
```

Such shorthand is for the convenience of the programmer only and is never required for your program. Here are the other C assignment operators:

Operator	Operation	Sample	Result
+=	plus-equals	i += 3;	i = i + 3;
-=	minus-equals	i -= 3;	i = i - 3;
*=	times-equals	i *= 3;	i = i * 3;
/=	divide-equals	i /= 3;	i = i / 3;
%=	modulus-equals	i %= 3;	i = i % 3;
>>=	r.shift-equals	i >>= 3;	i = i >> 3;
<<=	l.shift-equals	i <<= 3;	i = i << 3;
&=	bitwise and-equals	i &= 3;	i = i & 3;
^=	bitwise exclusive or equals	i ^= 3;	i = i ^ 3;
|=	bitwise inclusive or equals	i |= 3;	i = i | 3;

(See also descriptions of bitwise operators, below.)

Relational and Logical Operators

Relational operators compare two values. Values may be numerical or strings of characters. Results of the comparison test is either true (1) or false (0).

The relational and logical operators are:

```
>  >=  <  <=  ==  !=  ||  ! &&
```

It is perhaps easiest to understand the function of these symbols through example:

Operator	Operation	Sample	Result
>	Greater than	4 > 3	1 (true)
		2 > 4	0 (false)
<	Less than	4 < 3	0 (false)
		2 < 4	1 (true)
>=	Greater than or equal to	4 >= 2	1 (true)
		2 >= 4	0 (false)
<=	Less than or equal to	4 <= 2	0 (false)
		2 <= 4	1 (true)
==	Equals	4 == 4	1 (true)
		4 == 3	0 (false)
!=	Not equal	4 != 4	0 (false)
		4 != 3	1 (true)
&&	Logical AND	3>4 && 3>9	0 (false)
		3<4 && 3<9	1 (true)
||	Logical OR	3>4 || 3>9	0 (false)
		3>4 || 3<9	1 (true)
!	Unary negative	!3	0 (false)

The unary negative operator is used to change or convert a non-zero (true) value to 0, or a zero or false operand into 1. The classic example is:

```
if (!inword)
```

which will generate a 0 if the **inword** test evaluates to 0 (false). K&R say of the above statement that it "reads quite nicely (as) 'if not in word' ".

Increment and Decrement Operators

C provides a simple system for increasing or decreasing the value of a variable. A double-plus sign (++) increases or *increments* the value of a variable by 1; a double-minus sign (--) decreases or *decrements* the value of a variable by 1. Here are some examples:

Operator	Operation	Sample	Result
++n	increments by one	++a	variable a increased by 1

--n	decrements by one	--b	variable b decreased by 1

The increment and decrement operators can be placed either before or after the variable or *operand* which they are operating upon. In some situations the placement is significant. The rule is:

> If the increment or decrement operator appears before the variable, the variable is incremented before its value is used. If the increment or decrement operators are placed after the variable, the value is used before the variable's value is changed. Thus, to lean on two more examples from K&R:
>
> ```
> n = 5;
> x = n++;
> ```
>
> will set **x** to 5, but
>
> ```
> n = 5;
> x = ++n;
> ```
>
> sets **x** to 6.
>
> It is important to note that in both cases **n** is incremented to the value 6. The difference is that in Form [1] the assignment operation is performed prior to incrementing the value of **n**.

Bitwise Operators

Several operators are available to modify the bits of character variables. In the following operations the initial value of **x** is 3, **y** is 2 and **z** is 1 as in:

```
x = 3;
y = 2;
z = 1;
```

Here are the bitwise operators and their effects:

Operator	Operation	Sample	Result
&	bitwise AND	y&z	0
|	bitwise inclusive OR	x|y	3
^	bitwise exclusive OR	x^y	2
<<	left shift bit	z<<3	8
>>	right shift bit	x>>1	1
~	one's complement (unary)	~z	-2

Regarding ~, the one's complement operator, results may vary depending upon the type of object since there may be a differing number of bits involved. The following program illustrates this:

```
main()
/* illustrates one's complement for various objects */
{
char a = ~1;
int b = ~1;
long c = ~1;
unsigned d = ~1;
     printf("%u %u %u %u\n",a,b,c,d);
/* answers are: 254 65534 65534 65535 */
}
```

Assorted Operators and Expressions

There are a number of other single or double-character operators including:

```
*     &     ,     ?     :     ->     .
```

which are described below:

*** the Indirection Operator**

The asterisk (*) does double duty as an indirection operator in dealing with pointers and as the more familiar multiplication operator. The term *indirection* refers to the fact that the program is calling for the contents of memory at the address assigned to the pointer variable.

For example:

```
y = *p;
```

assigns to **y** the contents of whatever the variable **p** points to in memory.

In the program:

```
main()
{
char a, *b;
b = "my text";
a = *b;
printf("%c", a);
}
```

the variable **a** will indirectly be assigned the character **m** (decimal 109) through the indirection operator associated with variable **b.**

& the Address-of Operator
The address operator (&) gives the address of an object in memory. For example:

```
p = &i;
```

assigns the memory address pointed to by the variable **i** to **p.** To illustrate:

```
main()

/* illustrates useless use of address-of operator */
{
int i;
i = 40;

      printf("i's value is %d.\n", i);
      printf("i's address on your machine is %ld.",
      &i);
}
```

, the Comma Operator
The comma operator (,) is often used in **for** statements. It allows multiple initializations of variables and multiple processing of indices. A pair of expressions separated by a comma is evaluated left to right. In practice this enables constructions such as:

```
for (sum = 0, i = 1; i <= n; ++i);
```

which sets the variable **sum** to 0 and **i** to 1.

Another example:

```
for (sum = 0, i = 1; i <= n; --i, ++sum);
```

which will decrement the variable **i** and increment the variable **sum** each pass through the **for** loop.

? : the Conditional Expression Operators

The conditional expression operators represent an abbreviated form of the if-else construction and are used in the following manner:

```
expression_1 ? expression_2 : expression_3
```

If **expression_1** is true (non-zero), program control passes to **expression_2.** If **expression_1** is false (zero), control passes to **expression_3.** The prototype could as easily be written using the **else-if** commands:

```
if(expression_1)
      expression_2;
else
      expression_3;
```

In both cases either **expression_2** or **expression_3** is evaluated, but not both.

The conditional operators should be used only in expressions which are complete on a single logical line.

. the Structure Member Operator

Members of structures (and **unions**) are accessed directly using the structure member operator (.) to connect the specific structure with its member. Using the structure member, it is possible to find and/or alter the value of any member of a structure.

The classic K&R **struct** example is similar to:

```
struct date {
      int day;
      int month;
      char mon_name[4];
};

struct date d;
```

The following code sets the day of the month within the structure **date:**

```
d.day = 28;
printf("Today's is the %dth", d.day);
```

The structure member operator works the same with **unions**, which are really special cases of **structs**.

-> the Structure Pointer Operator

Members of structures (and unions) are accessed indirectly using the structure pointer operator (->). Since C isn't APL (a language which uses special screen symbols), the structure pointer operator is constructed of a hyphen and a greater-than sign.

Using the structure pointer operator is possible to find and/or alter the value of members of structure (and **unions**) indirectly.

We'll use the same example program fragment:

```
struct date {
        int day;
        int month;
        char mon_name[12];
};
```

Notice that the last member in the structure is a 12-element character array which will contain the name of a month. We can now declare and initialize an array of structures of type **struct date** as follows:

```
struct date year[3] ={
        {5,1,"JANUARY"},
        {4,2,"FEBRUARY"},
        {10,3,"MARCH"}
        };
```

We can also declare a pointer to a structure of type **struct date** as follows:

```
struct date *pyear;
```

Using the structure pointer operator we can print out the name of the second month with:

```
pyear = &year[1];
printf("Name of the %d month is: %s", pyear->month +
1, pyear->mon_name);
```

The **union** pointer operator works the same way as the structure operator.

Operator Precedence

The following chart describes the order of evaluation of commonly used C operators. The listing is in decreasing order of precedence, meaning that the higher in the chart the operator appears, the earlier it will be evaluated.

As mentioned above, in cases where several different operators are listed on a single line, the order of evaluation is determined by the associativity -- either **left-to-right** or **right-to-left.** If the associativity is left-to-right, the evaluation will proceed from the left. If the associativity is right-to-left, the evaluation will proceed from the right. For example, the expression:

```
3 * 4 + 2
```

has two operators of equal precedence. But because the associativity is left-oriented, **3 * 4** will be evaluated before **4 + 2** and the correct answer will be 14 rather than 18.

Parentheses operators are given the highest priority. Thus the above expression could be made to evaluate to 18 by changing it to:

```
3 * (4 + 2)
```

Using parenthesis you can force the evaluation of any low-precedence expressions before higher-precedence expressions.

Most operator operations are associated from left to right, but key operations such as assignments are evaluated right to left as in: **a = 3.**

Operator	Associativity
() [] -> .	left to right
! ~ ++ -- - (type) * & sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= >>= <<= &= ^= |=	right to left
,	left to right

Precedence and Order of Evaluation of Operators***

*** Adapted from K&R, p. 49.

SECTION 4. COMMANDS, KEYWORDS and FUNCTIONS

Summary

In Part A, all RUN/C terms are listed under somewhat arbitrary groupings. The term's syntax is also usually given, although the syntax may not be complete.*

Part B contains a comprehensive description in alphabetical order of all RUN/C commands, functions and keywords.

---

* Terms marked with asterisks are only available in RUN/C Professional.

Part A. RUN/C terms by categories

Summary The categories of RUN/C terms and their meanings are:

Category	Meaning
Commands	Instructions which can be given to the interpreter.
Keywords	C terms and keywords.
Data types	The various sizes and formats used to stored numerical information.
Character- and string-related functions	Functions primarily used to manage characters and strings.
I/O functions	Functions primarily used to address the printer, screen or other I/O ports and addresses.
Memory-related functions	Functions used to manage memory available to the interpreted program.
File functions	Functions primarily used to manage files and file-related data.
Math functions	Mathematical functions.
Miscellaneous	Lists other functions and special terms.

Summary RUN/C commands are interpreter statements which can only be given at the Command level or through a debugging function or command.

@
Provides access to RUN/C's immediate mode.

AUTOSAVE [n]
Automatically saves the program in memory to disk every *nn* commands.

BREAK*
Simulates a <Ctrl-c> at a specific point in the program, thus providing a breakpoint. Used in conjunction with the **runc()** function.

C (Continue)
Continue execution of a program after it has been interrupted.

CLIST
Displays all RUN/C commands.

CLS
Clears the screen and places the cursor and the **Ok** prompt at the top of the screen.

D (Dump)
Displays the current values of active variables after the program has been interrupted.

DIR [filespec]
Displays the names of specified files on the default or specified drive/directory.

E
Call the built-in program editor. **E** is an alias for **EDIT**.

E (End)
Terminates execution of a program after it has been interrupted.

EDIT
Calls the program editor and begins editing at the previous editing point, if any. If the program has not previously been edited during the session, the

cursor will be placed at the beginning of the first line of the program.

EDIT [nnnn]
Calls the program editor and begins editing at the specified line.

EDIT END
Calls the program editor and begins editing at the last program line.

EDIT ERROR
Calls the program editor and begins editing on the line where the last error was recognized. If the error was not tied to an **autoline** number, the cursor will be placed on the last program line.

EDITOR
Sends the program currently in memory to an external editor selected by the user.

FILES [filespec]
Displays the names of [specified] files on a drive/directory.

FLIST
Displays names of all built-in RUN/C functions.

FRE
Returns the number of bytes of user memory available to a program.

H (Help)
Provides help regarding mini-menu commands which are only available after the program has been interrupted.

HELP
Displays system-dependent information.

I (Interactive)*
Enters the Interactive (debug) mode after program execution has been interrupted.

KILL [filespec]
Deletes the specified file from the default or specified drive/directory.

L (Ldump)
Print the current values of active variables after the program has been interrupted.

LIB*
Displays the names of Loadable Library commands and provides a brief description of each.

LIBDIR (or LIBD)*
Displays the names of the Loadable Libraries on the default or specified drive/directory.

LIBLOAD (or LIBL)*
Loads an available Loadable Library from the default or specified drive/directory.

LIBSHOW (or LIBS)*
Displays information about loaded libraries and the functions they contain. Wildcards are supported.

LIBUNLOAD (or LIBU)*
Releases memory which had been occupied and assigned to Loaded Libraries.

LIST [line a-line b]
Displays [portions of] the program in memory.

LLIST [line a-line b]
Prints [portions of] the program in memory.

LOAD [filespec]
Reads a program into memory.

NEW
Deletes the program currently in memory and clears the screen.

NLIST [line a-line b]
Displays [portions of] the program in memory without autoline numbers.

NLLIST line a-line b
Prints specified [portions of] the program in memory without *autoline* numbers.

PREC
Displays the C operator precedence table.

PROFF
Turns the profiler (**PRON**) off.

PRON
Activates the built-in program profiler which counts the number of times each line in the previously executed program was interpreted. The profiler report appears when the program is subsequently **LIST**ed.

RUN
Executes the program in memory.

SAVE [filespec]
Saves a program in memory to the specified name. Pathnames are supported.

SET
List available **SET** commands.

SET EDITOR [filespec]
Establishes an internal 'alias' name for the **EDITOR** command.

SET STEP* [on/off]
Sets program single stepping on or off. Stepping is accomplished by pressing any key.

SET TRUST [on/off]
Turns on or off the default trapping of pointer values outside the normal user data area. In addition, the automatic trapping of NULL-valued pointers is disabled or re-enabled.

SHELL [command]
Permits the temporary transfer of control from RUN/C to the operating system or directly to another executable program.

SYSTEM
Unconditionally returns control to the operating system.

TAB [nn]
Sets the spacing between tabular stops. The default is 5, the minimum 2 and the maximum 12.

TRACE
Traces all variable values during the execution of a program when the program is RUN.

TROFF
Turns the program line trace (TRON) off.

TRON
Traces the execution of program statements by displaying lines of source code interspersed with any program output.

TYPE [filename]
Displays the contents of a disk file without affecting the program currently in memory, if any.

UNLINK [filespec]
Deletes a specified file from the default or specified drive/directory.

VAL [variable name]*
Reports the value of any specified variable from either the **I (Interactive)** mode or through the **runc()** function.

VER
Displays the RUN/C version number and the date of creation.

Summary C keywords comprise the syntax of the C programming language. This includes statements such as **while** and **for**, as well as data types such as **char** and operators such as ==. (C operators are described in Section 3.)

Statements

auto
Declares a variable within a function to be an automatic.

break
Causes an immediate exit from the enclosing loop or switch body.

case x:
Controls branching within a **switch** statement.

continue
Causes program control to continue with the next iteration of a loop.

default:
Specifies a particular course of action when none of the cases within a **switch** statement match.

#define xxxx(zzzz) yyyy
Automatically replaces string **xxxx** with string **yyyy** each time the former appears. Arguments (represented as **(zzzz)**) are supported.

do (statement(s)) ... while (expression(s))
Executes a statement or statements as long as the tested expression remains true.

else(statement(s))
Specifies an alternative course of action if the condition part of an **if** statement is false.

else if(expression(s)) ... (statement(s))
Allows for multiple choice decisions within the **if . . . else** decision system.

extern
Allows the declaration of a variable as being defined in another source file.

for(initial value(s); test; adjust value(s))
{statement(s)}
Defines conditions under which a statement or block of statements are executed repeatedly.

goto label:
Goes to a labeled point elsewhere within the function.

if(expression(s)) statement(s)
Executes a program statement (or series of statements) based upon an expression being true (non-zero).

#include [filespec] or **#include <filespec>**
Allows data from an external file to be included in the currently executing program.

register
Allows the declaration of a variable as a register variable.

return(expression)
Terminates the execution of the current function, returning control to the calling function, and -- if an expression is present -- passing its value back to the calling function.

static
Creates a variable of the specified data type which will hold its current value even after program control has passed to another function.

switch(x)
Switches control to one of several parallel **case** statements if there is an exact match between the **switch** constant and a **case** constant or to a **default** statement, if one exists.

typedef
Provides a means for creating new names for existing data types, including aggregates.

void
Declares a function which does not return a value.

while(expression(s)) statement(s)
Executes a statement or statements if the test expression is true, continuing execution for so long as that expression remains true.

Data types

char
Creates a variable limited to a single character value (a positive decimal value 0-255).

double
Creates a variable with range of 15-16 decimal places of precision (+/-10e-307 to +/-10e308).

float
Creates a variable with up to seven decimal places of precision (+/- 10e37 to +/- 10e38).

int
Creates a signed variable 16 bits in length (-32768 to 32767).

long
Creates an integer variable up to 32 bits in length (-2,147,483,648 to 2,147,483,647).

short
Creates a signed variable 16 bits in length (-32768 to 32767).

struct
Creates a definition for a group of variables which can be referenced under a single group name.

union
Provides a single data area which can hold multiple members. As such, a **union** allows manipulation of different kinds of data in a single area of storage.

unsigned
Creates a positive variable 16 bits in length (0 to 65,535).

Summary RUN/C has more than 100 functions. Categories include character- and string-related functions, file functions, memory-related functions, I/O functions and math functions.

Character- and string-related functions

atof(x)
Converts an ASCII string to a double-precision floating point number.

atoi(x)
Converts an ASCII string into an integer number.

atol(x)
Converts an ASCII string into a **long** number.

x=getch()
Obtains a character from the console.

x=getchar()
Obtains a character from the keyboard.

x=gets(s)
Obtains a string of characters from the keyboard.

x=inkey()
Determines whether a key has been hit by continuously polling the keyboard; when a key is hit the function returns its ASCII code (or other code for non-ASCII characters), without printing the character on the screen. If no key is hit, a non-key-hit code (-2) is returned.

isalnum(x)
Tests character, returning non-zero if it is an alphanumeric character.

isalpha(x)
Tests character, returning non-zero if it is alphabetic character.

isascii(x)
Tests character, returning non-zero if it is an ASCII character.

iscntrl(x)
Tests character, returning non-zero if it is a control character.

iscsym(x)
Tests character, returning non-zero if it is a valid character for a C identifier.

iscsymf(x)
Tests character, returning non-zero if it is a valid first character for a C identifier.

isdigit(x)
Tests character, returning non-zero if it is a decimal digit (0-9).

isgraph(x)
Tests character, returning non-zero if it is a graphic character (excluding blank).

islower(x)
Tests character, returning non-zero if it is lowercase.

isodigit(x)
Tests character, returning non-zero if it is an octal digit (0-7).

isprint(x)
Tests character, returning non-zero if it is printable (including blank).

ispunct(x)
Tests character, returning non-zero if it is a punctuation character.

isspace(x)
Tests character, returning non-zero if it is white space.

isupper(x)
Tests character, returning non-zero if it is uppercase.

isxdigit(x)
Tests character, returning non-zero if it is a hexadecimal digit (0-A).

printf("formatted string",argument(s))
Prints format string with arguments inserted to the standard output device (usually the console).

putch(x)
Sends a character directly to the console.

putchar(x)
Sends a character to standard output device (usually the screen).

puts("string")
Outputs a string to the standard output device (usually the screen).

scanf("formatted string",argument(s)
Captures and formats data input by the standard input device (usually the console).

strcat(x,y)
Concatenates two strings.

strcmp(x,y)
Compares contents of two strings.

strcpy(x,y)
Copies one string to another.

strlen(x)
Measures length of a string.

tolower(x)
Converts an uppercase character to lowercase.

toupper(x)
Converts a lowercase character to uppercase.

ungetch(x)
Sends a character just captured back to the console keyboard buffer.

I/O functions

x = aget()
Obtains a character from the communication port.

aput(x)
Sends a character to communications port.

x = inp(port)
Obtain a value from a port.

intdos(inreg,outreg)
Generates a DOS interrupt.

intdosx(inreg,outreg,segment)
Generates a DOS interrupt which requires the segment registers to be set.

int86(inreg,outreg)
Generates a CPU (machine specific) interrupt.

int86x(inreg,outreg,segment)
Generates a CPU (machine specific) interrupt that requires the segment registers to be set.

locate(y,x,mode)
Moves the cursor or determines its location.

lprintf("formatted string", argument(s))
Sends formatted output to the printer.

lputc(x)
Sends a character to the printer.

lputs("string")
Sends a string of characters to the printer.

outp(port,value)
Sends a value to a port.

peek
Reads a byte in a location in memory.

poke
Writes a byte to a location in memory.

setcom(port,mode)
Initializes an asynchronous communication port for various communication parameters.

sprintf(ds,"formatted string",argument(s))
Generates a formatted string in memory.

sscanf(ds,"formatted string",argument(s))
Performs formatted input conversions from a string in memory.

Memory-related functions

calloc(x,y)
Dynamically allocates and clears a block of memory.

free(x)
Releases memory which has previously been dynamically allocated.

malloc(x)
Dynamically allocates a block of memory.

movmem(source,destin,num)
Copies a block of memory to a target area.

sizeof(x)
Calculates the size in bytes of a data type or data object.

File I/O functions

close(x)
Closes an unbuffered file.

clrerr(x)
Clears the error flag for the specified file. This function is available for files opened under *Level 2* I/O.

creat(filespec, mode)
Creates a new disk file under *Level 1* I/O.

exit()
Flushes any open file buffers and closes all buffered files before terminating program execution.

fclose(fp)
Concludes interaction to a buffered file.

feof(fp)
Checks for end-of-file for a file accessed with *Level 2* I/O.

ferror(fp)
Checks the error status of a file opened under *Level 2* I

fflush(fp)
Writes any data being held in a file's buffer to the disk.

x=fgetc(fp)
Reads a single character from the referenced buffered file.

x=fgets(x,n,fp)
Reads n-1 characters from the referenced buffered file.

FILE
Calls a structure built into RUN/C which contains all information necessary for the management of disk files by interpreted programs.

fopen(filespec,mode)
Opens a buffered file.

fprintf(fp,"format string",argument(s))
Sends formatted output to a file.

fputc(x,fp)
Write a character to the specified buffered file.

fputs(x,fp)
Sends a character string to a buffered file.

fread(buffer,size,num_blocks,fp)
Reads blocks of data from a buffered file.

freopen(filespec,mode,fp)
Reopens a buffered file.

fscanf(fp,"formatted string",argument(s))
Performs formatted input from a file.

fseek(fp,n,mode)
Seeks to a new position in a buffered file.

x=ftell(fp)
Returns the current position of the file pointer.

fwrite(buffer,size,num_blocks,fp)
Writes blocks of data to a buffered file.

x=getc(fp)
Obtains a character from a file.

lseek(file,n,mode)
Seeks to a specified position in a file which has been opened for unbuffered file I/O.

open(filespec,mode)
Opens a disk file for unbuffered read or write operations.

putc(x,fp)
Send a character to the specified file.

read(file,buffer,x)
Reads data from an unbuffered file.

rename(x,y)
Renames a disk file.

rewind(fp)
Resets the file position pointer to the first byte of the buffered file.

ungetc(x,fp)
Pushes a character back on the file from which it was obtained.

unlink(filespec)
Deletes a file from the default or specified drive/directory.

write(file,buffer,x)
Writes data to an unbuffered file.

Math functions*

acos(x)
Calculates the arc cosine of a number.

asin(x)
Calculates the arc sine of a number.

atan(x)
Calculates the arc tangent of a number.

atan2(y/x)
Calculates the arc tangent of the quotent of two numbers.

ceil(x)
Calculates the closest whole number just above a double precision number.

cos(x)
Calculates the cosine of a number.

cosh(x)
Calculates the hyperbolic cosine of a number.

exp(x)
Calculates the exponential value of a number.

fabs(x)
Calculates the absolute value of a number as a floating point value.

floor(x)
Calculates the closest whole number value at or just below a double precision number.

log(x)
Calculates the logarithm of a number.

log10(x)
Calculates the base 10 logarithm of a number.

* In RUN/C Professional all mathematical functions are available in the MATHLIB Loadable Library system supplied on the distribution disk.

pow(x,y)
Calculates the power of a number.

x = rand()
Generates a random number.

sin(x)
Calculates the sine of a number.

sinh(x)
Calculates the hyperbolic sine of a number.

sqrt(x)
Calculates the square root of a number.

srand(x)
Resets the random number seed used by **rand**.

tan(x)
Calculates the tangent of a number.

tanh(x)
Calculates the hyperbolic tangent of a number.

Miscellaneous

cls()
Clears the screen from the program being interpreted.

dosdate(x,mode)
Copies the system date to a string in the specified form.

dostime(x,mode)
Copies the system time to a string in the specified form.

main(argc, argv)
The primary function in a C program; a required component of any executable RUN/C program. Arguments are optional.

run("COMMAND")*
Permits execution of RUN/C commands -- including breakpoints -- from within interpreted programs.

stderr
Provides an error reporting facility for the running program independent of whatever is the currently assigned standard output device (**stdout**).

stdin
Provides an input facility for the running program.

stdout
Provides an output facility for the running program.

Part B. RUN/C terms in alphabetical order

Summary In this subsection commands, keywords and functions are listed in alphabetical order with commands beginning with punctuation marks (such as **#define** or **@**) in their ASCII order). Here are the description categories and their meanings:

PURPOSE	Tells what the term does.
FORM	Provides one or several standard format(s) for the term.
USAGE	Describes in detail how the term may be used. In addition, the sample may be explained.
NOTES	Provides user notes, suggestions and warnings and other tidbits of information such as comparisons between related commands, keywords or functions.
LIMITS	Describes any limitations on the item in question compared to its application in a standard C compiler environment.
RETURNS	Tells what values, if any, are returned by a function and how to interpret them.
EXAMPLE	Provides an example program illustrating use of the term. Often, the same example is supplied on the distribution disk under the name of the command, keyword or function.
SEE ALSO	Refers the reader to related terms described elsewhere in the manual.

The categories above are only used when appropriate. For example, there is no RETURNS category for Commands.

PURPOSE Provides for the automatic replacement of an identifier with another string throughout the program.

FORM

```
[1] #define   ONE          1
    #define   PI           3.141592654
    #define   GREETING     "Hello, world!"

[2] #define   SQUARE(x)    ((x) * (x))
    #define   PR(x)        printf("%s", x)
    #define   ADD(x,y)     (x + y)

[3] #define   SUBTRACT(x,y)     ((x) - (y))
    #define   MINUS(x,y)        SUBTRACT(y,x)

[4] #define  then
```

USAGE **#define** is a preprocessor directive which make substitutions in your program prior to interpretation of the code.

#define can be expressed in the form:

directive identifier[(parameter, .., ..)] [definition]

where the **directive** is **#define**, the **identifier** is a character or alphanumeric string, the optional **parameter list** contains one or several variables within parentheses (separated by commas if there is more than one parameter) and the optional **definition** replaces the identifier with the definition string wherever it occurs.

In its most basic form **#define** permits the creation of more "readable" and potentially more "portable" C programs by allowing the substitution of one string for another wherever it appears in the program. **#define** also makes it possible to assign constant values or terms to identifiers which are easier for the programmer to input and for the reader of the code to recognize.

It its more complex form (illustrated in the second and third groups of examples above), **#define** can be used for creating sophisticated *macro* substitution systems which can potentially simplify program design and speed up program execution by doing work that would normally be assigned to independent functions.

In the above samples:

[1] #define ONE 1

Each time the word ONE is encountered it will be treated as if the numeral 1 were present.

#define PI 3.141592654

when the term PI is encountered, the program will behave as if you had placed 3.141592654 in the program.

#define GREETING "Hello, world!"

If the term GREETING is used, the string "Hello, world!" will be substituted for it. Carrying our substitution system a step further it is possible to make C appear like quite another language:

```
#define BEGIN {
#define END   }
#define PROGRAM  main()
#define PRINT  printf
```

Now the program:

```
PROGRAM
BEGIN
      PRINT(GREETING);
END
```

will produce:

```
Hello, world!
```

[2] #define SQUARE(x) ((x) * (x))

will produce the square of a number. For example:

```
SQUARE(5)
```

will expand into the token string:

```
((5) * (5))
```

which, of course, is 25. Note the extra parentheses around each x. If these were not used it would possible to get the wrong result as shown in the following example:

```
#define SQUARE(x) (x*x)
```

In the macro call:

```
SQUARE(1+2)
```

the preprocessor will expand the line into:

```
(1+2 * 1+2)
```

which equals 5 instead of 9 (2*1=2+1+2=5) since C's rules of precedence require that multiplication be performed before addition. But by adding parentheses as in:

```
SQUARE(x)   ((x) * (x))
```

the preprocessor will expand the line to:

```
((1+2) * (1+2))
(1+2=3 ; 1+2=3; 3*3=9)
```

giving us 9, the correct answer.

In complex macros multiple levels of parenthesis and other operations are resolved in the same order as that used by the C operations table (see Section 3).

```
#define   PR(x)        printf("%s\n", "x")
```

provides for substitution of the argument given into a **printf** statement. For example:

```
PR(test)
```

will print the string **test** followed by a newline. A macro along similar lines would be:

```
#define puts(x) printf("%s\n",x)
```

Indeed, this is how many "apparent" functions are often created by C function library writers.*

```
#define ADD(x,y)   ((x) + (y))
```

creates a macro which adds together two parameters which are presented, as in:

```
ADD(2,4)
 result is: ( (2) + (4) )
```

or

```
ADD(a,b)
 result is: ( (a) + (b) )
```

[3]
```
#define MINUS(x,y)      SUBTRACT(y,x)
#define SUBTRACT(x,y)  ((x) - (y))
```

in this pair of macros, the identifier **MINUS(x,y)** is expanded in terms of the string **SUBTRACT(y,x)**, which itself is expanded into **((x) - (y))**. Thus the macro invocation:

```
MINUS( MINUS(3,4), 5)
```

is expanded in several steps as a preprocess to the program interpretation (or compilation, if you are using a compiler):

```
SUBTRACT (5, MINUS(3,4))
```

the outermost **MINUS** is expanded according to the **SUBTRACT** token-string resulting in:

```
((5) - MINUS(3,4))
```

the **MINUS** identifier is expanded in terms of its token string resulting in:

```
((5) - SUBTRACT(4,3))
```

the inner **SUBTRACT** identifier is expanded to its token string resulting in:

* See your compiler manual for information on "functions" which are implemented as macros and about possible side effects from same.

((5) - ((4) - (3))
At run-time (or compile time, with a smart compiler) this expression evaluates to:

4
the final result.*

[4] #define then
substitutes nothing (a *NULL body)* for the defined term. This makes it possible for Pascal converts to write code like:

```
if(a == 255)
 then printf("Abort mission!!");
```

which the preprocessor will turn into conventional C code without the pseudo-keyword **then.**

In keeping with K&R C if an opening parenthesis with no intervening white space appears immediately following the first portion of the **#define** statement (the *identifier),* the C preprocessor assumes that a **#define** with arguments is being presented.

NOTES

* The maximum number of defined terms is set at 25 as of this writing (see the on-line **HELP** screen for the current setting on the maximum number of **#defines**). However, this number can be changed when RUN/C is invoked from DOS. For example:

```
RC -d50
```

will permit up to 50 **#defines** in a program including any in **#include**'d files. This also effectively sets a "nesting" limit on **#defines** at 49.

* For the sake of program readability **#define** statements should appear at or near the beginning of the program.

---

* In terms of RUN/C it is important to note that this simplification is not done by the preprocessor. Rather, the full computation is done at interpretation time each time the line is encountered. Thus code using the example macro call instead of the constant 4 will run a little slower, and if the code is within a loop the speed difference could be noticeable.

* In case a term is **#define**'d twice, the last definition encountered will be used.

* RUN/C currently allocates approximately 200 bytes of memory for each **#define** when the program is loaded. Thus 25 **#defines** preallocate about 4,600 bytes of RAM even if no **#defines** are used in the program.

* If a term is redefined in the program in memory the last definition will be active. For example:

```
#define FIVE      5
#define FIVE      500
```

will result in **FIVE** being evaluated as 500. However, if a term is redefined in an **#include**'d file, the definition in the **#include** file will prevail. Any macro **#defines** in the program in memory will affect the **#include**'d material and vice-versa. In general, redefinition of macros is not a recommended programming practice.

* More than 24 macro occurrences on a single line (including macros produced by expanding macros) will cause an error. This is meant to keep RUN/C from getting into a recursive mode swapping mutually-defined macros for the rest of its natural life. To illustrate, here are two **#define** statements:

```
#define A B
#define B A
```

Note that **A** is defined to be **B** and **B** to be **A**. This is known as mutual recursion. When a statement such as:

```
printf("%d", A);
```

is encountered, the interpreter will replace **A** with **B** and then **B** with **A** 24 times before generating an error message to the effect that mutual recursion of macro definitions is suspected. Without this safeguard the only way out of such a mutually recursive loop would be to press <Ctrl-c>).

* The value of macro **#defines** is enhanced by the

fact that the macro can be used to exchange data of any data type. Thus:

```
#define mult(a,b)       ((a)*(b))
```

will multiply two numbers together regardless of whether the numbers are integers, doubles, floats or entirely new data types created with a structure. Of course normal care must be exercised not to mix and match data types in an illegal fashion.

* Program lines containing macro calls will be shown in their expanded form when displayed during program execution by means of the **TRON** command.

* In RUN/C the following constants are predefined:

Term	Definition
EOF	-1
NULL	0L (zero long)

and cannot be **#define**'d or redefined.

* In C, it is an often used (but non-binding) convention that words in capital letters are *macros.* These are often considered *manifest constants* (such as **EOF** or **NULL**), as compared to variables.

* For some creative employment of **#defines** -- which are supported by RUN/C -- see A. Feuer's C Puzzle Book. For a more detailed discussion on the use and abuse of the **#define** preprocessor statement see Harbison & Steele's C--A Reference Manual.

* In keeping with K&R C, **#define** statements must begin in the first column of the line on which they appear.

LIMITS

* In K&R C a **#define** symbol is available only to those portions of the program subsequent to definition. RUN/C is temporarily idiosyncratic in that it makes **#define** symbols globally available -- as if all **#defines** were placed at the beginning of the file. To insure that interpreted and compiled programs behave the same way, **#defines** should be placed at or near the beginning of your program.

* **#undef** and other preprocessor directives are not presently supported in RUN/C.

RETURNS Nothing.

EXAMPLE

```
#define     LOW 1896
#define     HIGH LOW + 108
#define     STEP 1
#define     PRINT(x) printf("%d is a leap year\n", x)

main()
{ /* define.c--illustrates use of #define */

int year;
for(year = LOW; year <= HIGH; year += STEP)
        if(year % 4 == 0 && year % 100 != 0 || year % 400 == 0){
                PRINT(year);
        }
}
```

SEE ALSO #include, TRON

PURPOSE Allows data from an external file to be included in the currently executing program.

FORM

```
#include "test.h"
```

or

```
#include <test.h>
```

USAGE **#include** calls in the named C source code file -- often called a *header file* using the extension ".h" -- and merges it with the program in memory when the program is run.

NOTES

* **#include** statements may be nested. Thus a file which is included may itself contain an **#include** statement that includes yet another file. RUN/C imposes no limit on the depth of **#include** nesting; most C compilers, however, do impose an arbitrary limit.

* The **#include** statement should be placed on the left margin of the line at or near the beginning of the program. **#include** and **#define** statements should appear before the first function in the program; never within a function.

* In C, header files traditionally have the extension ".h". Some of the header file names frequently used by C compilers are:

Header file	Contents
stdio.h	standard i/o information
ctype.h	**is_** function data
math.h	math function data

Such header files typically contain complex definitions. For example, stdio.h usually contains the definition of the built-in preprocessor symbol **FILE**.

Because of certain exigencies implicit in interpretation (vs. compilation) some of the items typically found in header files are built into RUN/C. These include the definitions of **NULL**,

FILE and **EOF** and declarations of the built-in functions that return values other than **char** or **int** (e.g., **malloc**, which returns a character pointer).

These definitions and declarations cannot currently be changed within the RUN/C context. Consequently, any attempt to include a file named STDIO.H will fail and the message:

```
Warning: ignoring attempt to include stdio.h
```

will be generated -- although program execution will continue. (Of course, stdio.h could be renamed or called from another drive, but attempts to include compiler header files are likely to be frustrating.)

* When a file is **#include**'d, it is appended to the end of the file in memory. If an error occurs in that file, its nature and the line number will be displayed when the error is detected. The only problem here is that the line number noted will be greater than any line in your program. If, for example, a 100-line program has an error in the first line of the **#include**'d file, it will be flagged as an error on line 101.

* If an error is noted in an **#include**'d file, there is no way to directly change the contents of the file. It must be edited separately using either RUN/C's built-in editor or your own.

* If an attempt is made to include a file which does not exist on the default drive or the specified drive/directory, a message such as:

```
Can't open program.c
```

will appear and program execution will cease.

RETURNS Nothing.

EXAMPLE

```
#include "8087test.c"
/* #include.c   -- illustrates unusual use of
include statement */
```

PURPOSE Provides an immediate mode for the execution of built-in C functions and single line programs.

FORM

```
[1] @ printf("Hello world");

[2] @ printf("%f", sqrt(349898.98237));

[3] @ int b=75; printf("%f", (float)b);
```

USAGE @ provides a means for the execution and testing of built-in RUN/C functions. In RUN/C Professional the Immediate mode can be used in conjunction with the Interactive mode (available through the mini-menu) to execute and test functions in the interpreted program and in loaded libraries.

In the above samples:

[1] `@ printf("Hello world");`
executes the built-in **printf** function.

[2] `@ printf("%d", (int)sqrt(349898.98237));`
prints an integer version of the square root of the supplied number.

[3] `@ int b=75; printf("%f", (float)b);`
casts the integer **b** to floating point format and prints its new value.

The Immediate mode is ideal for testing the calling conventions of available functions or for doing some quick math. Programs in memory will be unaffected by use of the Immediate mode.

The Immediate mode is considered **line 0** by the interpreter and line-oriented syntax errors will noted that an error has occurred at **line 0.** It is not difficult to generate error messages through misuse of the Immediate mode. For example the command:

```
@ print 8
```

will elicit the error message:

```
*** Error # 21 at line 0
Variable or function not found
```

since **print** has not been declared. Similarly the statement:

```
@ printf("%d", 3)
3
*** Error # 191 at line 0
Illegal termination of expression
```

will print the correct answer, but will also remind you that the immediate mode is no less particular than RUN/C in terms of proper terminations of expression. A ';' is required.

NOTES

* Unlike many versions of BASIC, variables which were declared and available during **RUN** are not available after the program has completed its work and the **Ok** prompt has reappeared.

* In RUN/C Professional the value of program variables can be changed after the program has been interrupted with a <Ctrl-c> or a breakpoint using the Immediate mode in conjunction with the mini-menu interactive mode; this technique is described in detail is Section 6.

EXAMPLE See above.

SEE ALSO runc(), BREAK, I(Interactive), Section 6

PURPOSE Calculates the arc cosine of a number.

FORM

```
double result;
double number;

result = acos(number);
```

USAGE **acos** computes the arc cosine or inverse cosine of a number, returning a radian value in the range of 0 to *pi*. Both the called number and the returned value should be *double precision.*

NOTES In RUN/C Professional this and other transcendental math functions are available in the loadable library MATHLIB.

RETURNS arc cosine of the number; if the number is greater than 1, 0 is returned.

EXAMPLE

```
main()
{ /* acos.c -- obtains arc cosine of a number */

double number, result;
number = 0.5;
result = acos(number);
printf("%f", result);
/*   The correct answer is:  1.047198 */
}
```

PURPOSE Obtains a character from the communication port.

FORM

```
char c;

c = aget();
```

USAGE **aget** will wait for and return the next available character from the currently active asynchronous communication port.

In the above sample:

`char c;`
initializes a variable to hold the captured character.

`c = aget();`
gets the first available character from the communication port and assigns it to **c**.

NOTES

* The function **aget** is based on a similarly-named function from Lattice's C Food Smorgasbord. The object library file or source code from these utility functions is needed in order to compile a RUN/C program using **aget**.

* Make certain your modem is properly hooked up and initialized; otherwise a warm reboot will be required.

RETURNS -1, if error is detected.
Character, if successful.

EXAMPLE

```
main()
{    /* aget.c -- tests aget() asynch function
     */
int ret;
ret = aget();
printf("%d", ret);}
```

SEE ALSO aput, setcom

PURPOSE Sends a character to the communications port.

FORM

```
int ret;
char c;

ret = aput(c);
```

USAGE **aput** sends a single character to the currently active asynchronous communication port.

In the above sample:

`int ret;`
declares a variable to hold the returned value.

`char c;`
declares a variable to hold the value of the character being sent.

`ret = aput(c);`
sends the character to the communication device. The value returned from the function will be assigned to **ret**.

NOTES

* The function **aput** is based on a similarly named function found in Lattice's C Food Smorgasbord. The object library file or source code from these utility functions is needed in order to compile a RUN/C program using **aput**.

* Make certain your modem is properly hooked up and initialized; otherwise a warm reboot will be required.

RETURNS -1, if an error is detected.
Character sent, if successful.

EXAMPLE

```
main()
{

/*
     aput.c -- illustrates use of aput()
      ... make sure your equipment is hooked up
     and port   properly initialized!
*/

int ret,;
char c;

c = 32;

     ret = aput(c);
     printf("%d", ret);

/*
     Send a space to the communication port;
     correct answer is 32
*/
}
```

SEE ALSO aget, setcom

PURPOSE Calculates the arc sine of a number.

FORM

```
double result;
double number;

result = asin(number);
```

USAGE **asin** computes the arc sine or inverse sine of a number, returning a radian value in the range *-pi/2* to *pi/2*. Both the passed number and the returned value should be double precision.

NOTES In RUN/C Professional this and other transcendental math functions are available in the loadable library MATHLIB.

RETURNS Arc sine of the number; if the number is greater than 1, 0 is returned.

EXAMPLE

```
main()
{  /* asin.c -- computes arc sine of a number */

double number, result;
number = 0.5;
result = asin(number);
printf("%f", result);
     /*    The correct answer is:  0.523599 */
}
```

PURPOSE Calculates the arc tangent of a number.

FORM

```
double result;
double number;
result = atan(number);
```

USAGE **atan** computes the arc tangent (inverse tangent) of a number, returning a radian value in the range of *-pi/2* to *pi/2*. Any positive or negative number is valid. Both the passed number and the returned value should be double precision.

NOTES In RUN/C Professional this and other transcendental math functions are available in the loadable library MATHLIB.

RETURNS Arc tangent of the number.

EXAMPLE

```
main()
{    /* atan.c -- computes arc tangent of a
        number */

double number, result;
number = 1.0;
result = atan(number);
printf("%f", result);
/*   The correct answer is:  0.785398 */
}
```

PURPOSE Calculates the arc tangent of the quotent of two numbers.

FORM

```
double result;
double x;
double y;

result = atan2(y,x);
```

USAGE **atan2** computes the arc tangent or inverse tangent of the quotent of two numbers, returning a radian value in the range of *-pi* to *+pi*. Both the passed numbers and the returned value should be double precision.

NOTES In RUN/C Professional this and other transcendental math functions are available in the loadable library MATHLIB.

RETURNS arc tangent of the first number divided by the second. **atan2(y,x)** = **atan(y/x)** gives the resulting angle in the radian value range *pi/2* or *-pi/2*. **x** may equal 0.

EXAMPLE

```
main() /* atan2.c -- tests atan2() */
{

double result;
result = atan2(1.0,2.0);
printf("%f", result);
/*   The correct answer is:  0.463648 */
}
```

PURPOSE Converts an ASCII string to a double-precision floating point number.

FORM

```
double result;
char *n;

result = atof(n)
```

USAGE **atof** converts an ASCII character string with or without scientific notation to a double-precision floating point number.

In the above sample:

`double result;`
declares a double-precision number to hold the result.

`char *n;`
declares a pointer to the address of the ASCII string to be converted.

`result = atof(n);`
assigns the converted number to the double-precision variable **result.**

In converting the number, leading white space is ignored (spaces, tabs, newlines). A plus (+) or minus (-) sign indicating positive or negative value may precede the string. If neither is present, the number is assumed to be positive. A decimal point may be used to express decimal fractions. An e or E may be included to indicate exponentation. A plus (+) or minus (-) may also follow the exponent marker (e or E), to indicate whether the exponent is positive or negative. The conversion stops when the first non-recognized character or non-leading white space is encountered.

Thus, if the char array **n** equals **-987.653E+7MY NUMBER**, the - would indicate a negative number. The **E+7** indicates an exponent to seven additional orders of magnitude. The **M** and subsequent characters are not taken as part of the number.

NOTES Neither hexadecimal nor octal representations are allowed in the ASCII string.

RETURNS Floating point number in double precision format.

EXAMPLE

```
main()
{    /* atof.c -- Converts ASCII string to
     floating point number */
char *c;
c = "    -123.456e7";
     printf("%lf", atof(c));

/* correct result is -1234560000.000000 */
}
```

SEE ALSO atoi, atol

PURPOSE Converts an ASCII string into an integer number.

FORM

```
int result;
char *n;
result = atoi(n);
```

USAGE **atoi** converts an ASCII character string into an integer number.

In the above sample:

`int result;`
declares an integer to hold the result.

`char *n;`
declares a pointer to the address of the ASCII string to be converted.

`result = atoi(n);`
assigns the converted ASCII number to an integer variable.

In calculating the integer, leading white space is ignored (spaces, tabs, newlines). A plus (+) or minus (-) sign may precede the string to indicate a positive or negative number. If neither is present, positive is assumed. The string conversion ends at the first non-recognized character or non-leading white space.

NOTES Octal and hexadecimal representations (such as **\016**) are allowed.

RETURNS Integer conversion of the ASCII string which was input.

EXAMPLE

```
main()
{
/*
     atoi.c  --  converts  ASCII  string  to  an
     integer number
*/

char *c;
int number;
     c = "-1234TEST";
     number = atoi(c);
     printf("Integer number is: %d", number);
/* correct result is -1234 */
}
```

SEE ALSO atof, atol

PURPOSE Converts an ASCII string into a long integer.

FORM

```
long result;
char *n;

result = atol(n);
```

USAGE **atol** converts an ASCII character string in an integer number.

In the above sample:

`long result;`
declares a long integer to hold the result of the calculation.

`char *n;`
declares a pointer to the address of the ASCII string to be converted.

`result = atol(n);`
assigns the converted ASCII number to the integer variable.

In calculating the long integer, leading white space is ignored (spaces, tabs, newlines). A plus (+) or minus (-) sign preceding the string indicates a positive or negative number. If neither is present, the number will be considered positive. The string conversion ends at the first non-recognized character or non-leading white space.

NOTES Octal and hexadecimal representations such as:

```
atol(0x10);
```

are allowed.

RETURNS Long integer conversion of an ASCII string.

(Continued next page)

EXAMPLE

```
main()
{
/*
     atol.c -- converts ASCII string to a long
     integer
*/

{
char *c;
long result;
c = "    -123234.456e7";
result = atol(c);
printf("\nlong result is %ld", result);
/* correct result is -123234 */
}
```

SEE ALSO atof, atoi

PURPOSE Allows the declaration of an automatic variable.

FORM
```
auto double dd, ff;
```

USAGE **auto** allows the explicit -- and optional -- declaration of an automatic variable within a function. Some programmers prefer to use the **auto** declaration to distinguish an automatic variable from a **static**. In RUN/C the declaration is cosmetic, however, since any variable within a function that is not declared **static** is handled as an **auto** -- this is also the case with standard C compilers.

NOTES According to Harbison & Steele, **auto** is permitted only in declarations of variables at the heads of blocks. In RUN/C **auto** declarations are somewhat less restrictive, but they must be within functions.

EXAMPLE
```
main()

{
auto double e = 523423234.234;
    printf("auto double e = %g", e);
}
```

PURPOSE Automatically saves the program in memory to disk every *n* commands.

FORM AUTOSAVE *n*

USAGE **AUTOSAVE** regularly saves the program in memory to the current drive/directory under the special filename PROGRAM.C. When **AUTOSAVE** is active, the program in memory will be saved every *n* times the **Ok** prompt appears. While a program is being saved to disk the message:

```
Saving line n
```

will be displayed. (Generally, the disk drive light will also be on while the file is being saved.)

If a previous version of PROGRAM.C exists on the disk it will be renamed PROGRAM.BAK before the **AUTOSAVE**.

Calling **AUTOSAVE** without a numeric argument turns it off. The message:

```
Autosave OFF
```

will appear. See Example 2, below.

If AUTOSAVE is called with a numeric argument, as in:

```
AUTOSAVE 8
```

the message:

```
Autosave every 8 commands
```

will appear.

If **AUTOSAVE** is active when the **SYSTEM** command is invoked, a final **AUTOSAVE** to the filename PROGRAM.C will be made before RUN/C returns you to the DOS operating system level.

NOTES In RUN/C Professional **AUTOSAVE** can be called from the executing program using the **runc()** function.

EXAMPLES:	Command	Effect
[1]	**AUTOSAVE 10**	The program in memory will be saved at every 10th appearance of the **Ok** prompt.
[2]	**AUTOSAVE**	The second call to **AUTOSAVE**, without a numeric argument, turns off the **AUTOSAVE** feature.

SEE ALSO SAVE, HELP, SYSTEM

PURPOSE Causes an immediate exit from the enclosing program statement block.

FORM `break;`

USAGE **break** exits from an enclosing **do, for, while** or **switch** block. Program execution continues at the next executable statement following the block. If the **break** statement occurs in a nested block, only the innermost block is exited.

break is frequently used within the **switch** function, where it terminates or prevents execution of a series of case statements.

RETURNS Nothing.

EXAMPLE

```
main()      /* demonstrates break */

{

     int x;

     x = 0;

     for( ; ;) {
          if (x >= 10)
               break;
     else
               printf("%d\n", x++);
     }
     printf("That's all folks!\n");

/* counts to 10, then breaks out of loop */
}
```

SEE ALSO `switch`

PURPOSE Simulates a <Ctrl-c> at a specific point in program execution, thus providing a predictable breakpoint in the code.

FORM `runc("BREAK");`

USAGE **BREAK** is a special breakpoint command that can be used in conjunction with the **runc()** function available in RUN/C Professional. The **BREAK** command is described in Section 6 of the Professional's manual; parts of that discussion will not be repeated here.

In the above sample:

```
runc("BREAK");
```

is the equivalent of placing a breakpoint in the program which, in turn, is the equivilent of pressing <Ctrl-c>. Like other functions, execution of breakpoints can be triggered by various logical events such as:

```
if(i < 0)
      runc("BREAK");
```

When the **BREAK** command is encountered program execution will stop as if you had pressed <Ctrl-c>. The difference is, obviously, that you can precisely control the breakpoint using the **runc()** function in conjunction with **BREAK**. When the **BREAK** is encountered program control will go to the RUN/C Professional mini-menu:

```
Break in Line n
      Continue, Dump, Ldump, Interactive, Help or End (C,D,L,I,H,E)?
```

Any of the six mini-menu commands are available including **I** (Interactive) which will allow you to view and manipulate variables and **C** (Continue) which will allow you to continue with program execution.

NOTES

* There are no limits on the number of breakpoints that can be placed in your program.

* Although the mini-menu display reports the number of the line in which the breakpoint occurred, breakpointing is not line oriented, and you can have multiple uses of **runc("BREAK");** on a single program line, as with:

```
puts("Pause..."); runc("BREAK"); puts("Brief resumption"); runc("BREAK");
```

EXAMPLE See above.

SEE ALSO runc(), Continue, Dump, Ldump, Interactive, Help or End, Section 6

PURPOSE Continue execution of a program after it has been interrupted.

FORM

```
Break in line n .
     Continue, Dump, Ldump, Interactive, Help or
     End (C,D,L,I,H,E)?

     C
```

USAGE Continue, a mini-menu command, allows you to resume program execution after it has been interrupted either by a <Ctrl-c> or a breakpoint. Execution resumes at the point where the program's action was interrupted.

Upon interruption the following menu will appear:

```
Break in line n
  Continue, Dump, Ldump, Interactive, Help or
End (C,D,L,I,H,E)?
```

If **C** is pressed, the program will continue beginning with execution of the named line.

For more information on the use of <Ctrl-c> see Section 1. For more information on the use of breakpoints see Section 6.

EXAMPLE

```
main()      /* demonstrates Continue option */
{
int a;

for(a = 0; a < 40; ++a)
     printf("%d \n",a);
}
```

```
RUN

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
```

At this point the display of these interesting numbers is interrupted by pressing <Ctrl-c>. The following choices appear:

```
Break in line 7
Continue, Dump, Ldump, Interactive, Help or End
(C,D,L,I,H,E)?  c
```

If **c** is selected, program execution will resume and the rest of the series will be displayed.

```
18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 ...
```

SEE ALSO D(Dump), E(End), H(Help), I(Interactive), L(Ldump), Section 6

PURPOSE Dynamically allocates and clears a block of memory.

FORM

```
char *p;
unsigned num_elements;
unsigned element_size;

p = calloc(num_elements, element_size);
```

USAGE **calloc** allocates and clears (sets to zeros) a contiguous area of memory.

In the above sample:

`char *p;`
declares a pointer.

`unsigned num_elements;`
declares a positive number to hold the number of elements desired in the block of memory.

`unsigned element_size;`
declares a positive number to hold the size of each element in bytes.

`p = calloc(num_elements, element_size);`
All bytes within the allocated area (**num_elements** multiplied by **element_size**) are set to zero. The returned address of the base of the allocated area in memory is assigned to the **p** pointer.

Typically, the second argument is a **sizeof** expression of the form:

```
p = calloc(n, sizeof(object));
```

returning a pointer to the block of memory if successful; otherwise a NULL is returned. The most likely reason for a NULL being returned is that there was not enough memory available to fill the request.

How calloc works
calloc returns a character pointer to the start of a block of contiguous memory. The size of the memory area is determined by the product of the number of elements by the size of each element. Once selected, all

bytes within the allocated area are set to zero.

For example, if you wish to allocate space for 100 integer values:

```
p = calloc(100, sizeof(int));
```

would do the job.

Similarly, if you had a structure named **p_and_l** which occupied 40 bytes, and wanted to set aside space for the profit and loss data of the Fortune 500, the call to **calloc** might be:

```
p = calloc(500, sizeof(p_and_l);
```

Of course it would be equally valid to allocate memory by positive numbers, as in:

```
p = calloc(144, 24);
```

which would allocate and set to zero 3,456 bytes of contiguous memory.

NOTES

* Here are the major differences between **malloc** and **calloc:**

  1) **malloc** allocates a block of memory a given number of bytes long, but does not initialize the bytes within the area to any particular value. Thus the allocated memory will initially contain "random garbage". **calloc,** on the other hand, "zeros out" the bytes within the allocated area.

  2) **calloc** takes 2 arguments instead of 1, the product of which will determine the number of bytes allocated.

* Both **calloc** and **malloc** return a character pointer, so if a different data-type pointer is needed, the return value must be cast accordingly. For example:

```
int *ip;

ip = (int *) calloc(57,2);
```

In this case, the pointer is an integer. The value returned by **calloc** must be *cast* as an integer pointer (rather than the default character pointer) before the proper value can be assigned to **ip**. If this is not done RUN/C will generate the message:

```
Illegal operands combination
```

* The returned value should always be checked for a NULL before continuing with the use of allocated memory.

RETURNS NULL, if insufficient memory is available.
Pointer to block of memory, if successful.

EXAMPLE
```
main() /* calloc.c -- demonstrates calloc() */
{
char *memloc, *p;
int i;

memloc = calloc(10,10);
p = memloc;
if (p != NULL)
     for (i = 0; i < 100; i++)
          printf("%d ", *p++);
     i = free(memloc);
     if (i == 0)
          puts("\nSuccessful release");
     else
          puts("\nRelease failure");
     puts("Correct output: 100 zeroes and
     successful release");
}
```

SEE ALSO malloc, free

PURPOSE Controls branching within a **switch** statement.

FORM

```
int c;

switch(c) {
     case '0':
          (statement);
          break;
     case 'z':
          (statement);
          break;
     default:
          (statement);
     }
```

USAGE **case** helps to control the action path within a **switch** statement. A specific constant is evaluated in each **case** statement. In this respect the **switch-case** statements are similar to the **else-if** construction. The main difference, as you probably have noted, is that **switch-case** allows many alternatives built into an easy to understand decision-tree system. The limitation, which you probably also noted, is that the cases are limited to single value constants.

NOTES

* In plain English, **case** works like this:

  *if case **a** ... do this;*
  *if case **b** ... do that;*
  if none of the above cases apply
  ... take default action route if it exists ...
  or if not, do nothing at all.

* A successfully-matched **case** causes execution to go from that point until either the end of the **switch** block is reached or a **break** statement is found. This allows the likes of:

```
int c;
switch(c){
     case 'A':
     case 'B':
          puts("Case A or B");
          break;
     case 'C':
          puts("Case C");
     }
```

But this feature means that **break** statements are needed to keep from executing the contents of subsequent **case** statements.

* **case** statements are only used within **switch.**

* If **default** is not used and no cases match the master expression in **switch,** program control passes out of **switch** with no action being taken.

* **case** statements can be in any order, but **default,** if it appears, should be after the last case.

RETURNS Nothing.

EXAMPLE See example under **switch.**

SEE ALSO switch

PURPOSE Calculates the closest whole number just above a double precision number.

FORM

```
double result;
double number;

result = ceil(number);
```

USAGE **ceil** returns the smallest integer value that is greater than the supplied double precision number. This has the effect of rounding off a decimal number to the next nearest full number.

NOTES In RUN/C Professional this and other transcendental math functions are available in the loadable library MATHLIB.

RETURNS Smallest double value greater than or equal to the double-precision number.

EXAMPLE

```
main()      /* finds ceil of a decimal number */

{
     double x;
     int result;

     x = 14239.5324;

     result = ceil(x);
     printf("%d", result);
}
```

SEE ALSO `floor`

PURPOSE Creates a variable limited to a single character value (a positive decimal value of 0 to 255).

FORM

```
[1] char a = 32;

[2] char *p;

[3] char s[80] = "this is my string";
```

USAGE **char** is a data type limited to a single positive 8-bit character value from 0 to 255 (the ASCII character set).

In Form [1], **char a** is declared and set equal to 32, a space. In [2], character pointer **p** is declared. It may later be set equal to other values. In [3], an 80-element character array is declared and the string, followed by a NULL value, inserted in the first 17 elements of the array.

NOTES In C, simple data type names (such as **char**) have four distinct uses, namely:

a) in declaring variables and aggregate numbers;

b) in casting one type of value to another such as:

```
c = (char)65;
```

c) as an argument to **sizeof**, as in:

```
int i = sizeof(char);
```

d) as an element in a **typedef** statement used in creating a new data type or as an alias for an existing data type such as:

```
typedef char ASCII;
```

(Continued next page)

EXAMPLE

```
main() /* char.c -- 2 uses of char data type */
{
char *string;
char character;
character = 'a';

string = "test of time";

printf("%c %s", character, string);

     /* result will be: "a test of time" */
}
```

SEE ALSO double, float, int, long, short, unsigned

PURPOSE Displays all RUN/C commands.

FORM **CLIST**

USAGE **CLIST** displays the names of all RUN/C commands on the screen. This listing can only be called from the Command level.

NOTES

* The repertoire of commands below may not be accurate. The on-line CLIST command provides the most current listing.

* In RUN/C Professional **CLIST** can be called from the executing program using the **runc()** function.

EXAMPLE

```
Ok

CLIST

?           EXEC        LIBLOAD     NLIST       SYSTEM
@           FILES       LIBS        NLLIST      TAB
AUTOSAVE    FLIST       LIBSHOW     PREC        TRACE
BREAK       FRE         LIBU        PROFF       TROFF
CLIST       HELP        LIBUNLOAD   PRON        TRON
CLS         KILL        LIST        RUN         TYPE
DIR         LIB         LLIST       SAVE        UNLINK
E           LIBD        LOAD        SET         VAL
EDIT        LIBDIR      NEW         SHELL       VER
EDITOR      LIBL

Ok
```

SEE ALSO FLIST, HELP

PURPOSE Closes a file.

FORM
```
int status;
int file;

status = close(file);
```

USAGE **close** concludes *Level 1* I/O to a particular file and frees the automatically assigned file number for use in accessing another file.

If a file is accessed with **open, close** must be used if the file was modified in any way. Otherwise, the end-of-file marker and the data on disk assigned to that file name may not be properly updated.

In the above sample:

`int status;`
declares a variable to hold the returned code.

`int file;`
declares a variable to hold the automatically assigned file number.

`status=close(file);`
closes the file. The returned value from the function will be assigned to **status.**

If the file is closed successfully, 0 will be returned. Otherwise, an EOF will be returned.

The most likely reason for an EOF error is that an attempt was made to close a file which had not been opened or which had been improperly opened. For example, it is illegal to use **close** to close a file which was opened for *Level 2* buffered I/O by **fopen.**

RETURNS EOF, if an error has been detected.
0, if successful.

EXAMPLE

```
main()

    /* close.c -- illustrates use of close() */

{
char name[20];
char buf[1];
int fin;
int i;

    printf("Name please: ");
    gets(name);
    if ((fin = open(name,0)) == -1)
    {     printf("No such file: %s\n",name);
          exit();
    }

    while (0 != (i = read(fin, buf, 1)))
          {
          if (i <0){
                printf("error on read\n");
                close(fin);
                exit();
                }
          else putchar(buf[0]);
          }

    close(fin);
}
```

SEE ALSO open, fclose

PURPOSE Clears the error flag for the specified file.

FORM

```
FILE *fp;

clrerr(fp)
```

USAGE **clrerr** is used to reset the error flag for a specified file once an error has been detected while doing *Level 2* (buffered) I/O.

Generally, file errors are caused by disk full conditions and detected by **ferror.** Once an error has been detected, the status of the file must be reset by **clrerr** before the file can be accessed again. If this is not done, error conditions will continue to be reported, even if the condition which caused the error has been corrected.

NOTES Every effort should be made to correct the condition which caused the error report, prior to reattempting the operation which generated the error code.

RETURNS Nothing.

EXAMPLE See example under ferror.c.

SEE ALSO fopen, ferror

PURPOSE Clears the screen from the command line.

FORM **CLS**

USAGE **CLS** clears the screen and positions the cursor at the top of the newly cleared screen. Unlike the **NEW** command, **CLS** does not affect memory contents. The primary purpose of **CLS** is to make it easier to list program lines from the top of the screen.

NOTES

* The **CLS** command should not be confused with the **cls** function which clears the screen from within the user's program.

* In RUN/C Professional **CLS** may be called from the executing program using the **runc()** command.

EXAMPLE

```
Ok

CLS

Ok
```

SEE ALSO NEW, cls

PURPOSE Clears the screen.

FORM `cls();`

USAGE **cls** clears the screen and positions the cursor at the top of the newly cleared screen.

NOTES

* **cls** is not commonly found in the libraries of C compilers because the code for clearing the screen varies greatly from terminal to terminal. (In UNIX, the file which contains a collection of the various clear screen and other terminal commands is aptly named "curses".)

* If the RUN/C built-in **cls** function is included in a file which you attempt to compile, an error message will be generated during the compilation process unless you include a clear screen function of the same name. Such a file is included on the RUN/C distribution disk in two forms: RCIBMLIB.C, for IBM PC computers and compatibles and RCANSBLIB.C for systems supporting the ANSI screen addressing standard. See Section 1 for more information on ANSI.SYS.

RETURNS Nothing.

EXAMPLE

```
main()
{cls();}
```

SEE ALSO **CLS**

PURPOSE Causes program control to continue with the next iteration of the loop, if any.

FORM

```
while(. . .)
     {
          statement_1;
          if(. . .)
               continue;
          statement_2;
     }
```

USAGE **continue**, somewhat paradoxically, ends the current pass through a **while, do** or **for** loop. Program execution resumes with the next iteration of the loop.

In the above sample, if the expression contained within the **if** statement's test is true, **continue** will cause program execution to skip to the end of the enclosing loop which brings it back to the test in the **while** statement. If the test expression within the **if** is false, execution will proceed to **statement_2.**

RETURNS Nothing.

EXAMPLE

```
main()
{    /*  continue.c  --  illustrates  use  of
     continue; */

int i;

for(i = 1; i<6; i++){
     if(i == 3)
     continue;
     printf("%d ",i);
     }
/* correct answer is 1 2 4 5 */
}
```

SEE ALSO break

Math function

PURPOSE Calculates the cosine of a number.

FORM

```
double result;
double number;

result = cos(number);
```

USAGE **cos** returns the cosine of a number expressed in radians. Both the passed number and the returned value are expressed as doubles.

RETURNS Cosine of the number.

NOTES In RUN/C Professional this and other transcendental math functions are available in the loadable library MATHLIB.

EXAMPLE

```
main() /* cos.c -- computes cosine of a number */
{
double x;
double result;
x = 23.0;
     result = cos(x);
     printf("cosine equals %f", result);

          /* Correct answer is: -0.532946 */
}
```

PURPOSE Calculates the hyperbolic cosine of a number.

FORM

```
double result;
double number;

result = cosh(number);
```

USAGE **cosh** computes the hyperbolic cosine of a number expressed in radians. Both the passed number and the returned value are expressed as doubles.

NOTES In RUN/C Professional this and other transcendental math functions are available in the loadable library MATHLIB.

RETURNS Hyperbolic cosine of the double-precision number.

EXAMPLE

```
main()
{    /* cosh.c -- computes cosine of a number */

double x;
double result;

     x = 0.23;

     result = cosh(x);
     printf("%f",result);

/* Correct answer is: 1.026567 */
}
```

PURPOSE Creates a new disk file.

FORM

```
int file;
char *filename;
int mode;

file = creat(filename, mode);
```

USAGE **creat** declares and names a new file. The file name must follow MS-DOS naming conventions. For more information on proper file names see Section 1 and your DOS manual.

The primary difference between **open** and **creat** is that **open** can only be used on existing files. **creat** can be used on existing files or otherwise, but whenever it is used any existing data in the file will be deleted. Both the current file position pointer and the end-of-file pointer are set to zero (indicating an empty file).

In the above sample:

`int file;`
declares a variable to hold the file number.

`char *filename;`
declares a pointer to the address of the file name.

`int mode;`
declares a variable to hold the selected mode.

`file = creat(filename, mode);`
a file is opened under **filename** for a particular mode of operations (see below). The returned value from the function will be assigned to file.

(Continued next page)

The following values may be used for the mode:

Decimal	Hexadecimal	Meaning
0	0x0	Read only, with text file translation.
1	0x1	Write, with text file translation.
2	0x2	Read/Write, with text file translation.
-32768	0x8000	Read only, without text file translation.
-32767	0x8001	Write, without text file translation.
-32766	0x8002	Read/Write, without text file translation.

NOTES

* *Text file translation* refers to the replacement of newlines with carriage return/linefeed sequences on "writes" and vice-versa on "reads".

* When the returned value is evaluated, a value of 0 or greater means success. An unsuccessful effort will result in an EOF being returned.

RETURNS EOF, if an error has been detected.
File number to access file with, if successful.

(Continued next page)

EXAMPLE

```
main()
{    /* creat.c -- illustrates use of creat() */

char name[20], nameout[20];
char buf[1];
int fin, fout;

     printf("Input name, please: ");
     gets(name);
     if ((fin = open(name,0)) == -1)
     {     printf("No such file: %s\n",name);
           exit();
     }
     printf("Output name, please: ");
     gets (nameout);
     if ((fout = creat(nameout, 1)) == -1)
           exit();
     while(read(fin, buf, 1) >= 0)
     {     putchar(buf[0]);
           write(fout,buf,1);
           if (buf[0] == 26) {
                 close(fin); close(fout);
                 }
     }
     close(fin); close(fout);
}
```

SEE ALSO open, fopen

PURPOSE Displays the current values of active variables after the program has been interrupted.

FORM

```
Break in line n
     Continue, Dump, Ldump, Interactive, Help or
     End (C,D,L,I,H,E)? D
```

USAGE Dump, a mini-menu command, allows you to determine the value of active variables after you have interrupted program execution by pressing <Ctrl-c> or if the program encounters a breakpoint. When the **Dump** option is selected the variable values are displayed in three categories: 1) automatic variables declared within the current function; 2) automatic variables within the calling function or functions; and 3) global and static variables, including file pointers.

This line-oriented debugging aid is useful when trying to determine why your program is going awry.

This option is available only after a program's execution has been interrupted. Upon such interruption, the following menu will appear:

```
Break in line n
     Continue, Dump, Ldump, Interactive, Help or
     End (C,D,L,I,H,E)?
```

If **D** is selected, the value of all active variables will be displayed. Then the mini-menu will reappear, allowing you to select another option.

For more information on the use of <Ctrl-c> see Section 1. For more information on the use of the **BREAK** command in conjunction with the **runc()** function see Section 6.

NOTES

* Use the <Ctrl-s> display toggle to keep the DUMP display from scrolling by faster than you can read it.

* An alternative to using **L(Ldump)** mini-menu command is to use the <Ctrl-p> (or <Ctrl-PtrSc> on PC compatibles) key combination in conjunction with **D(Dump)** to cause the printer to echo screen output.

EXAMPLE

```
NLIST

double angle;
main()
{
int i = 0.0;

        for (angle = i; angle < 3.1416; angle += 0.1) {
                prsin(angle);
        }

}

prsin(d)
double d;
{
        printf("%lf\n",asin(sin(asin(sin(asin(sin(d)))))));
}

RUN

0.000000
0.100000
0.200000
0.300000
0.400000
0.500000
0.600000
0.700001
0.800000
```

At this point we interrupt the display of these numbers by pressing <Ctrl-c>. The following choices appear:

```
Break in line 16
        Continue, Dump, Ldump, Interactive,
        Help or End (C,D,L,I,H,E)?
```

If D is selected, the following will be displayed:

```
Automatic variables of current function:
d:  double 0.800000

Automatic variables of calling function(s):
i:  int 0

Global and static variables:
angle:  double 0.800000

stderr:     file pointer at 240312
stdout:     file pointer at 240294
stdin:      file pointer at 240276
```

The current names and values of automatic variables of the current function, the calling function and any active global and static variables are displayed. Then we continue with the rest of the program:

```
      Continue, Dump, Ldump, Interactive, Help or
End (C,D,L,I,H,E)? C
```

```
0.900000
1.000000
1.100000
1.200000
1.300000
1.400000
1.500000
1.541593
1.441593
1.341593
1.241593
1.141593
1.041593

.. etc.
```

Ok

SEE ALSO C(Continue), E(End), H(Help), I(Interactive), L(Ldump), TRACE, TRON, PRON

PURPOSE Specifies a particular course of action when none of the **cases** within a **switch** statement matches the value given.

FORM

```
int c;

switch(c) {
     case '0':
          (statement);
          break;
     case 'z':
          (statement);
          break;
     default:
          (statement);
     }
```

USAGE **default** provides an alternative course of action if none of the cases within **switch** matches the search constant. If **default** is not used and no cases match the search constant, program control will leave **switch** without any action being taken.

LIMITS RUN/C requires that the optional **default** statement be placed after the last **case** statement within **switch.**

RETURNS Nothing.

EXAMPLE See example under **switch.**

SEE ALSO switch

PURPOSE Displays the names of specified files on a drive/directory.

FORM

[1] DIR

[2] DIR filespec

[3] DIR drive: \directory\ filespec

[4] DIR drive:

USAGE **DIR**, as used in Form [1] above, provides an alphabetical display of the files on the default drive/directory. When a *filespec* or drive/directory is provided, as shown in Forms [2] and [3], all files matching the filename are displayed. Standard naming conventions for file names, drive names, and directory pathnames as described in Section 1 apply.

The specified filespec may contain DOS directory "wildcards": the asterisk (*) and the question mark (?). An asterisk will match any character and all following characters in its position in a file name or extension. See Example 2, below. A question mark will match any character in the file name or extension in its same exact position. See Example 5, below.

When a pathname is specified, the drive/directory will be displayed just above the listing of the files. See Example 4, below.

Use of the quotation marks before and after the filespec is optional; however where quotation marks are used there must be a space between **DIR** and the (quoted) filespec.

FILES is a synonym for **DIR**.

NOTES In RUN/C Professional **DIR** can be called from the executing program using the **runc()** function.

LIMITS When the **DIR** command attempts to list a directory or portion thereof, the names of the files are arbitrarily broken up into alphabetical groups of about 90 file names each.

EXAMPLES

	Command	Effect
[1]	**DIR**	Displays all files on the current (default) drive/directory.
[2]	**DIR *.c**	Displays only files with the extension ".C" on the current drive and directory.
[3]	**DIR B:**	Displays all the files on the current directory of the "B:" drive.

[4] **dir d:\cprogr\trial\test*.***

Displays all files beginning with "TEST" on the **cprogr** directory, **trial** subdirectory.

```
D:\cprog\trial\
TEST 1.3
TEST 2.2
Ok
```

[5] **DIR "TESTC??.C"**

Lists each file on the current drive/directory which has a filename beginning with "TESTC" followed by any two other characters, and with an extension of ".C".

SEE ALSO SHELL, FILES

PURPOSE Executes a statement or series of statements for so long as the tested expression remains true.

FORM

```
do
     statement(s)
while(condition);
```

USAGE **do .. while,** executes one or multiple statements, then evaluates a condition or conditions. The major difference between a regular **while** statement and the **do .. while** *construct* is that in the former case the condition is evaluated first; in the latter, the operation is performed prior to the initial evaluation. Thus with **do .. while,** the statement or statements will be performed at least once; in a **while** statement, the operation will never be performed if the test condition evaluates to false.

NOTES Care must be exercised to insure that the **while** condition can eventually reach a false state; otherwise the loop will continue indefinitely unless interrupted by a **break, continue, goto** or from the keyboard.

RETURNS Nothing.

EXAMPLE

```
main()
{
int i;
i = 1;

do
     printf("%d\n", i++);
while (i <= 5);
}
```

SEE ALSO break, while

PURPOSE Copies the system date to a string in the specified form.

FORM

```
char *sysdate;

int format;
char *prt_ret;

prt_ret = dosdate(sysdate, format);
```

USAGE **dosdate** obtains the date known to DOS and places it in the target string in the display format selected. With a NULL-byte terminator, **prnt_ret** points **past** the received data.

In the above sample:

`char *sysdate;`
declares a pointer to the system date.

`int format;`
declares a variable to hold the selected format code.

`char *prt_ret;`
declares a pointer to hold the returned value.

`prt_ret = date(sysdate, format);`
calls the **dosdate** function which determines the system date and formats it according to the selected mode.

The function will return the formatted system date and assign it to the area in memory pointed to by **sysdate.**

(Continued next page)

The following chart depicts **dosdate**'s formatting modes and their results:

format value	format	example
0	BCD (yy mm dd in 3 bytes)*	841208
1	yymmdd	841208
2	yy/mm/dd	84/12/08
3	yy-mm-dd	84-12-08
4	MON dd, yyyy	DEC 8, 1984
5	Month, dd, yyyy	December 8, 1984
6	dd MON yy	08 DEC. 84
7	dd MON yyyy	08 DEC. 1984

NOTES

* In the formats above **MON** refers to a three-character abbreviation of the month; **mm** references the month by number; **yyyy** refers to the full year (four numerals); **yy** refers to the last two digits of the calendar year, i.e., 85. **dd** refers to the day by number.

* Be sure the string you supply is long enough to hold the date in the desired format.

* The function **dosdate** is similar to the function **date()** in Lattice's C Food Smorgasbord. The object library file or source code from these utility functions is needed in order to compile a RUN/C program using **dosdate.** In addition, the name of the function in your program will need to be changed from **dosdate** to **date.**

* The system date is most conveniently set at the DOS level with the built-in DOS **DATE** command. You can also set the date within a RUN/C program using a system interrupt. See **intdos** and DOS manual for details on setting the date.

* Binary coded decimal (BCD) considers one byte or two nybbles (of 4 bits each) at a time. In BCD the range of allowable nybble values is limited to 0-9; there are two digits to a byte.

---

* Returned in 3 binary-coded decimal (BCD) bytes.

RETURNS Updated pointer to next byte in string.

EXAMPLE

```
main()      /* dosdate.c -- tests various formats
               for date */
{
char ac[80];
int format, n;
char *acpoint;

for(n = 1; n < 8; n++){
     format = n;
     dosdate(ac, format);
     puts(ac);
     }
}
```

SEE ALSO dostime, SHELL

PURPOSE Copies the system time to a string in the specified form.

FORM

```
char *systime;

int format;
char *ptr_time;

ptr_time = dostime(systime, format);
```

USAGE **dostime** obtains the time of day according DOS's internal clock and places it in the target string with a zero-byte terminator in the display format selected.

In the above sample:

`char *systime;`
initializes a pointer to the system time.

`int format;`
initializes a variable to hold the selected format code.

`char *ptr_time;`
initializes a pointer to hold the returned value.

`ptr_time = dostime(systime, format);`
calls the **dostime** function which determines the system time and formats it according to the selected mode.

The function will return the formatted system time and assign it to the area in memory pointed to by **systime.**

The following chart shows the formatting modes and their results:

(Continued next page)

format value	format	example
0	BCD (hh mm ss dd in 4 bytes)*	16 16 10 80
1	hhmmss	015120
2	hh:mm:ss	01:51:20
3	hhmmssff	01512069
4	hh:mm:ss.ff	01:51:20.91
5	hh:mm	01:51
6	hh:mm.ss pp	01:51:21 am
7	hh:mm pp	01:51 am

RETURNS Updated pointer to next byte in string.

NOTES

* In the formats above **hh**, the hour, is relative to midnight or noon. In addition, **hh** is "12" both at midnight and at noon. **pp** is am or pm. **mm** is minutes and **ss** is seconds. **ff** is hundredths of a second.

* Be sure the string you supply is long enough to hold the time in the desired format.

* The function **dostime** is similar to Lattice's C Food Smorgasbord function named **time()**. The object library file or source code from these utility functions is needed in order to compile a RUN/C program using **dostime**. In addition, you will need to change the name of the function in your program from **dostime** to **time**.

* The system time is set at the DOS level, although you can set the time from within RUN/C using **SHELL** with the command:

```
SHELL TIME
```

(See your DOS manual for details on setting the time.)

* Returned in binary-coded decimal (BCD) digits (4 bytes).

EXAMPLE

```
main()
{    /* dostime.c -- illustrates formats of
     dostime() */

char ac[80];
int format, n;
char *acpoint;

for(n = 1; n < 8; n++){
     format = n;
     dostime(ac, format);
     printf("At the return, the time is: ");
     puts(ac);
     }
}
```

SEE ALSO dosdate, SHELL

PURPOSE Creates a variable with a range of 15 to 16 decimal places of precision (+/-10e-307 to +/-10e308).

FORM
```
double a;
a = 23.293472394;
```

USAGE **double** provides the most precise data type available in C. It also is the most memory-consuming data type, taking up 64 bits each time it is declared. Using **double**, accurate calculations to the 15th or 16th decimal place can be obtained.

NOTES

* A **double** can be positive or negative. Positive is assumed unless a minus sign precedes the number assigned to the variable.

* In C, simple data type names (such as **double**) have four distinct uses, namely:

  (1) in declaring variables and aggregate numbers;

  (2) in casting one type of value to another such as:

```
d = (double) 0;
```

  (3) as an argument to **sizeof**, as in:

```
int i = sizeof(double);
```

  (4) as an element in a **typedef** statement used in creating a new data type or as an alias for an existing data type such as:

```
typedef double BIG;
```

SEE ALSO `char, float, int, long, short, unsigned`

PURPOSE Terminates execution of a program after it has been interrupted.

FORM

```
Break in line n
     Continue, Dump, Ldump, Interactive, Help or
     End (C,D,L,I,H,E)? E
```

USAGE End, a mini-menu command, allows you to terminate program execution after you have pressed <Ctrl-c> to interrupt program execution or after a breakpoint has been encountered. Upon interruption the following message will appear:

```
Break in line n
     Continue, Dump, Ldump, Interactive, Help or
     End (C,D,L,I,H,E)?
```

If **E** is selected, the program will terminate and the **Ok** prompt will reappear, ready for further commands.

For more information on the use of <Ctrl-c> and see Section 1. For more information on the use of the **BREAK** command in conjunction with RUN/C Professional's **runc()** function see Section 6.

NOTES

* The program can be run again using the **RUN** command.

* All buffered (*Level 2*) files are automatically closed when **E**(End) is used.

(Continued next page)

EXAMPLE LIST

```
main()

{

int a;

for(a = 0; a < 40; ++a)
     printf("%d ",a);

}

RUN

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

At this point we interrupt the display of these numbers by pressing <Ctrl-c>. The following choices appear:

```
Break in line 7
     Continue, Dump, Ldump, Interactive, Help or
     End (C,D,L,I,H,E)? E
```

If E is selected, program execution will terminate and

```
Ok
```

will reappear, signifying re-entry to the Command Mode.

SEE ALSO C(Continue), D(Dump), H(Help), I(Interactive), L(Ldump), Section 6

PURPOSE Calls RUN/C's built-in screen editor.

FORM

[1] EDIT

[2] E

[3] EDIT n

[4] EDIT END

[5] EDIT ERROR

USAGE **EDIT** and its shorter form **E** call the screen editor built into RUN/C. The editor offers search-and-replace, block move and WordStar and BASIC editor compatibility. It is fully described in Section 2 of this manual; highlights of that discussion will be repeated here.

The five forms of calling the built-in editor are:

[1]	**EDIT**	Editing will begin either on the first line of the program in memory or, if the program has previously been edited, at the last position of the cursor.
[2]	**E**	The abbreviation **E** can always be used as an alias for the **EDIT** command.
[3]	**EDIT *n***	The editing cursor will be placed at column 1 of the specified line. If "1" is selected, editing will begin at the top of the program. If a line number equal to or greater than the number of lines in the program, editing will begin on the last line.
[4]	**EDIT END**	Editing will begin at the end of the program.
[5]	**EDIT ERROR**	Editing will begin on the program line where the last error was detected. In addition, the most recent error

message will appear on the third line of the editor's command menu. If no error has occurred, editing will begin at the last location of the cursor. If the error cannot be identified with a line number, the cursor will be positioned on the last program line.

The line specified is the autoline number assigned by the interpreter. A line is always terminated by a newline (carriage return/linefeed) and may be up to 250 characters long (although keeping lines to a maximum of 80 characters will simplify your editing considerably). It is not possible to edit or otherwise directly change an autoline number.

Once you are editing your program, text may be changed, deleted, inserted, copied, moved, written to disk or obtain from disk. Any changes to your program are automatically made in the interpreter and will be reflected when the program is **RUN**. Of course the changes won't be "permanent" until you **SAVE** your program under its current or another named.

NOTES

* On IBM PC compatible systems, several function keys are dedicated to invoking the editor. In particular, <F6> will produce the string **EDIT ERROR.**

* There is at present no "UNDO" command, so text should be deleted or written over with care.

EXAMPLE See Section 2.

SEE ALSO EDITOR, LIST, Section 2

PURPOSE Allow use of a user-selected substitute full-screen editor.

FORM EDITOR

USAGE **EDITOR** is a special command which allows you to integrate the text editor or word processing program of your choice into the RUN/C program creation process. The four steps in using the **EDITOR** command are:

1) Select an appropriate editor.
2) Create **EDITOR.**
3) Enter **EDITOR.**
4) Save your newly edited program and exit **EDITOR.**
5) Re-enter RUN/C with your newly edited program.

Taking these in order:

Selecting an appropriate editor. Your editor must be capable of accepting a filename to be edited as a command line argument, as in:

```
WS MYFILE.DOC
```

or

```
VEDIT TEST.001
```

Editors without this capability will not automatically load the program saved by RUN/C as PROGRAM.C (see below).

If your EDITOR has any *overlay* files, these too will need to be on the default drive directory.

Creating **EDITOR.** For **EDITOR** to work properly, a text editor or word processing program named either EDITOR.EXE or EDITOR.COM (depending upon whether the program you are attempting to rename is an ".EXE" or a ".COM" file) must be on your default drive/directory. Alternatively, the **SET EDITOR** command, described in this Section and Section 2, may be used. Another alternative is described in NOTES, below.

If you choose to rename your editor simply use DOS' RENAME command:

```
REN WS.COM EDITOR.COM
```

will do the job for WordStar.

Entering EDITOR. When you give the command:

```
Ok
EDITOR<enter>
```

from the RUN/C command line the following operations are performed:

A. The program in memory is saved to the default drive/directory under the file name PROGRAM.C. (If a file of that name already exists, it will be renamed PROGRAM.BAK and any pre-existing PROGRAM.BAK erased.)

B. The text editor or word processor named EDITOR will be loaded into memory along with PROGRAM.C. RUN/C will remain resident in memory, but will not be accessible until you leave your EDITOR.

Saving your newly edited program and exiting EDITOR. After you have finished editing the program save it using the save commands which are standard for the screen editor or word processor you are using. Then exit the EDITOR with the commands you would normally use to return to the operating system.

For example, the command <Ctrl-k Ctrl-x> in WordStar will save the file in memory and return to the operating system level.

Re-entering RUN/C. As you exit EDITOR, RUN/C will automatically regain control of the system and reload the newly edited PROGRAM.C. The message:

```
Loading line n
(Back in RUN/C)

Ok
```

will be displayed. You can use the LIST or EDIT commands to verify that your modified program is in memory.

NOTES

* There must be sufficient memory for the editor and RUN/C to be co-resident. For more information on this subject, see the discussion of **EDITOR** and **SHELL** in Section 2.

* If PROGRAM.C has been deleted from disk, no text will appear when RUN/C returns.

* If no **EDITOR** is available the message:

```
Bad command or file name
Press any key to return to RUN/C . . .
```

will be displayed. After you press a key the unaltered PROGRAM.C will be reloaded.

* If no program is in memory, the message:

```
Can't go to editor without program
```

will appear when an attempt is made to access **EDITOR**. In some situations, of course, it is desirable to go to your full screen editor without a program in memory. To do this simply use **EDIT** to enter RUN/C's screen editor and create a program consisting of one or two blank lines. Then try **EDITOR** again.

* Editors (e.g., WordStar in document mode) may insert some "high bit set" (ASCII greater than 127) characters in the text. Generally this presents no problem, since RUN/C resets ("zeros") the high bit of every character when it loads a file. Some word-processing programs, however, may be entirely unsuitable for a cameo role as "EDITOR".

* Since the special file PROGRAM.C is used by **SHELL, SAVE, AUTOSAVE** as well as **EDITOR**, it is important not be become dependent on the illusory permanence of the contents of this generic file. Frequently save your program to the file name of your choice.

* A faithful RUN/C user pointed out a way to use your favorite editor without having to rename the file or use the **SET EDITOR** command. Here's how:

Set up a batch file named EDITOR.BAT and containing only

edname %1

where **edname** is the name of your editor (without extensions). RUN/C's EDITOR command will then invoke your editor with PROGRAM.C via the batch file assuming that your favorite editor is on the current disk/directory or the *DOS path* and that any *overlay* files required by your editor are in the appropriate drive/directory. If in doubt about the available of your program through the pathname system, exit RUN/C and attempt to invoke your editor from the DOS command line.

(Continued next page)

EXAMPLE NLIST

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

RUN
Hello, world!

Ok

EDITOR
```

(The full screen editor is entered, the program is edited and saved according to the editor's requirements. When the editor is exited, RUN/C regains control, signified by the message:

```
Press any key to return to RUN/C
Saving line n
Ok
```

This time when the program is listed and run the altered program appears.)

```
NLIST

main()
{
      printf("Hello, C!\n");
}
Ok

RUN

Hello, C!
Ok
```

SEE ALSO Section 2 and EDIT, SET EDITOR, SHELL, AUTOSAVE

PURPOSE Specifies an alternative course of action if the conditional part of an **if** statement is false.

FORM

```
if(expression)
     statement_1;
else
     statement_2;
```

USAGE **else** is the "or" to **if**'s "either", since only one of the two statements associated with an **if-else** combination will be executed. In the above sample, if the expression following the **if** is true, **statement_1** is executed. If the expression following the **if** is false, the interpreter looks for the next keyword or expression. If it is an **else**, then **statement_2** is executed.

NOTES

* In plain English, the **if-else** combination is:

  if (this condition is true)
  execute this statement(s);
  else
  execute this statement(s);

* **else** should not be used without a preceding **if** statement. If it is, the error message:

  `'else' without 'if'`

  will appear.

* **else** can be used with another **if** statement. See **else if** below.

RETURNS Nothing.

EXAMPLE

```
main()      /* else.c -- illustrates use of else */
{
int number, remainder;

     printf("Enter any ole odd number.\n");
     scanf("%d", &number);

     remainder = number % 2;

     if(remainder == 0)
          printf("%d!   Now   we're   even.\n",
          number);
     else
          printf("There's something odd about
          %d. Try again.\n", number);
}
```

SEE ALSO if, else if

PURPOSE Allows for multiple choice decisions within the **if ... else** decision system.

FORM

```
if (condition)
     statement_1;
else if (condition)
     statement_2;
else if (condition)
     statement_3;
else if (condition)
     statement_4;
else
     statement_5;
```

USAGE **else if** provides for a "multiple choice" decision structure based on the evaluation of a primary condition and a series of alternative conditions. If the first condition is true, the course of action will be taken and none of the subsequent **else if** or **else** statements will be evaluated. If the condition(s) tested by the initial **if** is false, the conditions of one or several **else if** statements are tested until one of them evaluates to true or the tests have been exhausted. If no true condition is found and if an **else** statement is the last statement in the series, it will be executed, otherwise no action will be taken.

NOTES Strictly speaking, **else if** is not a statement in the C language; it is rather the result of the logical combination of the **else** and **if** statements. Practically speaking, most C programmers will probably share our view that the construction merits its very own entry in this section.

RETURNS Nothing.

(Continued next page)

EXAMPLE

```
main()
{
/*
     elseif.c -- illustrates use of else if
     keyword combination
*/
int number;
char sign;

     printf("Please type in a number.\n");
     scanf("%d", & number);

     if (number < 0)
               sign = '-';
          else if (number == 0)
               sign = '0';
          else
               sign = '+';
          printf("Sign is %c\n", sign);
}
```

SEE ALSO if, else

PURPOSE Simultaneously loads and executes the named program.

FORM [1] EXEC *filename*

or

[2] EXEC *filename arg1 arg1*

USAGE EXEC provides the ability to combine the LOAD and RUN commands into a single operation. If the program requires any arguments to be passed to it from the command line these are appended after the filename, as shown in sample 2, above.

NOTES * In RUN/C Professional the EXEC command can be called from the program using the special runc() function as in:

```
runc("EXEC main test")
```

which would load and execute the program main.c and pass the string test to it. In RUN/C Professional this opens up the possibility of "chaining" C programs through the interpreter. Here are two usable schemes:

[1]

```
prog1{ ...
       runc("exec prog2"); }

prog2{ ...
       runc("exec prog3"); }

prog3{ ...
       runc(" .. etc.
```

[2]

```
menuprog     /* contains code to get user selection
                and branch to appropriate EXEC */

menuprog { selection }

       prog1{ ...
              runc("exec menuprog"); }

       prog3{ ...
              runc("exec menuprog"); }

       etc.
```

While this may not be the software solution to the Starwars space defense program, it does provide a considerable amount of batch processing capability.

* **EXEC** is the equivilent of invoking RUN/C with a program name from the DOS command line. The only difference is that when a program is invoked from the DOS level loads and runs successfully you will be returned to DOS. When **EXEC** loads and runs successfully, program control ultimately returns to RUN/C.

* Redirected I/O cannot be used in conjunction with the **EXEC** command. Thus:

```
exec main test > CAPTURE
```

is not currently supported.

EXAMPLE See above.

SEE ALSO LOAD, RUN

PURPOSE Flushes buffers for any open buffered files and closes those files, then terminates program execution.

FORM `exit()`

USAGE **exit** terminates execution of the current program after first flushing any output file buffers and then closing any open files which were opened with **fopen.** This function is useful for terminating a program without having to percolate through a series of calling functions.

NOTES

* **exit** closes only those files which are being accessed using *Level* 2 buffered file I/O (**fopen, fclose,** etc.) Files accessed using the *Level* 1 interface (such as **open, close,** etc.) are not closed by **exit.**

* Although **exit** executes an **fclose** for any open *Level* 2 files as part of its duties, it is a good programming practice to close files explicitly whenever there is no longer an immediate need for them to be open.

RETURNS -1, if an error has been detected.
0, if successful.

(Continued next page)

EXAMPLE

```
main() /* exit.c -- illustrates use of exit() */
{
int i = 99;
while (i != 10){
     printf("Type a number between 1 and 10: ");
     scanf("%d", &i);
     testi(i);
     }
puts("10 entered. Done.");
}

testi(i)
int i;
{
     if(i < 1 || i > 10){
          printf("i = %d\n", i);
          puts("(You cheated. I'm quit.)");
          exit();
          }
}
```

SEE ALSO fopen, fclose, fflush

PURPOSE Calculates the exponential value of a number.

FORM

```
double result;
double x;

result = exp(x);
```

USAGE **exp** computes the exponential value of a double-precision number.

This can be expressed algebraically as:

$$result = e^{number}$$

Both the passed number and the result should be double-precision.

NOTES In RUN/C Professional this and other transcendental math functions are available in the loadable library MATHLIB.

RETURNS exponential value to the power of number, if successful.

EXAMPLE

```
main()      /* exp.c -- computes exponential value
               of a number */
{

double x = 1.0;
double y = 0.0;

    printf("exp of 1 is %f; of 0, %f.", exp(x),
    exp(y));

/*  Correct result: 1 is 2.718282; 0 is
    1.000000 */
}
```

PURPOSE Allows the declaration of a variable as an **extern**, meaning that it is defined in another source file which will be available only when compiled program modules are linked together.

FORM `extern char string[], block[];`

USAGE **extern** in a compiled C environment provides a means of using variables in one module which are defined in another module. The linker resolves the situation when the program is compiled.

It is important to note that in RUN/C the **extern** declaration is merely accepted by the interpreter if it occurs globally; the declared variable is treated like any other global variable.

NOTES

* In RUN/C **extern** produces an error which interrupts program execution when used within a function. However only a warning:

  `extern ignored`

  is issued when the term is used globally (outside a function). The warning is significant because of the difference in meaning of **extern** between RUN/C and compiler environments which typically depend upon the successful linking of multiple modules when medium- to large-size programs are being created.

* Similarly to **extern**, the storage class identifier **register** is permitted in interpreted programs, but has no direct effect.

EXAMPLE

```
extern int d = 5;

main()

{
     printf("extern int d = %d", d);
}
```

PURPOSE Calculates the absolute value of a number as a floating point value.

FORM

```
double result;
double number;

result = fabs(number);
```

USAGE **fabs** converts a negative number to a positive number without changing its numeric value. Thus, **fabs** is the equivalent of multiplying a number by its own sign. Both the passed number and the returned value are expressed as doubles.

NOTES In RUN/C Professional this and other transcendental math functions are available in the loadable library MATHLIB.

RETURNS Absolute value of a number as a floating type value.

EXAMPLE

```
main() /* fabs.c -- illustrates use of fabs() */

{
double number;

double result;
     number = -3122.3242;

     result = fabs(number);
     printf("%.4f", result);

/*   Correct answer is: 3122.3242 */
}
```

PURPOSE Concludes interaction to a buffered file.

FORM

```
int ret;
FILE *fp;

ret = fclose(fp);
```

USAGE **fclose**, which stands for "file close", concludes the interaction with a particular file or device. It is, generally speaking, the opposite of **fopen**, since it completes the processing of a file and releases all related resources.

When **fclose** is invoked, the status of the file is checked. If the file has been opened in write mode, any data in the file's memory buffer is written to the file before it is closed. This is known as "flushing the buffer".

In the above sample:

`int ret;`
declares variable **ret** to hold the returned value.

`FILE *fp;`
declares the file pointer.

`ret = fclose(fp);`
closes the file currently pointed to by **fp**. The returned value from the function is assigned to **ret**.

If the operation is successful, 0 will be assigned to the return variable. If it fails, a -1 value will be returned. The most likely reasons for an error are: 1) the file specified was not opened for *Level 2* buffered output via **fopen**; 2) **fp** does not point to a valid file descriptor; or 3) the file could not be closed due to an error at the operating system level such as disk drive not ready, disk error or disk full.

NOTES

* As the last buffer may not be written until **fclose** is called, data may be lost if the file is not properly closed.

* **fclose** operates on a particular file via file pointers. Thus the instruction:

```
fclose(file1);
```

closes the file associated with the *file pointer* named **file1** which points to the actual I/O structure assigned to this disk file.

* **fclose** is called automatically for all files opened with **fopen** when the **exit** function is used. However, it is good programming practice to close files explicitly. The fact that only a finite number (and a surprisingly small finite number at that!) can be simultaneously held open on most DOS systems should provide additional incentive for closing files and devices that have no reason to be open. See you DOS manual for details on the FILES environmental command which allows changing the maximum number of files which may be open from the default (eight on most systems).

RETURNS -1, if an error is detected.
0, if successful.

EXAMPLE

```
main()    /* fclose.c -- illustrates use of
             fclose() */
{FILE *fp;
char name[20], c;

     printf("Filename: ");
     gets(name);
     if ((fp = fopen(name,"r")) == NULL)
          exit();
     while((c = getc(fp)) != EOF)
          putchar(c);
     clrerr(fp);
     fclose(fp);
}
```

SEE ALSO fopen, exit

PURPOSE Tests location of file position marker regarding its end-of-file status.

FORM

```
int ret;
FILE *fp;

ret = feof(fp);
```

USAGE **feof** checks to see if the file position pointer is currently set at the last byte in the file.

In the above sample:

`int ret;`
declares a variable to hold the returned value.

`FILE *fp;`
declares the file pointer.

`ret = feof(fp);`
examines the file pointed to by **fp**, assigning the returned value to **ret**.

If the file pointer is at the end-of-file, a non-zero value will be returned.

NOTES Incorrect end-of-file indications may be received if an error has been previously generated and the error condition was not canceled by **clrerr**.

RETURNS 0, if not end of file.
Non-zero, if end of file.

(Continued next page)

EXAMPLE

```
main()     /* feof.c -- demonstrates feof() */
{
     FILE *fi;
     int c, i;

     if((fi = fopen("test.fil","r")) == NULL) {
          printf("Can't open test.fil\n");
          exit();
     }
     while ((c = getc(fi)) != EOF)
          ;     /* dummy statement */
     if(feof(fi))
          puts("At end of file");
     else
          puts("Not at end of file");
     fclose(fi);
     puts("Correct answer (if test.fil exists):
     At end of file");
}
```

SEE ALSO fopen, ferror, clrerr

PURPOSE Checks the error status of a file.

FORM

```
int ret;
FILE *fp;

ret = ferror(fp);
```

USAGE **ferror** obtains an error condition report on a particular file which has been opened for *Level 2* (buffered) file I/O. If an error condition exists, a non-zero value is returned; otherwise a 0 is returned.

In the above sample:

`int ret;`
declares a variable to hold the returned value.

`FILE *fp;`
sets the file pointer.

`ret = ferror(fp);`
examines the file pointed to by **fp**. The returned value from the function will be assigned to **ret**.

If an error condition exists, a non-zero value will be returned.

Once an error condition is found for a particular file, further calls will result in -1 values being returned until the error condition is cleared with **clrerr**.

RETURNS -1, if an error is detected.
0, if successful.

(Continued next page)

EXAMPLE

```
main()    /* ferror.c -- illustrates use of
          ferror() */
{
FILE *infile, *outfile;
char in_name[20], outname[20], c;
int ret;

     printf("Input filename: ");
     gets(in_name);
     printf("Output filename: ");
     gets(outname);
     if ((infile = fopen(in_name,"r")) == NULL)
          exit();
     if ((outfile = fopen(outname,"w")) == NULL)
          exit();
     while((c = getc(infile)) != EOF){
          if(ferror(outfile) != -1)
               putc(c,outfile);
          else{
               puts("You're file had an error
               condition; we fix");
               clrerr(outfile);
               break();
          }
     }
}
```

SEE ALSO fopen, clrerr

PURPOSE Writes any data being held in a file's buffer to the disk.

FORM

```
FILE *fp;

fflush(fp);
```

USAGE **fflush** writes data from the portion of memory reserved for gathering data which will eventually be written to a file on disk.

In normal file management, **fflush** is used regularly to guard against loss of data due to program crashes, system crashes, power interrupts, operator errors, etc. The file remains open after the function is used.

In the above sample:

`int ret;`
declares a variable to hold the returned value.

`FILE *fp;`
sets up the file pointer.

`ret = fflush(fp);`
moves the file's data held in the buffer to the named file. The returned value from the function will be assigned to **ret.**

If the operation is successful, a 0 is returned. A non-zero value is returned if an error is detected. Causes of an error may include: 1) trying to flush data to a file which was not opened with **fopen;** 2) **fp** not pointing to a valid file descriptor; or 3) the entire contents of the buffer could not be written due to an error at the operating system level such as a disk full.

RETURNS Non-zero, if error.
0, if successful.

(Continued next page)

EXAMPLE

```
main()
{
/*
     fflush.c -- illustrates fflush() by sending
     2 lines to a file
*/
     FILE *fo;
     char dest[25];

     printf("Output filename? ");
     gets(dest);
     if((fo = fopen(dest,"w")) == NULL) {
          printf("Can't open %s\n",dest);
          exit();
     }
     fputs("a short string\n",fo);
     printf("Press a key...");
     getchar();
     fflush(fo);
     printf("\nPress a key...");
     getchar();
     fputs(" and a longer one",fo);
     fclose(fo);
     puts("\nCorrect answer (in file): a short
     string and a longer one");
}
```

SEE ALSO fopen, ferror, clrerr

PURPOSE Reads a single character from the referenced file.

FORM

```
int ret;
FILE *fp;

ret = fgetc(fp);
```

USAGE **fgetc** reads a character from the indicated file.

In the above sample:

`int ret;`
declares a variable to hold the returned value.

`FILE *fp;`
sets the file pointer.

`ret = fgetc(fp);`
gets a character from the file pointed to by **fp.** The returned value from the function will be assigned to **ret.**

The returned value is the character read; EOF, if the end of the file has been reached.

NOTES

* **fgetc** as well as **getc** and **getchar** return integers because they must be able to return a -1 value on sensing end-of-file (a value which a character variable cannot hold since it is limited to positive values 0 through 255). It is a perfectly legitimate, common practice to assign the returned positive integer(s) to character variables.

* In the case of RUN/C, there are no differences between **getc** and **fgetc.** See your C compiler manual for a discussion of the differences and applications of these functions for your compiler.

RETURNS Character value read.

(Continued next page)

EXAMPLE

```
main()
{
/*
     fgetc.c -- demos fgetc() and fputc() by
     copying a file with console echo
*/
{
     FILE *fi, *fo;
     int c;
     char source[25], dest[25];

     printf("Input filename: ");
     gets(source);
     printf("Output filename: ");
     gets(dest);
     if((fi = fopen(source, "r")) == NULL) {
          printf("\tCan't open %s\n",source);
          exit();
          }
     if((fo = fopen(dest, "w")) == NULL) {
          printf("\tCan't open %s\n",dest);
          exit();
          }
     while((c = fgetc(fi)) != EOF) {
          putchar(c);
          fputc(c, fo);
          }
     fclose(fi);
     fclose(fo);
     puts("\n{DONE}");
}
```

SEE ALSO getc, getchar, fgets

PURPOSE Reads *n*-1 characters from a file.

FORM

```
char *p;
char *s;
int n;
FILE *fp;

p = fgets(s, n, fp);
```

USAGE **fgets** reads up to a specified number of characters from a file or until a newline (<enter>) is encountered. A NULL byte is appended to the end of the string if the specified number of characters have been successfully read.

In the above sample:

`char *p;`
declares a character pointer to hold the returned value.

`char *s;`
declares a buffer to hold the input string.

`int n;`
declares a variable to hold the number of bytes to be read.

`FILE *fp;`
sets the file pointer.

`p = fgets(s,n,fp);`
reads up to *n*-1 characters from the current position in the file pointed to by **fp**, assigning the characters to memory pointer **p**.

p will equal NULL if end-of-file or an other error condition is detected. Otherwise **p** will equal **s**. **s** should point to an array of chars with at least *n* bytes.

RETURNS NULL, if an error is detected.
Original string pointer, if successful.

EXAMPLE

```
/* line-at-time file reader */

#define MAXLEN 80

main()
{    /* fgets.c -- demonstrates use of fgets()
     */

FILE *fi;
char source[25], line[MAXLEN], *p;

     printf("Input filename: ");
     gets(source);
     if((fi = fopen(source, "r")) == NULL) {
          printf("\tCan't open %s\n",source);
          exit();
     }
     p = line;
     while(p != NULL) {
          p = fgets(line,MAXLEN,fi);
          printf(line);
     }
     fclose(fi);
     puts("\n{DONE}");
}
```

SEE ALSO gets, fgetc, fputs

PURPOSE Refers to a structure built into RUN/C which declares all the pertinent information needed for file access (buffer size, filename information, etc.)

FORM FILE *fp

USAGE FILE is used in conjunction with a pointer to a file name in *Level* 2 buffered I/O to create and keep track of files being accessed by the program in *Level* 2 (buffered) I/O.

In the above sample:

```
FILE *fp;
```

creates a file pointer named **fp.**

EXAMPLE

```
main()      /* file.c -- illustrates use of FILE
               structure */

{
FILE *fp;
char name[20], c;

     printf("Filename: ");
     gets(name);
     clrerr(fp);
     if ((fp = fopen(name,"r")) == NULL)
          exit();
     while((c = getc(fp)) != EOF)
          putchar(c);
     fclose(fp);
}
```

PURPOSE Displays the names of specified files on a drive/ directory.

FORM [1] FILES

[2] FILES *filespec*

[3] FILES *drive: \directory\ filespec*

[4] FILES *drive:*

USAGE **FILES**, as used in Form [1] above, provides an alphabetical display of the files on the default drive/directory. When a *filespec* or drive/directory is provided, as shown in Forms [2] and [3], all files matching the filename are displayed. Standard naming conventions for file names, drive name, and directory pathnames as described in Section 1 apply.

The specified filespec may contain DOS directory "wildcards": the asterisk (*) and the question mark (?). An asterisk will match any character and all following characters in its position in a file name or extension. See Example 2, below. A question mark will match any character in the file name or extension in its same exact position. See Example 5, below.

When a pathname is specified, the drive/directory will be displayed just above the listing of the files. See Example 4, below.

Use of the quotation marks before and after the filespec is optional; however where quotation marks are used there must be a space between **FILES** and the (quoted) filespec.

NOTES * **DIR** is a synonym for **FILES**.

* In RUN/C Professional **FILES** can be called from the executing program using the **runc()** command.

LIMITS When the **FILES** command attempts to list a directory or portion thereof, the files are divided into groups of about 90 names each and then alphabetized.

EXAMPLES

	Command	Effect
[1]	**FILES**	Displays all files on the current (default) drive/directory.
[2]	**FILES *.c**	Displays only files with the extension ".C" on the current drive and directory.
[3]	**FILES B:**	Displays all the files on the current directory of the "B>" drive.

[4] **files d:\cprogr\trial\test*.***

Displays all files beginning with "TEST" on the **cprogr** directory, **trial** subdirectory:

```
D:\cprog\trial\

TEST 1.3
TEST 2.2
Ok
```

[5] **FILES "TESTC??.C"**

Lists each file on the current drive/directory which has a filename beginning with "TESTC" followed by any two other characters, and with an extension of ".C".

SEE ALSO SHELL, DIR

PURPOSE Displays names of all built-in RUN/C functions.

FORM **FLIST**

USAGE **FLIST** lists the built-in RUN/C functions on the screen.

NOTES

* The repertoire of built-in functions may vary from this list. The on-list **FLIST** command will provide the most current list of available functions.

* In RUN/C Professional **FLIST** can be called from the executing program using the **runc()** command.

EXAMPLE

```
Ok

FLIST

        Built-in functions:
acos    creat   fputs   intdos  isxdigit  printf  sqrt
aget    dosdate fread   intdosx locate    putc    srand
aput    dostime free    isalnum log       putch   sscanf
asin    exit    freopen isalpha log10     putchar strcat
atan    exp     fscanf  isascii lprintf   puts    strcmp
atan2   fabs    fseek   iscntrl lputc     rand    strcpy
atof    fclose  ftell   iscsym  lputs     read    strlen
atoi    feof    fwrite  iscsymf lseek     rename  tan
atol    ferror  getc    isdigit malloc    rewind  tanh
calloc  fflush  getch   isgraph movmem    runc    tolower
ceil    fgetc   getchar islower open      scanf   toupper
close   fgets   gets    isodigit outp     setcom  ungetc
clrerr  floor   inkey   isprint peek      sin     ungetch
cls     fopen   inp     ispunct poke      sinh    unlink
cos     fprintf int86   isspace pow       sprintf write
cosh    fputc   int86x  isupper

Ok
```

SEE ALSO HELP, CLIST

PURPOSE Creates a variable with up to seven decimal places of precision (+/- 10e-37 to +/- 10e38).

FORM

```
float a, b, c, d;
a = 23.293472394;
b = -9273.923;
c = 69.92734;
d = 2.797E-15;
```

USAGE **float** provides for single-precision floating point arithmetic up to seven places of precision and an optional fractional part.

NOTES

* In C, simple data type names (such as **float**) have four distinct uses, namely:

  1) in declaring variables and aggregate numbers;

  2) in casting one type of value to another such as:

     ```
     d = (float) 0;
     ```

  3) as an argument to **sizeof**, as in:

     ```
     int i = sizeof(float);
     ```

  4) as an element in a **typedef** statement used in creating a synonym for an existing data type such as:

     ```
     typedef float DECIMAL;
     ```

* **float** is generally preferred to **double** in situations where it is important to conserve memory; but **double** may execute faster since all floating-point computations are normalized to **double** and then converted back to **float** in RUN/C. This operation takes a little extra time that could become noticeable in extensive computations. The trade-off is that rounding off errors will occur with **floats** after six or seven decimal places.

* A float may be positive or negative. Positive is assumed unless a minus sign *prepends* the number assigned to the **float** variable.

* The construct **long float** is not supported. Use **double** instead.

SEE ALSO char, double, int, long, short, unsigned

PURPOSE Calculates the closest whole number value at or just below a double precision number.

FORM

```
double result;
double number;

result = floor(number);
```

USAGE **floor** returns the largest integer value that is less than or equal to the double number.

NOTES In RUN/C Professional this and other transcendental math functions are available in the loadable library MATHLIB.

RETURNS Largest integer value less than or equal to the double-precision number.

EXAMPLE

```
main()
{  /* floor.c -- demonstrates use of floor() */
     prfloor(0.58);
     prfloor(1.99);
     prfloor(-0.58);
     prfloor(-1.99);
     puts("\nCorrect    output    is:    0.000000
     1.000000 -1.000000 -2.000000");
}

prfloor(d)
double d;
{
     printf("%g ",floor(d));
}
```

SEE ALSO ceil

PURPOSE Opens a file for buffered read/write operations.

FORM

```
FILE *fp;
char *filename;
char *mode;

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

USAGE **fopen** opens a file for *Level* 2 (buffered) input/output (I/O). The filename must conform to DOS conventions, described in the DOS manual and in Section 2. DOS pathnames are supported.

fopen's access mode is specified as a string, and must be one of the following:

Mode	Effect
r	Open an existing file for reading. Input only.
w	Create a new file, or erase and recreate an existing one, for writing. Output only.
a	Create a new file, or append to an existing one, for writing. Output only.
r+	Open an existing file for update (both reading and writing), starting at the beginning of the file. Input/output.
w+	Create a new file, or erase and recreate an existing one, for update (both reading and writing). Input/output.
a+	Create a new file, or append to an existing one, for update (both reading and writing). Input/output.

The mode character must be lower case. Note that the write (**w** and **w+**) options discard any data in the file. The **a** and **a+** options append to the end of an existing file or create a new one.

When a file is opened for updating (**r+**), both reading and writing may be performed on the file. Opening the file to append (**a+**) forces any data in the buffer to be written to the current end of file, regardless of the current file pointer position.

In order to change a file's mode setting, the file must first be closed with **fclose**.

In the above sample:

`FILE *fp;`
declares a file pointer.

`char *filename;`
declares a pointer to the file name.

`char *mode;`
declares a pointer to the access mode.

`fp = fopen(filename, mode);`
opens the named file for the specified mode of operation. The returned value from the function will be assigned to **fp**.

The returned value will either be the automatically assigned file pointer value or a NULL value if an error is detected.

Generally, an error occurs when: 1) an invalid mode was specified; 2) the file was not located; 3) the file could not be created; or 4) the maximum number of files was already open.

NOTES

* A file must exist in order to open it with **fopen** access modes **r** or **r+**. If the file does not currently exist, **w**, **w+**, **a** or **a+** may be used to create it.

* **fopen** opens a file for reading and/or writing in *translated mode.* This means that characters are changed when they are moved from the disk into memory and vice-versa. For example, when reading, a carriage return (0D hex) followed by a linefeed (0A hex) will be translated to just a single linefeed (LF). Thus:

  \r\n becomes **\n**

  When writing the opposite occurs: a linefeed (0Ah) will be expanded to a carriage return, linefeed (0D0Ah).

In addition, a <Ctrl-z> (decimal 26) will be changed to EOF.

This translation provides compatibility with UNIX operating system conventions. Its effects are normally benign, except when raw binary data needs to be read or written, in which case the automatic translation can become an obstacle. Fortunately provisions have been made. If you want to open a file for reading or writing, but do not want the normal translation services, append a b (standing for *binary mode*) to the mode designation as in:

"rb" instead of **"r"**

or

"wb+" instead of **"w+"**

* Many (although not all) text editors append a <Ctrl-z> at the end of a file to indicate that there is no more usable data. In *translated mode*, this byte is detected and passed to the program as an EOF. In *untranslated mode (binary mode)*, this byte will be given to the user as a <Ctrl-z>, and if the file contains any additional data, it will be read.

* As of this writing, the number of files which may be open is automatically set at eight by DOS. This number includes the screen, RUN/C (an open file to be sure) and possibly others. We recommend raising the number of files which can be held open simultaneously by inserting a statement such as:

```
FILES = 20
```

in your CONFIG.SYS file which must reside on the boot-up drive/directory. See the DOS manual for details regarding setting DOS's environmental variable FILES. If an attempt is made to open more files than are allowed by the system, the results will be unpredictable.

RETURNS NULL, if error.
file pointer for specified file, if successful.

EXAMPLE

```
main()
{  /* fopen.c -- copies a file onto the disk */
FILE *fi, *fo;
char c, source[25], dest[25];

     printf("Input filename? ");
     gets(source);
     printf("Output filename? ");
     gets(dest);
     fi = fopen(source, "r");
          if(fi == NULL){
               puts("Couldn't open input file;
               exiting program.");
               exit();
          }
     fo = fopen(dest, "w");
          if(fo == NULL){
               puts("Couldn't open output file;
               exiting program.");
               exit();
          }
     c = 0;
     while((c = getc(fi)) != EOF)
          putc(c, fo);
     fclose(fi);
     fclose(fo);
     puts("Done");
}
```

SEE ALSO open, fclose, feof, ferror, freopen, clrerr, fflush, fseek, ftell

PURPOSE Sets up detailed parameters for repeated execution of a statement or block of statements.

FORMS

```
[1] for( initialize [, more initialize ... ];
         condition; iteration [more iteration...] )
              statement(s) within loop;

[2] for( initialize ...; ; iteration)
              statements(s);

[3] for ( ; ; )
              statement(s);
```

USAGE **for** can be thought of as a counter, governing the number of times a given operation or set of operations is performed. The statement has three distinct parts, separated by semi-colons.

In form [1], above, the first portion, *initialize*, performs any one-time housekeeping desired at the start of the loop, typically execution of an expression that initializes a variable (e.g., **i = 0**); note that the initialize portion may consist of two or more expressions separated by commas (e.g., **i = 0, a = 10**).

The second parameter, *condition*, is an expression that is evaluated for zero (stop) or non-zero (continue) with each cycle through the loop. A **for** statement works like a **while** statement inasmuch as it executes the loop body every time the condition evaluates to non-zero (true).

The third portion, *iteration*, is evaluated after each performance of the loop body statement(s) but before each new test of the *condition*. It is typically a simple incrementation of an index variable, but in fact may be any legitimate expression, or group of two or more expressions separated by commas.

The C **for** statement is imperfectly anologous to its BASIC namesake. K&R point out that any C **for** statement can be thought of as:

```
initialize;
while (condition {
      [ body ] statement(s);
      iteration;
}
```

Let's dissect the following example of a **for** statement:

```
for (a = 20; a > 0; --a)
      printf("a: %d\n", a);
```

1) The initialization portion of **for** is executed and the index variable's value is set to 20 (**a = 20**).

2) The *condition* portion of **for** is evaluated. If the result is true (**a** is non-zero), the statements inside the **for** loop will be executed.

3) If the result is not true (**a** is not greater than 0), the **for** loop has completed its work and program execution will pass to the next series of statements beginning with those immediately following the body of the **for** loop.

4) If the result was true (2, above), after the body of the loop has been executed, the iteration portion of **for** is executed (in our example, **a** is decreased by 1).

5) The test portion is again executed (2, above) and so on through the loop until **a** is equal to or less than 0.

Notice that part 1, the initialization portion of **for**, is executed only once and that part 3, the modification portion, is not executed at all unless the test portion of **for** evaluates to true and the statements enclosed by the **for** have been executed at least once.

Expressions within **for** can be quite complex. For example:

```
for(count = 0; (c = getchar()) != EOF; ++count);{
     statement(s);
          }
```

uses **getchar** to capture characters from the keyboarc until a <Ctrl-z> (indicating end-of-file) is detected.

Meanwhile all the statements controlled by the **for** loop will be executed each time a character is obtained from the keyboard.

It is also possible to increment or decrement several variables at once using the comma operator (,) in the form:

```
for(c=0; (c=getchar()!=EOF); ++count, --mem){
     statement(s);
     }
```

In the above example both the value of the **mem** and **count** variables will be changed each time a character is captured.

The two semi-colons within the **for** statement's parentheses are required although, interestingly, the initialization, test and modification statements are optional. Form [2], above, illustrates this pattern. Since in this case there are no conditions under which **for** will cease to function, the loop will continue forever unless something inside **for**'s block (such as a **break, exit, goto** or **return** statement) passes execution to another portion of the program or terminates it altogether. Of course the program can be interrupted from the keyboard at any time.

A variation on Form [2] occurs whenever the test portion of the statement is missing, as in:

```
for( i = 0 ; ; i++ )
      printf("%d\n", i);
```

NOTES Here are two equivalent C statements, the first expressed using **for**, the second using a **while**:

```
[A]  for(a = initial_value; a > n; a--)
          {
               statement1;
               statement2;
               ..
          }
```

```
[B]   a = initial_value;
      while(a > n){
            statement1;
            statement2;
            ..
            a--;
      }
```

RETURNS Nothing.

EXAMPLE

```
main()
{ /* for.c -- illustrates use of for keyword */
char s[10];

int i;

     puts("tests for statement");
     for (i = 0; i < 10; i++) {
          s[i] = i + 65;
     }
     for (i = 0; i < 10; i++) {
          printf("%c",s[i]);
     }
          /* correct answer is: ABCDEFGHIJ*/
}
```

SEE ALSO while

PURPOSE Sends formatted output to a file.

FORM

```
FILE *fp;

fprintf(fp,"format string", arg1, argn);
```

USAGE **fprintf**, like its cousins **printf**, **lprintf** and **sprintf**, is designed to format data. In the case of **fprintf**, the formatted text will be sent directly to a file. (For a complete discussion of **fprintf** formatting options see **printf**.)

In the above sample:

`FILE *fp;`
declares a file pointer.

`fprintf(fp, "format", arg1, arg2, arg3 .. arg n);`
prints the formatted string to the file indicated by **fp**.

fprintf uses several formatting codes based around the prototype **%a**. The lower case letter which follows the **%** tells the print function how to handle data being passed by variables into the program.

The data type options and their meanings are:

option	data type
%c	character
%d	decimal
%e	exponential floating point
%f	fixed decimal floating point
%g	uses **e** or **f** format, whichever is shorter
%o	octal
%s	string
%u	unsigned decimal
%x	hexadecimal
%%	prints "%"

Like **printf**, **fprintf** allows you to optionally place additional formatting instructions between the **%** and the conversion character. This is the *precision field*. It provides further control over the form of the printed output.

The form can be expressed as:

```
% - # . # l a
```

(In actual usage there are no spaces between the codes in the precision field; they are included here for readability.)

The conversion options and their meanings are:

Option	Meaning
-	left justify output
#	minimum number of spaces in field
.#	for whole numbers, determines the number of characters which can be passed to the field. For floating point numbers, the second # tell how many decimal places to provide for the formatted number.
l	declares that the **int, float** or **unsigned** shall be treated as a **long.**

NOTES Since **fprintf** writes to a file, the referenced file must be opened with **fopen** before the function can be used.

RETURNS Nothing.

(Continued next page)

EXAMPLE

```
main()
{    /* fprintf.c -- demonstrates fprintf()
     function */
FILE *fo;
char c, dest[25];
int i;
char *cp;
double d;

     printf("Output filename? ");
     gets(dest);
     if((fo = fopen(dest,"w")) == NULL) {
          printf("Can't open %s\n",dest);
          exit();
     }
     c = 'A';
     i = 999;
     cp = "This is an arbitrary string.";
     d = 123.456;
     fprintf(fo,"%c %d %s %lf\n", c, i, cp, d);
     fclose(fo);
     puts("Correct answer (in file):A 999 This is
     an arbitrary string. 123.456000");
}
```

SEE ALSO printf, fscanf

PURPOSE Write a character to the specified file.

FORM

```
int ret;
char c;
FILE *fp;

ret = fputc(c, fp);
```

USAGE **fputc** sends a single character to the indicated file.

In order to use **fputc**, the file specified by the file pointer must have been previously opened by an **fopen** statement.

In the above sample:

`int ret;`
declares a return variable.

`char c;`
declares a character variable to hold the character to be sent to the file.

`FILE *fp;`
sets the file pointer.

`ret = fputc(c, fp);`
places value of **c** in the file via the file pointer. The returned code is assigned to **ret**.

If successful, the **char** value will be assigned to **ret**; if unsuccessful, **ret** will be given an EOF value. A disk full situation is a common cause of an error value being returned.

NOTES In RUN/C there is no difference between **putc** and **fputc**. The same is not necessarily true, however, for C compiler environments. See your compiler manual for details on the differences between these functions in that particular implementation of C.

RETURNS EOF, if an error is detected.
Character value, if successful.

EXAMPLE

```
main()      /* fputs.c -- sends a character string
               to a file */
{
int ret;
FILE *fi, *fo;
char c, source[50], dest[50];

     printf("test  filename?  (use  TEST.FIL  or
     other scratch file): ");
     gets(source);
     fi = fopen(source, "w+");
     ret  =  fputs("abcdefghijklmnopqrstuvwxyz",
     fi);
     fclose(fi);
     puts("Done");
}
```

SEE ALSO lputc, fputs, printf

PURPOSE Sends a character string to a file.

FORM

```
int ret;
char *s;
FILE *fp;

ret = fputs(s, fp);
```

USAGE **fputs** outputs a string to a file. Characters from the string are written to the specified file until a zero byte is encountered. The zero byte is not sent. In order to properly function, the file specified by the file pointer must have previously been opened by the **fopen** statement.

In the above sample:

`int ret;`
declares an integer to hold the returned code.

`char *s;`
declares a pointer to the character string.

`FILE *fp;`
sets the file pointer.

`ret = fputs(s, fp);`
sends the string **s** into the file pointed to by **fp.** The returned value from the function will be assigned to **ret.**

If an error condition is detected, an EOF will be returned. Common causes of error conditions include disk full, file improperly opened and the previous error condition not being canceled with **clrerr.**

RETURNS EOF, if an error is detected.

SEE ALSO puts, fgets, printf

(Continued next page)

EXAMPLE

```
main()
/*
     fputs.c -- fputs() sends a character string
     to a file
*/
{
int ret;
FILE *fi, *fo;
char c, source[50], dest[50];

     printf("test filename? ");
     gets(source);
     fi = fopen(source, "w+");
     ret  =  fputs("abcdefghijklmnopqrstuvwxyz",
     fi);
     fclose(fi);
     puts("Done");
}
```

SEE ALSO fgets, fprintf

PURPOSE Returns the number of bytes of memory available.

FORM FRE

USAGE FRE displays the number of bytes of memory available to programs and their data running under the RUN/C interpreter.

NOTES

* The number of available bytes equals the total amount of system memory less the amount occupied by the system (COMMAND.COM and other hidden DOS files), the amount of memory taken by RUN/C and any other co-resident programs or processes such as RAM disks.

* The number of bytes free will be reduced proportionately when a program is resident in memory.

* In RUN/C Professional FRE can be called from the executing program using the **runc()** command as in:

```
runc("FRE");
```

EXAMPLE

```
FRE
64332 bytes

Ok
```

PURPOSE Reads blocks of data from a file.

FORM

```
int ret;
char *buffer;
int blk_size;
int num_blks;
FILE *fp;

ret = fread(buffer, blk_size, num_blks, fp);
```

USAGE **fread** (standing for file read) is a *Level* 2 (buffered) I/O function. The function reads the next group of blocks of contiguous data from a file, placing the data read in a buffer in memory. After the operation is performed, the file's character position pointer is advanced one byte beyond the end of the last block of data read. **fread** can only be used with a file which has been opened with **fopen** or **freopen.**

In the above sample:

`int ret;`
declares a variable to hold the returned code.

`char *buffer;`
declares a character pointer to the beginning of a point in memory where the data read will be held.

`int blk_size;`
declares an integer to hold the size of the blocks of data.

`int num_blks;`
declares an integer to hold the number of blocks of data to be read each time the function is called.

`FILE *fp;`
declares the file pointer.

`ret = fread(buffer, blk_size, num_blks, fp);`
reads **n** blocks of data from a point in the file pointed to by **fp** into a buffer pointed to by the character pointer **buffer.** The size of each block is set by **blk_size.** The actual number of blocks read is returned to **ret.**

The return count is equal to the actual number of blocks read unless an EOF or other error is encountered, in which case a 0 or -1 is returned.

fread and its complementary function **fwrite** are often used to perform I/O on blocks of equal size such as structures. One common calling sequence using **fread** might be:

```
ret = fread(buffer, sizeof(struct1), num_bl, fp);
```

NOTES The returned value from **fread** should be checked before using the data accumulated in the read buffer. If the returned value differs from the number of blocks which were to be processed, a possible error has occurred. The **ferror** function can be used to help determine the cause of the discrepancy.

RETURNS -1, if error is detected.
0, if end-of-file.
number of blocks read, if completely successful.
actual number of blocks of bytes read, if partially successful.

EXAMPLE

```
/* fread.c -- demonstrates fread() */
/*    run fwrite.c first to make test file */

struct employee {
     char name[50];
     long socsec;
     long salary;
};

#define STAFFSIZE 3

struct employee staff[STAFFSIZE];

main()
{
int i, blcount;
FILE *efile;

if ((efile = fopen("staff.tst","rb")) == NULL) {
     puts("Can't open input file.");
     exit();
     }
blcount   =   fread((char   *)   staff,   sizeof(staff[0]),
STAFFSIZE, efile);
fclose(efile);
printf("%d blocks actually read.\n",blcount);
for (i = 0; i < STAFFSIZE; i++) {
     show_emp(i);
     }
}

show_emp(no)
int no;
{
printf("\nEmployee #%d: %s\n",no + 1,staff[no].name);
printf("Social Security number: %ld\n",staff[no].socsec);
printf("Annual Salary: $%ld\n\n",staff[no].salary);
}
```

SEE ALSO fwrite, fopen, ferror

PURPOSE Releases memory which has previously been dynamically allocated.

FORM

```
int ret;
char *cp;

ret = free(cp);
```

USAGE **free** releases (unallocates) a portion of memory which has been previously allocated by **malloc** or **calloc.**

In the above sample:

`int ret;`
declares a variable to hold the returned value.

`char *cp;`
sets the memory block pointer.

`ret = free(cp);`
frees previously allocated memory which is addressable through the **cp** pointer. The returned value from the function will be assigned to **ret.**

A 0 value is returned if the memory has been released correctly; a -1 value will be returned in case of error.

The pointer should have the same value as that returned by the **malloc** or **calloc** function. **free** checks the pointer value for validity; that is, verifies that it is an element of the memory pool. If **cp** does not point to the beginning of a block of memory that was obtained by means of **malloc** or **calloc** an error will result and execution will stop.

NOTES

* Once freed, memory can be reused by the running program.

* Memory allocations are automatically released at the end of program execution, even if the **RUN** is halted by an error or with <Ctrl-c>.

RETURNS -1, if invalid block pointer.
0, if successful.

EXAMPLE See example under **malloc.**

SEE ALSO `malloc, calloc`

PURPOSE Reopen a file for buffered disk operations.

FORM

```
FILE *fpr;
char *filename;
char *mode;
FILE *fp;

fpr = freopen(filename, mode, fp);
```

USAGE **freopen** attaches a new file to a previously used file pointer.

In the above sample:

`FILE *fpr;`
sets the return value pointer.

`char *filename;`
declares a pointer to the file name.

`char *mode;`
declares a pointer to the selected access mode.

`FILE *fp;`
sets the file pointer.

`fpr = freopen(name, mode, fp);`
the specified file is reopened under the control of **fp**. The returned value from the function will be assigned to **fpr**.

In the sample, **fpr** will hold the file pointer number if the operation is successful. If an error is detected, a NULL will be returned. Among the causes of errors are: 1) an invalid mode was specified; 2) the file could not be created (possibly because the disk was full); or 3) too many files were already open.

As there are only a limited number of file pointers available, it is desirable to conserve their use. **freopen** is primarily used by programs which may open and close many files, but which hold open only a few files at a time.

When **freopen** is used, the file previously pointed to by the selected pointer is automatically closed. The name

and mode arguments and their meanings are the same as those available to **fopen**, namely:

Mode	Effect
r	Open an existing file for reading. Input only.
w	Create a new file, or erases, then recreate an existing one, for writing. Output only.
a	Create a new file, or append to an existing one, for writing. Output only.
r+	Open an existing file for update (both reading and writing), starting at the beginning of the file. Input/output.
w+	Create a new file, or erases, then recreate an existing one, for update (both reading and writing). Input/output.
a+	Create a new file, or append to an existing one, for update (both reading and writing). Input/output.

The mode character must be lower case. Note that the write (**w** or **w+**) option discards any existing data in the file. The **a** and **a+** options append to the end of an existing file, or create a new one.

When a file is reopened for updating (**r+**), both reading and writing may be performed on the file pointer. Reopening the file to append (**a+**) forces all data in the buffer to be written to the end of the file, regardless of the current file pointer position. If a **b** is appended to the mode argument, the *binary mode* (untranslated mode) will be used.

For more information on file modes see **fopen**.

(Continued next page)

EXAMPLE

```
main()
{   /*  freopen.c  --  illustrates  use  of
    freopen.c() */
FILE *fp;
char name[20], c;

     printf("Filename: ");
     gets(name);
     clrerr(fp);
     if ((fp = fopen(name,"r")) == NULL)
          exit();
     while((c = getc(fp)) != EOF)
          putchar(c);
     fclose(fp);

     printf("Filename: ");  /* here we go again
                               */
     gets(name);
     clrerr(fp);
   if ((freopen(name,"r", fp)) == NULL)
          /* here we reuse file pointer */
      exit();
     while((c = getc(fp)) != EOF)
          putchar(c);
     fclose(fp);

}
```

SEE ALSO **fopen**

PURPOSE Reads formatted input from a file.

FORM

```
int ret, i, j;
FILE *fp;

ret = fscanf(fp, "%d %d", &i, &j);
```

USAGE **fscanf**, like its cousins **scanf** and **sscanf**, scans input and reads formatted data. In the case of **fscanf** the data is obtained from a file in a form specified by the conversion string. The data is then placed in the variables that are pointed to by the pointer arguments.

In the above sample:

`int ret, i, j;`
declares an integer **ret** to hold the number of input items successfully matched, or -1. Also sets up two other integer variables: **i** and **j**.

`FILE *fp;`
sets the file pointer.

`ret = fscanf(fp, "%d %d", &i, &j);`
scans the specified file for input from the current position of the file pointer and assigns the first two items encountered to the addresses pointed to by **i** and **j**. The returned value from the function will be assigned to **ret**.

If the operation is successful, the number of data items successfully scanned will be assigned to **ret**. If none are passed successfully, a -1 value will be returned, indicating an error condition or improper formatting of the **fscanf** string.

fscanf provides for the extraction of information from a file in sequential order using the appropriate conversion characters of the format **%a** where the **%** alerts the computer that a special character is coming. The lowercase alphabetic character which follows tells **fscanf** what to do with the data being passed to variables in the program.

The data type options and their meanings are:

option	data type
%c	single character
%d	signed decimal
%f, %e, %E	floating point
%o	octal
%s	string
%u	unsigned decimal
%x	hexadecimal
%%	a percent sign will be expected

For a more complete explanation of the use and effects of these variable types, see **scanf.**

Like **scanf, fscanf** optionally allows you to place additional formatting instructions between the **%** and the conversion character. This is referred to as the *precision field.* It provides further control over the form of the printed output. This form can be expressed as:

% * # l a

(In actual usage there should be no white space in the precision field; it is included here for the convenience of the reader.)

The conversion options are:

- ***** indicates that the conversion is to be performed, but the resulting value is not to be returned.
- **#** specifies the maximum field width for the input data item.
- **l** (el) indicates that the data item is a **long** (long integer) or **double.**

Input is scanned through white space to locate the next input item in all cases except the **%c** char specifier, where the next input character is returned regardless.

Data matching
An alternative to insuring that scanned material is properly ignored is to provide an exact match for the material in the **scanf** formatted command. For example:

```
int a, b;
fscanf("Cows: %d Cowboys: %d", &a,&b)
```

should correctly extract the numbers 25 and 50 from the following input file data:

```
Cows: 25 Cowboys: 50
```

NOTES

All input targets for **fscanf** must be *pointers.* However, if the assignment suppression feature (*) is used, a pointer must not be supplied for that specifier since pointers are matched to data on a one-to-one basis. Since the asterisk suppresses returned data, **fscanf** would lose track of data being captured and formatted if a pointer was available for data which was never sent.

RETURNS

-1, if an error is detected.
number of input items for which data was found, if successful. (This includes all single character items (%c) in the control string.)

EXAMPLE

```
/*
demonstrates fscanf() (adapted from K&R p149)

Note that file fscanf.tst contains (only):
     25     54.32E-1   Thompson
     56789 0123 45a72 */

main()
{    /* fscanf.c -- shows use of fscanf() */
FILE *input;
int i;
float x;
char name[50];
if((input = fopen("fscanf.tst","r")) == NULL) {
     puts("Can't open fscanf.tst");
     exit();
}
fscanf(input,"%d %f %s", &i, &x, name);
printf("%18d %f %s ", i, x, name);
fscanf(input,"%2d %f %*d %2s", &i, &x, name);
printf("%d %f %s\n", i, x, name);
puts("Correct answer: 25 5.432000 Thompson 56
789.000000 45");
}
```

SEE ALSO

scanf, fprintf

PURPOSE Seeks to a new position in a file.

FORM

```
int ret;
FILE *fp;
long pos;
int mode;

ret = fseek(fp, pos, mode);
```

USAGE **fseek** shifts the current character position pointer of a file. The number of characters of movement (known as the *offset position*) is specified as a **long** and is changed relative to the beginning of the file (mode 0), to the current file position (mode 1), or to the end of the file (mode 2).

There is nothing particularly mysterious about the file position pointer; it is part of the built-in **FILE** structure and is actually just a number or counter which is modified by **fseek**, among other *Level* 2 I/O functions. The only way to access the file position pointer in *Level* 2 I/O is with **fseek**.

In the above sample:

`int ret;`
declares a variable to hold the returned value.

`FILE *fp;`
declares the file pointer.

`long pos;`
declares a long variable to hold the number of characters to be moved within the file.

`int mode;`
declares a variable to hold the selected mode.

`ret = fseek(fp, pos, mode);`
moves to the specified position in the file pointed to by **fp**, depending upon which mode is chosen. The value returned by the function is assigned to **ret**.

ret will hold a value of 0 if the move was successful; -1, if an error occurred.

NOTES

* pos in the above sample may be a positive or negative number, indicating forward or backward motion within the file.

* The primary difference between **lseek** and **fseek** is that **lseek** uses *Level* 1 (unbuffered) I/O with files opened with the disk primitive functions **open** or **creat**, while **fseek** is based on disk access via file pointers whose values are assigned via **fopen. fseek** is used in operations involving buffered file I/O.

RETURNS -1, if an error is detected.
0, if successful.

EXAMPLE

```
main()     /* fseek.c */
{    long pos;
     int ret;
     FILE *fi, *fo;
     int mode = 0;
     char c, source[25];

     printf("test filename? ");
     gets(source);
     fi = fopen(source, "w+");
     ret  =  fputs("abcdefghijklmnopqrstuvwxyz",
     fi);
     pos = 2;
     fseek(fi,pos,mode);
     pos = ftell(fi);
     printf("file position is %ld\n", pos);
     fclose(fi);
     puts("Done");
}
```

SEE ALSO fopen, lseek, FILE

PURPOSE Returns the position of the file pointer.

FORM
```
long pos;
FILE *fp;
pos = ftell(fp);
```

USAGE **ftell** reports the current file position which is the number of bytes from the beginning of the file to the current setting of the file position pointer.

In the above sample:

`long pos;`
declares a long variable to hold the current file position.

`FILE *fp;`
sets the file pointer.

`pos = ftell(fp);`
determines the current position of the file pointer relative to the beginning of the file. The returned value from the function will be assigned to the **pos.**

NOTES The **ftell** function only operates on files opened under *Level* 2 (buffered) file functions -- **fopen** and **freopen.**

RETURNS -1, if an error is detected.
Current file position, if successful.

(Continued next page)

EXAMPLE

```
main()    /* ftell.c -- illustrates use of
             ftell() */
{
     long pos;
     int ret;
     FILE *fi, *fo;
     char c, source[25];

     printf("test filename? ");
     gets(source);
     fi = fopen(source, "w+");
     ret = fputs("abcdefghijklmnopqrstuvwxyz",
     fi);
     pos = ftell(fi);
     printf("file position is %ld\n", pos);
     fclose(fi);
     puts("Done");
}
```

SEE ALSO fopen, fseek, FILE

PURPOSE Writes blocks of data to a file.

FORM

```
int ret;
char *buffer;
int blk_size;
int num_blks;
FILE *fp;

ret = fwrite(buffer, blk_size, num_blks, fp);
```

USAGE **fwrite** (standing for file write) is a *Level* 2 (buffered) I/O function which writes the next group of blocks of contiguous data to a file, reading from the beginning of a buffer in memory. After the operation is performed, the file's character position pointer is advanced just beyond the end of the last block of data which was written to it. **fwrite** can be used only with a file which has been opened with **fopen** or **freopen.**

In the above sample:

`int ret;`
declares a variable to hold the returned code.

`char *buffer;`
declares a character pointer to the beginning of a point in memory where the data which is to be written will be found.

`int blk_size;`
declares an integer to hold the size of the blocks of data.

`int num_blks;`
declares an integer to hold the number of blocks of data to be read each time the function is called.

`FILE *fp;`
sets the file pointer.

`ret = fwrite(buffer, blk_size, num_blocks, fp);`
writes n blocks of data to a point in the file pointed to by **fp** from a buffer pointed to by the character pointer **buffer.** The size of each block is set by **block_size.** The actual number of blocks read is returned to **ret.**

The return count is equal to the actual number of blocks written unless an EOF or other error is encountered, in which case a 0 or -1 is returned.

fwrite and its complementary function **fread** are often used to perform I/O on blocks of equal size such as structures. One common calling sequence using **fwrite** might be:

```
ret = fwrite(buffer,sizeof(struct1),num_bl,fp);
```

NOTES

The returned value from **fwrite** should be checked immediately after the function is used. If the returned value differs from the number of blocks which were to be processed, a possible error has occurred. The **ferror** function can be used to help determine the cause of the discrepancy.

RETURNS

-1, if error is detected.
0, if end-of-file.
number of blocks written, if completely successful.
actual number of blocks written, if partially successful.

(Continued next page)

EXAMPLE

```
/* fwrite.c -- demonstrates fwrite() */
/*     run to make test file for fread.c */

struct employee {
     char name[50];
     long socsec;
     long salary;     };

#define STAFFSIZE 3

struct employee staff[STAFFSIZE];

main()
{
     int i, blcount;
     FILE *efile;

     puts("Please input information about three
     fictional employees");
     puts("The information will be stored in
     STAFF.TST\n");
     for (i = 0; i < STAFFSIZE; i++)
          get_emp(i);
     if ((efile = fopen("staff.tst","wb")) ==
     NULL) {
          puts("Can't open output file.");
          exit();
     }
     blcount    =    fwrite((char    *)    staff,
     sizeof(staff[0]), STAFFSIZE, efile);
     fclose(efile);
     printf("%d          blocks          actually
     written.\n",blcount);
}

get_emp(no)
int no;
{
     printf("Employee #%d: Name? ",no + 1);
     gets(staff[no].name);
     printf("Social Security number? ");
     scanf("%ld",&(staff[no].socsec));
     printf("Annual Salary? ");
     scanf("%ld",&(staff[no].salary));
}
```

SEE ALSO fread, fopen, ferror

PURPOSE Obtains a character from a file.

FORM

```
int c;
FILE *fp;

c = getc(fp);
```

USAGE **getc** captures characters from a file, one at a time.

In the above sample:

`int c;`
: declares a variable to hold the next input character or EOF.

`FILE *fp;`
: sets the file pointer.

`c = getc(fp);`
: gets the next character from the file, assigning it to **c.**

NOTES

* **getc** -- as well as **fgetc** and **getchar** -- return integers which can be assigned to character variables. They return integers rather than **chars** since it must be possible to register a -1 on sensing end-of-file, a value which **char** cannot hold since it is limited to positive values 0 through 255.

* **getc** can only be used on files which have been opened for *Level* 2 I/O with **fopen.**

* This function differs from **getchar** in that **getc** can take a file pointer as an argument.

* In the case of RUN/C, there is no difference between **getc** and **fgetc.** See your compiler manual for any differences in that particular implementation of C.

RETURNS -1, if an error or end-of-file is detected.
Integer value of the character retrieved, if successful.

EXAMPLE

```
main()      /* getc.c -- simple file copier with
               console echo */
{
     FILE *fi, *fo;
     int c;
     char source[25], dest[25];

     printf("Input filename: ");
     gets(source);
     printf("Output filename: ");
     gets(dest);
     if((fi = fopen(source, "r")) == NULL) {
          printf("\tCan't open %s\n",source);
          exit();
     }
     if((fo = fopen(dest, "w")) == NULL) {
          printf("\tCan't open %s\n",dest);
          exit();
     }
     while((c = getc(fi)) != EOF) {
          putchar(c);
          putc(c, fo);
     }
     fclose(fi);
     fclose(fo);
     puts("\n{DONE}");
}
```

SEE ALSO fopen, fgetc, getchar

PURPOSE Obtains a character from the console.

FORM
```
int c;
c = getch();
```

USAGE **getch** captures a character from the user's keyboard.

In the above sample:

`int c;`
declares a variable to hold the returned character.

`c = getch();`
awaits the next character to be typed at the console. When it is typed, the character is assigned to **c**.

NOTES

* Notice that **c** is defined as an **int**. This is because a <Ctrl-z> (EOF) character will generate -1, a value which cannot be contained within a variable of type **char**.

* The primary difference between **getch** and **getchar** is that the former will send characters to the console screen even if other input has been redirected to another output device such as the printer or a communication port.

RETURNS Character typed at the console.

EXAMPLE
```
main() /* getch.c -- demos getch() and putch() */
{
     char c;
     int i;
     printf("Type a character and I'll print it
     back after a delay: ");
     c = getch();
     puts("");
     for (i = 0; i < 100; i++) ;
     putch(c);
     puts("\nCorrect output is your character
     ... after a brief delay.");
}
```

SEE ALSO getchar, putch

PURPOSE Obtains a character from the keyboard.

FORM

```
int c;

c = getchar();
```

USAGE **getchar** "captures" keystrokes from a computer keyboard or other standard input (**stdin**) device. (For more information on **stdin**, see Section 1.)

In the above sample:

`int c;`
declares an integer variable.

`c = getchar();`
gets the next character from the keyboard, assigning it to **c**.

NOTES

* **getchar** as well as **fgetc** and **getc** return integers which can be assigned to character variables. Integers values are returned rather than **char** values because it must be possible to generate a -1 on sensing end-of-file, a value which **char**, being limited to values 0-255, cannot hold.

* You may find some variation in the way compilers handle **getchar** and **putchar**. RUN/C processes characters immediately; some compilers wait until a newline is sensed.

RETURNS EOF, if an error is detected.
Integer value of the character retrieved, if successful.

(Continued next page)

EXAMPLE

```
main() /* getchar.c -- demonstrates getchar() */

{

char  c;
puts("Type any character\n");
c = getchar();
if (c == 1)
     puts("\nSmile, you're on candid computer,
     you typed a control-A !\n");
else
     puts("\nYou typed something other than a
     control-A");
}
```

SEE ALSO getc, putchar

PURPOSE Obtains a string of characters from the keyboard.

FORM
```
char *p;
char *s;

p = gets(s);
```

USAGE **gets** captures an input string from the keyboard or other standard input device. When **gets** is invoked a stream of characters is read until a *newline* (<enter> when input is being taken from the keyboard) is encountered. Then, **gets** replaces the newline character(s) with a zero byte, signifying the end of the string.

In the above sample:

`char *p;`
declares a pointer to a string.

`char *s;`
declares a pointer to the input string.

`p = gets(s);`
the newline-delimited string is placed in an area of memory referenced by the pointer **p**.

If the value returned by **gets** is NULL, an error has been detected. Otherwise a pointer to the captured string will be returned.

NOTES
* Since **gets** is not initialized to a particular length the input buffer must be large enough to accommodate the string. If a fixed array such as **p[50]** is used, any attempt to allocate a string larger than 49 characters to **p** will overwrite p's buffer and potentially "bomb" your program or other portion(s) of RUN/C program memory. **When a pointer is used, it must point to a data element with sufficient allocated memory to hold the expected input.**

* With RUN/C Ver. 2, the interpreter now valiently tries to protect the user from the types of novice mistakes using **gets** and **fgets** that all of us make now and then, namely getting a string and attempting to put it into an unititialized pointer. This protection is accomplished in two steps: (1) all

newly declared pointers are set to NULL unless otherwise initialized; (2) attempts to use built-in functions such as **gets** as in:

```
main()
{
char *p;
gets(p);
printf("%s", p);
}
```

will result in the message:

```
Error #238 at line 48:
Function call illegal with NULL-
valued pointer
```

The "trust-me" flag (activated with **SET TRUST ON**) is available to overcome this safety feature, if desired.

RETURNS NULL, if an error is detected.
Input character string pointer, if successful.

EXAMPLE

```
main()      /* gets.c -- demonstrates gets() */
{
char i[50];
printf("Type a string of characters.\n\n");
gets(i);
printf("Here's the string typed:\t%s\n\n",i);
}
```

SEE ALSO puts, scanf, SET TRUST

PURPOSE Goes to a labeled point elsewhere within the same function.

FORM

```
if (disaster)
     goto label;
  ..
  ..
  ..

label:
```

USAGE **goto** causes program control to branch unconditionally to a *label* located elsewhere within the currently executing function. If the named label is not found, program execution will be terminated. The label must be followed by a colon as in the example below.

NOTES **goto**, both the basis and bane of BASIC, is part of the C language, but is rarely used both for technical and political (and, some would say, religious) reasons.

goto statements are never needed in properly designed C programs because of the structured nature of the language. Instead of using absolute **goto**'s, program control is passed (or not passed) to another part of the program based on the use of **continue, return, break** and **exit** keywords.

In most cases using a **goto** is the equivalent of crying "uncle" or, as Kelley and Pohl wrote in A Book on C, "the **goto** is a primitive method of altering flow of control, which in a richly structured language is unnecessary."

But look. K&R suggested that **goto** might indeed be appropriate in a situation where a value -- such as an error test -- is evaluated in a deeply nested program segment and which, if true, causes the program to end or switch to some defensive measures.

As far as our two bits' worth goes, we believe that for the rational management of loops, **break** or **continue** is almost always more appropriate than **goto**. But **goto** may sometimes be justified because **break** will get you out of the innermost loop of your function only, while **goto** will take you nearly wherever you want to go within your function.

LIMITS RUN/C imposes some necessary and hopefully rational rules* on the use of **goto**. These rules are:

1. When a **goto** statement is reached and your program is within a **do ... while** or **for** loop, or a **switch** block, all operations within that block will close down as if a **break** statement had been executed.

2. If you are within some nested combination of loops and/or switch blocks, the effect will be that of enough breaks to get you out of all the loops and back to the "outer" level of the current function.

3. If the specified label is not found within the function or if the label is in an illegal position (see below), an error message will be displayed and program execution ended.

4. The label may not be placed in the following types of positions:

   a. The *then* or *else* arm of an **if** or **if ... else** statement.
   b. Inside the body of a **do, for** or **while** loop or a **switch** block.
   c. Inside any compound statement.
   d. Outside all functions.

5. If a label is to be used, it must be the first non-white space entity on its logical line.

6. The label must be followed by a colon with no intervening white space.

We hope that these restrictions will not hinder the use of a **goto** as it was intended to be used. Indeed, they should be helpful in properly employing this C orphan.

RETURNS Nothing.

* Most rules described were adapted from the C Reference Manual, Harbison/Steele, 1984.

EXAMPLE

```
main()      /* goto.c -- tests goto statement */

{
int a, b, c, d;
puts("Tests three if statements");
a = 3;
b = 4;
c = 4;
d = 6;

if(c == 4)
if(a == 3)
if(b == 4)
     goto label;
finish:
     printf("done");
     exit();
label:
     printf("it worked\n");
     goto finish;
}
```

SEE ALSO break, continue

PURPOSE Displays system-dependent information.

FORM HELP

USAGE HELP displays a help screen appropriate to your current activity. At the command level (the **Ok** prompt) HELP provides information about system limitations: Values in parenthesis are accurate as of this writing, but will change from version to version. The on-line HELP facility will provide the current statistics.

```
Use CLIST to see interpreter commands, FLIST to see built-in functions.
Use EDIT (or E) to enter or change your program. Variations:
        EDIT nn -- Enter editor with cursor on line number nn
        EDIT END -- Enter editor with cursor at end of program
        EDIT -- Enter editor at beginning of program or last position
        EDIT ERROR -- Enter editor with cursor where last error was found
Use Esc (Escape) to return to command level when done editing.

Predefined symbols: FILE, EOF, NULL, stdin, stdout, stderr
**** Quantitative MAXIMUM specifications for this version and session:
Significant characters in an identifier: 20
Program lines (this session): 2000
Blocks (framed by { and }) within a program: 500
Global and static variables (and arrays): 250
Automatic variables (and arrays) active at one time: 200
Functions defined within your program: 100
Labels ('goto' targets) in your program: 10
Arguments you may supply to a function (built-in or your own): 8
Memory allocations active at one time: 50
Preprocessor symbols #defined within a program (this session): 25
Aggregate definition space: 3000 bytes
```

NOTES

* **?** is a synonym for **HELP**.
* Specific HELP is also available at the when <Ctrl-c> is pressed.

SEE ALSO FLIST, CLIST

PURPOSE Transfers control from the executing program to RUN/C Professional's Interactive mode after program execution has been interrupted.

FORM

```
Break in line n
 Continue, Dump, Ldump, Interactive,
  Help or End (C,D,L,I,H,E)?

     I
```

USAGE Interactive mode permits independent 1) execution of most RUN/C commands, 2) execution of user program functions, 3) determination of the current values of existing variables, and 4) changing of program variables prior to resumption of program execution.

The Interactive mode is accessed only after a program has been interrupted either by pressing <Ctrl-c> while your program is executing or by using the **runc()** function in conjunction with the **BREAK** command. (For more information on the use of <Ctrl-c> see Section 1. For more information on RUN/C Professional debugging features see Section 6.)

Upon interruption the following menu will appear:

```
Break in line n
  Continue, Dump, Ldump, Interactive, Help or End (C,D,L,I,H,E)?
```

If **I** is pressed, the program will enter the Interactive mode and the special:

```
(Use HELP or ? for help with interactive mode.)
>>
```

prompt will appear. At this point you may 1) ask for help; 2) return to the mini-menu by pressing <enter> alone or typing **SYSTEM** at the Interactive mode prompt (>>); 3) execute a command or function available to the interactive mode; 4) determine or change a variable's value.

If you ask for help, a message similar to:

```
You can use most, but not all, interpreter commands to examine parts of
your program in its interrupted state. Commands of most likely interest:

      LIST [line range] -- displays your program's source code
      VAL variable name -- displays type and value of specified variable
      @ function-name(arguments); -- execute one of your program's functions
      @ variable name = expression; -- change a variable's value
      SET STEP ON/OFF -- turn single-step mode on or off
      TRON/TROFF -- turn source-line trace on or off

You cannot edit or erase your program in this mode.

Use SYSTEM or press the Enter key (alone) to leave this mode.
```

will appear.

(Continued next page)

EXAMPLE

```
main() /* demonstrates Continue option */
        {
                int a;

                for(a = 0; a < 40; ++a)
                printf("%d \n",a);
        }

RUN

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
```

At this point the display of these interesting numbers is interrupted by pressing <Ctrl-c>. The following choices appear:

```
Break in line 7
  Continue, Dump, Ldump, Interactive, Help or End (C,D,L,I,H,E)? c
```

If **i** is selected, the Interactive mode prompt will appear:

```
(Use HELP or ? for help with interactive mode.)
>>
```

One possibility is to reset the value of the **a** variable with the command:

```
>>@ a=0;
```

If this is done program execution resumes with:

```
18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 ...
```

SEE ALSO D(Dump), E(End), H(Help), I(Interactive), L(Ldump), Section 6

PURPOSE Executes a program statement (or series of statements) based upon an expression being true (non-zero).

FORM
```
if(expression)
     statement(s);
```

USAGE **if**, in its simplest form, tests a condition or expression contained within parenthesis. If the result of the test is true, the specified action, if any, is taken. If the result of the test is not true, no action is taken and program execution passes to the next statement. The action can, of course, be another **if** statement.

EXAMPLES [1]
```
main()
{
     int a;
     a = 3;
     if(a == 3) printf("3 it is!\n");
}
```

[2]
```
main()
{
  /* if.c -- tests three if statements */

int a, b, c, d;

a = 6;
b = 4;
c = 5;
d = 6;

if(c == 5)
if(a == 6)
if(b == 4)
     printf("it worked\n");
}
```

(In example 2 if **a**, **b** or **c** is set to anything other than its expected value -- thus making the statement false -- the phrase within **printf** will not be printed. In other words, all expressions in the series of **if**'s must be true in order for the statement(s) controlled by the **if**'s to be executed.

SEE ALSO `else, else if`

PURPOSE Polls the keyboard for a key hit, trapping both the next hit and its code.

FORM

```
int result;
result = inkey();
```

USAGE **inkey** tests for a key hit on the keyboard. If a hit is detected, the character's ASCII value is returned. If no key has been hit, a -2 value will be returned. Nothing will be echoed to the screen as a direct result of a keyhit immediately after **inkey** is called. The pattern of returned values can be expressed as:

Returned	Meaning
-2	no key has been pressed.
-1	<Ctrl-z> (EOF) has been hit.
>=0	a key which generates an ASCII character has been hit.
<-14	Some special key (such as a function key) has been hit.

The special key codes for the IBM PC are:

Inkey value	Meaning
-1	Ctrl-z
-15	backtab
-16 to -25	ALT Q,W,E,R,T,Y,U,I,O,P
-30 to -38	ALT A,S,D,F,G,H,J,K,L
-44 to -50	ALT Z,X,C,V,B,N,M
-59 to -68	F1 to F10
-71	Home
-72	up-arrow
-73	PgUp
-75	back-arrow
-77	forward-arrow
-79	End
-80	down-arrow
-81	PgDn

(Continued next page)

Inkey value	Meaning
-82	Ins
-83	Del
-84 to -93	F11 to F20 (Upper Case F1 to F10)
-94 to -103	F21 to F30 (Ctrl F1 to F10)
-104 to -113	F31 to F40 (ALT F1 to F10)
-114	Ctrl PrtSc
-115	Ctrl back-arrow
-116	Ctrl forward-arrow
-117	Ctrl End
-118	Ctrl PgDn
-119	Ctrl Home
-120 to -131	ALT 1,2,3,4,5,6,7,8,9,0,-,=
-132	Ctrl PgUp

NOTES

* Negative numbers are used for non-ASCII key hits (exception: <Ctrl-z> generates a -1 (EOF) value) so each unique key on the PC keyboard can have its own **char** value (+/- 255). This differs significantly from the PC's keyboard routine which returns a leading 0 for non-ASCII key hits.

* The perhaps obvious value of this function is that it enables the programmer to more completely control the user's environment. When a key is hit, the program recognizes that fact through the offices of **inkey** and the programmer can then have the program puts out whatever character or message, if any, is appropriate.

* Since negative values can be returned by **inkey**, you should not put the result immediately into a variable of type **char** until you know that the key was an ASCII character with a positive value between 0 and 255.

RETURNS Key code of key that was hit, if any.
-2, if no key has been hit.

(Continued next page)

EXAMPLE

```
main()      /* inkey.c -- illustrates use of inkey() */

{

puts("Title: The prompt of the anxious programmer.");

int c;

while((c = inkey()) == -2)
     puts("No character yet, it's time to worry!");

printf("The letter hit was: %c; it's code name is %d.", c,
c);
}
```

SEE ALSO fabs, getchar

PURPOSE Obtain a value from a port.

FORM

```
unsigned p;
int v;

v = inp(p);
```

USAGE **inp** captures a numeric value from the specified I/O port. The port must be in the range 0 to 65535. The value returned will be in the range 0 to 255.

In the above sample:

`unsigned p;`
declares a variable to hold the selected port number.

`int v;`
declares a variable to hold the returned value.

`v = inp(p);`
calls the specified port, assigning the returned value to **v**.

inp is the complement of **outp**.

NOTES Port assignments may vary from computer to computer, even in situations which claim complete compatibility. Refer to the technical manual of your computer for information on valid port numbers.

RETURNS Value at port.

EXAMPLE

```
main()      /* inp.c -- tests inp() */
{
int ret;

     ret = inp(0x43);
     printf("value returned is %u", ret);
}
```

SEE ALSO outp, peek

PURPOSE Creates a signed variable 16 bits in length (-32768 to 32767).

FORM

```
int a, b, c;
a = 2394;
b = 9273;
c = -699;
```

USAGE **int** is an abbreviation for integer (although the word integer cannot be used in place of **int**). A variable declared **int** is allocated 16 bits of data storage and can take on any whole number value from -32768 to 32767.

NOTES In C, simple data type names (such as **int**) have four distinct uses, namely:

(1) in declaring variables and aggregate numbers;

(2) in casting one type of value to another such as:

```
c = (int)'c';
```

(3) as an argument to **sizeof**, as in:

```
int i = sizeof(int);
```

(4) as an element in a **typedef** statement used in creating an alias for an existing data type such as:

```
typedef int REAL;
```

SEE ALSO char, double, float, long, short, unsigned

PURPOSE Generates a DOS function call.

FORM

```
union REGS r, *inreg, *outreg;
int flags;

r.h.ah = 5;
r.h.dl = 'X';
inreg = &r;
outreg = &r;
ret = intdos(inreg, outreg);
```

USAGE **intdos** permits execution of operating system functions from within a C program. Once the function call has been completed, program control is returned to the executing program.

In the above sample:

`union REGS r, *inreg, *outreg;`
declares one instance of a REGS union, and two pointers to such unions.*

`int flags;`
declares a variable to hold the DOS returned flags.

`r.h.ah = 5;`
places the number 5 in the **AH** register.

`r.h.dl = 'X';`
places the character **X** in the **DL** register.

`inreg = &r;`
sets **inreg** to point to the beginning point of **r.**

`outreg = &r;`
sets **outreg** to point to the beginning point of **r.**

`flags = intdos(inreg,outreg);`
performs a DOS interrupt, assigning any

* **union REGS is defined in the header file intrpt.h, available on the RUN/C distribution disk. See NOTES for a complete discussion.**

returned value to the **flags** integer variable. The particular interrupt and its effect is determined solely by the values in the relevant 8086/8088 machine registers.

Although a CPU value is returned by the **intdos** function itself, data processed via DOS interrupts is returned through the registers. Hence the concept of **inreg**, which assigns values to relevant registers and **outreg**, which accepts particular register values as returned by the interrupt.

In our sample, which sends the character **X** to the printer port using DOS interrupt 5, only the **inreg** register values are of interest. However the **outreg** position in the function must be supplied with the address of a **union**, since **intdos** requires that two parameters be passed to it, even if the addresses are identical.

The carry flag

The value returned by the **intdos** function itself are called CPU flags (also known as *processor status flags*). Here, in binary, are the meanings of these flags from the least bit (0) to most significant bit (15):

Bit	Meaning
0	Carry (if set, usually indicates failure)
2	Parity
4	Auxiliary carry (BCD operations)
6	Zero
7	Sign
8	Trap (for single step operations)
9	Interrupt enable
10	Direction (string operations)
11	Overflow

All other bits are unused.

For the programmer working in C it is bit 0 which is usually of the most interest. The *carry flag*, as it is called, denotes a failure of the operation if it is set to 1, 0 if successful.

Determining whether the carry flag has been set requires some fancy footwork. Generally, the value returned from **intdos** is place in an integer variable. In order to determine if bit 0 is set to 0 or 1 it is

necessary to *mask* the other bits in the 16-bit integer using the *bit-wise and* (&) operation.

For example, to test the success of the program fragment above, the program might be extended in the following manner:

```
flags = intdos(inreg,outreg);

if( flags & 1 {
      puts("Carry bit is 1");
      puts("...an error has occurred.");
      exit();
      }
```

Thus if the carry flag is equal to non-zero or has been *set* (i.e., an error was detected), the error message will be displayed and the **exit** function executed.

Some DOS and BIOS functions provide for additional information on errors by placing specific values in specified registers after the error is detected. Usually it is the **AL** or **AH** register which is passed information of this type. (This is one good reason to check for a carry flag error when doing interrupt processing; otherwise you might consider the returned register value data when, in fact, it is only an attempt to alert you to an error condition of some sort.)

Backgrounder:
Inside intrpt.h

The **union REG r** in the above sample draws upon a predefined term contained in the RUN/C header file intrpt.h mentioned above. Like other header files, intrpt.h must be **#include**'d when the program is run in order to be useful. Alternatively, the contents of intrpt.h can be placed directly within your program as we do in the example below.

Here is intrpt.h in its entirety:

(Continued next page)

```
/* intrpt.h */

struct XREG { short ax,bx,cx,dx,si,di; };
struct HREG {
      char al,ah,bl,bh,cl,ch,dl,dh;
      };
struct SREGS { short es,cs,ss,ds; };

union REGS {
            struct XREG x;
            struct HREG h;
      };
```

struct XREG sets up a structure of **shorts** containing two bytes each for the extended (high + low) system registers. **struct HREG** contains a byte each for high and low order registers considered separately. (For example, the extended register **AX** combines the contents of registers **AH** and **AL**). And **struct SREGS** contains the names of the four memory segment registers.

union REGS defines a union containing the **XREG** and **HREG** structures. This allows us to manipulate extended and/or single byte registers at will.

Fortunately, it is not necessary to master the dynamics of intrpt.h in order to use RUN/C's various interrupt facilities. Simply **#include** it at the beginning of your program and assign values to members as shown in the sample above and the example below. Important: The order of the register names in intrpt.h must not be changed since the values in the various data items are placed in registers in the order in which they appear in **REGS.**

Backgrounder:
... and now for something completely different
When we get around to writing Secrets of the DOS Masters, system interrupts will be one of the first topics. Once understood, system interrupts are not difficult to use; certainly of the topics related to assembly language programming (which this is), interrupts are the easiest to grasp.

The term interrupt refers to a machine-language special process unlike normal machine-language execution. While this sounds mysterious, it is in fact the only way

to make use of the basic operating system services that PC-DOS/MS-DOS provides.

Since the operating system controls all transfers of information into and out of the machine's processor, all C programs ultimately make use of the operating system for I/O even though the process is hidden from the casual programmer by the compiler or interpreter. In fact, the final (or most basic) step of any input/output is a request to DOS to actually do the transfer of information (e.g. **printf** calls DOS through this interrupt mechanism when it is putting characters on the screen).

It is also the job of DOS to take care of the file system on disk -- to understand filenames and directories, and to figure out exactly where (physically) the data that comprises a file is distributed on a disk.

A word on documentation
To use interrupts you must have a DOS manual that describes operating system calls; to make use of the BIOS, you must have a BIOS call listing or BIOS call interface documentation from the manufacturer of your machine. Without this information, you cannot proceed. If you have this documentation, it should be studied with care, since making mistakes on system call requests can be, to say the least, memorable.

The operating system is split into two parts: the BDOS (Basic Disk Operating System) and the BIOS (Basic Input/Output System). The interrupt functions **intdos** and **intdosx** make requests of the BDOS, and **int86** and **int86x** are primarily used to make requests of the BIOS. Or, looked at from another angle, MS-DOS/PC DOS manage the BDOS while the computer hardware is managed by the BIOS.

The BDOS, or DOS as it is more familiarly known, is very much easier to work with because it understands conceptual objects like filenames, whereas the BIOS understands only very low level requests like "move disk head to the following track". For the time being, let's concentrate on the BDOS and our gateway into it: the functions **intdos** and **intdosx.**

There are three steps to a successful system interrupt.

1) The machine's memory registers are partially or

completely filled with sufficient information to communicate to the operating system the details of your request.

2) An "interrupt" is performed, at which time the operating system takes over, examines the registers to figure out what action you want taken and performs the request.

3) After the operating system finishes, control is returned to the program. Some interrupt functions return relevant information to specific operating system or machine registers. In such situations, your program should test for the contents of a register after the interrupt has been performed. For example, information like "did the user hit a key?") or error information (e.g. "the file you referred to doesn't exist") may be available. (See the documentation on the specific operating system or machine function to determine if relevant information is returned through registers.)

While assembly language routines can directly access the machine's registers, C language routines cannot. And that, finally, is the reason for the existence of these function calls -- they simply allow you to set up the registers prior to an interrupt, make the interrupt and return the computer to the control of the locally executing C program.

Backgrounder:
Using interrupts

So how do we get values into the machine's registers in preparation for an interrupt? First we put the values into specific variables in the structure defined in the header file intrpt.h. (A copy of intrpt.h is on your distribution disk.) Then when **intdos** is called, it looks up these values, *pops* them into the machine registers, calls the interrupt, and then puts the register values after the interrupt is completed into another structure which is returned to you.

Whenever using the interrupt functions, you should:

```
#include "intrpt.h"
```

to have access to the structures needed to execute any type of interrupt.

Among the DOS system interrupts provided by standard MS-DOS 2.0 and greater are functions to:

* Display output
* Monitor the asynchronous communication adapter
* Print output
* Control console (keyboard) I/O
* Control disk I/O
* Create, delete and rename files
* Analyze (parse) a filename
* Get and set date and time
* Create a 'child' process
* Create, remove and change sub-directories
* Control attached devices
* Allocate and free memory
* Load or execute a program

NOTES

* Although using simple interrupts is relatively straightforward, many BDOS and BIOS functions are not. Technical manuals should be consulted for complete information on DOS and BIOS interrupts. For example the IBM's DOS 2.*n* manual provides detailed information in its Appendix D under the title: "DOS Interrupts and Function Calls".

* Do not execute interrupts with random values in the various registers. Such randomness can be hazardous to your computer and software. The best advice would have you do experimentation with interrupts on the floppy-disk based portion of your system -- that is, where the floppy disk is the default drive.

* Most compilers have a header file comparable to intrpt.h. Lattice C's dos.h, for example, contains similar register-oriented structures.

* **intdos** and **intdosx** are interrupt functions which should be applicable across the entire range of MS-DOS 2.x and 3.x operating systems. However **int86** and **int86x** functions are specific to the physical "hardware". Thus **int86** interrupt functions described in the IBM PC documentation probably will work on compatible systems, but may or may not work on other MS-DOS systems. In all cases, refer to the technical documentation for your

computer for the most accurate information on system functions.

* In our examples interrupt functions are called in hexadecimal, since this is the way they are usually listed in technical documentation. You could, of course, use a decimal number (or even octal) if you prefer.

* It important not to confuse the machine registers mentioned above, and the **register** variable class in the C language. While they both ultimately refer to the machine's hardware registers, the "register"-class variables are not pertinent to this discussion.)

RETURNS CPU flags.

EXAMPLE
```
struct XREG {short ax,bx,cx,dx,si,di; };
struct HREG {char al,ah,bl,bh,cl,ch,dl,dh; };
struct SREGS { short es, cs, ss, ds; };
union REGS {
     struct XREG x;
     struct HREG h;
     };

     /* we could have used #include "intrpt.h"
     instead */

main()     /* intdos.c -- tells drive */
{
union REGS r, *rp1, *rp2;

r.h.ah = 0x19; /* execute interrupt 19h */
rp1 = &r;
rp2 = &r;
intdos(rp1,rp2);
printf("\nYou are currently logged on to the
%c: drive.",r.h.al+65);
     /* the drive is returned as number */
     /* 0-n; we add 65 to get a letter */
     /* as in: drive A: or drive D: */
}
```

SEE ALSO intdosx, int86, int86x

PURPOSE Generates a DOS function call which requires the segment registers to be set.

FORM

```
#include "intrpt.h"

main()
{
union REGS r;
struct SREGS s;
char *p;

     r.h.ah = 9;
     p = "hello world!$";
     r.x.dx = (long) p & 15;
     s.ds = (long) p >> 4;
     intdosx( &r, &r, &s);
}
```

USAGE **intdosx** permits execution of a DOS interrupt from within a C program which require information on the memory segment. Once the function has been completed, program control is returned to the executing program.

(See **intdos** for a fuller discussion of the use and application of system interrupts.)

The difference between **intdos** and **intdosx** is that you can also provide data to be placed in the microcomputer's segment registers. **intdosx** should be used when either segment register information needs to be passed to the operating system for particular function calls and/or segment register information is returned by the operating system as a result of the system call.

In the above sample:

`struct SREGS s;`
declares a variable of type **SREGS** named **s**.

`char *p;`
declares a character pointer **p**.

`r.h.ah = 9;`
sets the **AH** variable in **REG r** to 9, meaning that system interrupt 9 will be executed when the **intdosx** function is encountered. Interrupt

9 is the "print-string" function. The string must be terminated by a dollar sign.

`p = "hello, world!$";`
makes **p** point to the dollar-sign terminated string **hello, world!$.**

`r.x.dx = (long) p & 15;`
casts the pointer **p** to a **long** and masks off all but the lowest four bits. This effectively converts a portion of the absolute address to an offset address. This is the offset of **p.**

`s.ds = (long) p >> 4;`
casts the pointer **p** to a **long** and rightshifts the number 4 bits to provide the segment aspect of the address. This is the segment of **p.**

`intdosx(&r,&r,&s);`
performs a system interrupt. The values in **REGS** and **SREGS** are placed in their appropriate registers and a DOS interrupt is executed.

Backgrounder:
How is a computer address like a zip code?
The architecture of the 8088-80286 family of Intel chips divides available memory into 64k segments. This type of chip architecture only permits direct manipulation of data using the *segment address* and a location within the segment, known as the *offset address.*

Unfortunately, this is not the only system of keeping track of memory locations on 16-bit systems. In fact, C programs generally keep track of data items held in memory using an entirely different system known as *absolute memory addressing.*

In a sense the absolute memory location could be compared to the 9-digit zip codes in popular around the U.S. Post Office while the segment/offset address system used by DOS and BIOS functions is comparable to the street/city/state method of locating a destination. (The city being the segment address and the street part of the offset address.)

Just as we could use a computer to convert the 9-digit zip code to a street/city/state address we can use a routine to convert our absolute addresses to segment/offset addresses and back again, if necessary. An example of this, also shown above, is:

```
r.x.dx = (long) p;
```

As it stands, this operation is not very useful but if we combine it with an operation to mask a part of the address we get:

```
r.x.dx = (long) p & 15;
```

Here we *and* the converted value of **p** by 15, thus making only the rightmost 4 bits significant and assigning them to the variable **r.x.dx**. Not coincidentally, it is the last 4 bits of the pointer cast as a **long** which can serve as the object's offset address.

The statement from the sample:

```
s.ds = (long) p >> 4;
```

also casts **p** as a long, but by *right-shifting* four bits only bits 5 and above are assigned to the DS register, which holds the segment address of the string.

If all this seems hopelessly confusing remember that the reason these acrobatics are necessary is to work around some of the inherent hardware limitations of 8086/8088/80286. Fortunately the form shown above can be used any time you need to convert a long pointer to a segment/offset memory address.

NOTES You are advised against executing interrupts with random values in the various registers. The best advice would have you do experimentation with interrupts on the floppy-disk based portion of your system -- that is, where the floppy disk is the default drive.

RETURNS CPU flags.

EXAMPLE

```
#include "intrpt.h"

main()     /* intdosx.c */
     /* prints creation time for a selected file
     */
     /* illustrating use of intdos and intdosx
     */
{char file[20];
union REGS r;
struct SREGS s;
int flags, year, month, day, hour, min, sec,
fhandle;
int c;

while (1)
{    printf("Please give exact filename (or
     <Enter> to exit): ");
     gets(file);
     if(file[0] == 0){
          exit();
     }
     r.h.ah = 0x3d;
     r.h.al = 0;
     r.x.dx = (long) file & 15;
     s.ds = (long) file >> 4;
     flags = intdosx(&r, &r, &s);
          /* opening file and getting file
          handle */
     if((flags & 1) == 1) {
                /* checking for error */
          puts("Error in opening file; probably
          doesn't exist");
          continue;
          }
     fhandle = r.x.ax;
          /* gets time and date using the file
          handle */
     r.h.ah = 0x57;
     r.h.al = 0;
     r.x.bx = fhandle;
     intdos(&r, &r);
```

(Continued next page)

```
            /* decompose binary numbers into date
            and time */
        year = (r.x.dx >> 9) & 63;
        month = (r.x.dx >> 5) & 15;
        day = r.x.dx & 31;
        hour = (r.x.cx >> 11) & 31;
        min = (r.x.cx >> 5) & 63;
        sec = r.x.cx & 31;

        printf("%s        created:        %d-%d-%d
        %d:%d:%d\n\n", file, month, day, year,
        hour, min, sec*2);
        r.h.ah = 0x3e;
        r.x.bx = fhandle;
        intdos(&r, &r);
        /* close file handle */
        }
}
```

SEE ALSO intdos, int86, int86x

PURPOSE Generates a CPU software interrupt.

FORM

```
union REGS r;

int flags;

int intrpt_no;
flags = int86(intrpt_no, &r, &r);
```

USAGE **int86** permits execution of software interrupts including machine-specific BIOS functions. Unlike **intdos** and **intdosx**, this function yields results tied to the computer hardware it is operating on rather than the operating system. Once the routine has been executed, control returns to the executing RUN/C program.

For a more general discussion of interrupts see **intdos**, above.

Note: this discussion centers around ROM BIOS functions in the IBM PC and completely compatible systems. Most if not all other MS-DOS systems use a ROM BIOS, but the function numbers and their effect may vary from IBM's.

BIOS functions allow you to:

* Control the video display (direct video I/O)
* Monitor the asynchronous communication adapter
* Print output
* Control console (keyboard) I/O
* Control low-level disk I/O (no knowledge of files and directories)
* Locate and identify attached devices
* Print the screen

In the above sample:

`union REGS r;`
declares a variable of type **REGS** named **r**.

`int flags;`
declares a variable to hold the DOS returned flags.

`int intrpt_no;`
creates a variable to hold the interrupt number.

`flags = int86(intrpt_no, &r, &r);`
Performs a software interrupt. The values in **REGS r** are placed in their appropriate registers and an interrupt is executed. The CPU returned value is returned to **flags**.

The **union REGS** is defined in intrpt.h. Simply **#include intrpt.h** at the beginning of your program and assign values to registers as shown in the sample and the example below.

NOTES

* Although using simple interrupts is relatively straightforward, making use of many BIOS functions is not. For more information please see the BIOS listing for your computer and other technical information on the machine. If you have a PC compatible, the listing in IBM's Technical Reference Manual is quite readable considering the fact that it is assembly language source code.

* There is some overlap between the DOS and BIOS interrupt functions. Determining which type of interrupt to use often requires a comparitively sophisticated understanding of the differences between the two systems and how the respective function accomplishes its work. Generally speaking, ROM BIOS functions provide faster execution. However, **int86** functions are generally not portable between the various MS-DOS systems.

* You are advised against executing interrupts with random values in the various registers. Such randomness can be hazardous to your computer and software. The best advice would have you do experimentation with interrupts on the floppy-disk based portion of your system -- that is, where the floppy disk is the default drive.

* Most C compilers have a header file comparable to intrpt.h. For the Lattice compiler this file is called dos.h.

* See also the example program WINDOWS.C on the RUN/C distribution disk for an interesting illustration of the power of BIOS function calls.

RETURNS CPU flags.

EXAMPLE

```
#include "intrpt.h"

main()     /* int86.c -- clears the screen on
              IBM PC */
{
union REGS r, *rp1, *rp2;

rp1 = &r;
rp2 = &r;
r.h.ah = 6;       /* scrolls page up */
r.h.al = 0;       /* blanks defined window */
r.h.ch = r.h.cl = 0;
     /* sets top row & left-most column to 0 */

r.h.dl = 79;      /* right-most column is 79 */

r.h.dh = 27;      /* lowest row is 24 */

r.h.bh = 7;
/*
     sets attribute number used to blank the
     lines within the defined window,
to 7.
*/

int86(0x10,rp1,rp2);
              /* executes interrupt 10h */
              /* 10h controls video RAM */
}
```

SEE ALSO intdos, intdosx, int86x

PURPOSE Generates a CPU software interrupt that requires the segment registers set.

FORM

```
union REGS r;
struct SREGS s;

int flags;

int intrpt_no;
flags = int86x(intrpt_no, &r, &r, &s);
```

USAGE **int86x** permits execution from within a C program of BIOS functions which require memory segment information. Once the function has been completed, program control is returned to the executing program.

int86x should be used when segment register information needs to be passed to the BIOS for particular function calls and/or segment register information is returned by the BIOS as a result of the system call.

In a sense, **int86x** complements **intdosx** by providing access to memory segment information for any software interrupt, usually (and most usefully) those tied to BIOS functions. Although there is little use for segment addresses in operating system BIOS functions (for that reason no example of the use of **int86x** is supplied), you may find uses for **int86x** either in your own programs or in other programming tools.

The format for **int86x** is the same as that for **int86** except that the former has an additional argument which references the segment registers; see **intdosx**. The header file intrpt.h can also be used for **int86x**.

In the above sample:

`union REGS r;`
declares a variable of type **REGS** named **r**.

`struct SREGS s;`
declares a variable of type **SREGS** named **s**.

`int flags;`
declares a variable to hold the DOS returned flags.

`int intrpt_no;`
creates a variable to hold the interrupt number.

`flags = int86x(intrpt_no, &r, &r, &s);`
performs a software interrupt. The values in **REGS r** and **SREGS s** are placed in their appropriate registers and an interrupt is executed. The CPU returned value is returned to **flags**.

See **intdos** and **int86** for a fuller discussion of the use and application of system interrupts in C programs.

NOTES You are advised against executing interrupts with random values in the various registers. The best advice would have you do experimentation with interrupts on the floppy-disk based portion of your system -- that is, where the floppy disk is the default drive.

RETURNS CPU flags.

SEE ALSO intdos, intdosx, int86

PURPOSE Checks a character for a particular condition, as described below.

FORM
```
char a;

is_something(a);
```

USAGE *is_something* is not really a function. Rather it is the generic name for a series of functions designed to evaluate what type of character is given to it as an argument.

There are 15 *is_something* tests. Here **c** is the **char** evaluated and the information on the value returned from the tests:

isalpha(c) non-zero if c is alphabetic, 0 if not.

Application: If the character evaluated is in the range **a-z** or **A-Z**, the result will be true.

isupper(c) non-zero if c is upper case, 0 if not.

Application: If the character evaluated is **A-Z**, the result will be true.

islower(c) non-zero if c is lower case, 0 if not.

Application: If the character evaluated is **a-z**, the result will be true.

isdigit(c) non-zero if c is a digit 0-9, 0 if not.

Application: If the character evaluated is **0-9**, the result will be true.

isodigit(c) non-zero if **c** is an octal digit 0-7, 0 if not.

Application: If the character evaluated is **0-7**, the result will be true.

isxdigit(c) non-zero if c is a hexadecimal, 0 if not.

Application: If the character evaluated is **0-9** or **A-F** or **a-f**, the result will be true.

isspace(c) non-zero if c is white space, 0 if not.

Application: If the character evaluated is space (32 decimal), tab (9 decimal) or a newline (10 decimal), the result will be true.

ispunct(c) non-zero if c is a punctuation character, 0 if not.

Application: If the character evaluated is in the decimal range 40-57. 72-100, 133-140 or 173-176, the result will be true.

isalnum(c) non-zero if c is alphabetic or digit, 0 if not.

Application: If the character evaluated is in the range **a-z**, **A-Z** or **0-9**, the result will be true.

isprint(c) non-zero if c is printable (including blank), 0 if not.

Application: If the character evaluated is in the range of decimal 32 through 126 (the ASCII set less control codes), the result will be true.

isgraph(c) non-zero if c is graphic (excluding blank), 0 if not.

Application: If the character evaluated is in the decimal range 32-127.

iscntrl(c) non-zero if c is control character, 0 if not.

Application: If the character evaluated is in the range 0-31 decimal, the result will be true.

isascii(c) non-zero if c is ASCII, 0 if not.

Application: If the character evaluated is in the decimal range of 0-127, the result will be true.

iscsym(c) non-zero if valid character for a C identifier, 0 if not.

Application: If the character evaluated is in the range of **a-z, A-Z, 0-9** or _ (decimal 95), the result will be true.

iscsymf(c) non-zero if valid first character for C identifier, 0 if not.

Application: If the character evaluated is in the decimal range of **a-z, A-Z** and _ (decimal 95), the result will be true.

NOTES Because the *is_somethings* are implemented in RUN/C as functions, you need not be concerned about side effects when using them. You should be aware, however, that most C compilers do implement these "is_tests" as *macros*. This means these "functions" are created using C's *preprocessor* **#define** facility. For example, the Lattice C compiler uses the common convention of defining all of the is_tests in a *header file* named ctype.h which must **#include**'d in the program if any of the *is_something* tests are to be used. See your compiler manual for details.

RETURNS Non-zero value if the test is true.
0, if the test is false.

(Continued next page)

EXAMPLE

```
main()
 /* is_tests.c -- illustrates 'is' functions */
{

char c, t;
puts("type any key on the keyboard\n");
c = getchar();
t = isalpha(c);
putchar(10);     /* linefeed */
if (t != 0)
     printf("that      was      an      alphabetic
     character\n");
t = isupper(c);
if (t != 0)
     printf("and it was upper case\n");
t = islower(c);
if (t != 0)
     printf("and it was lower case\n");
t = isdigit(c);
if (t != 0)
     printf("I saw that digit\n");
t = isxdigit(c);
if (t != 0)
     printf("you  can't  fool  me  with  that
     hexadecimal\n");
t = isodigit();
if (t != 0)
    printf("if you knew octal, like I know octal
...\n");
t = isspace(c);
if (t != 0)
     printf("that was a space or a tab\n");
t = ispunct(c);
if (t != 0)
     printf("ok,   so   you   know   how   to
     punctuate\n");
t = isalnum(c);
if (t != 0)
     printf("that     was     an     alphanumeric
     character\n");
t = isprint(c);
```

(Continued next page)

```
if (t != 0)
     printf("and it was printable\n");
t = isgraph(c);
if (t != 0)
     printf("that was a graphics character\n");
t = iscntrl(c);
if (t != 0)
     printf("that was a control character\n");
t = isascii(c);
if (t != 0)
     printf("well, at least it was ASCII\n");
t = iscsym(c);
if (t != 0)
     printf("it could be a valid character for a
     c identifier.\n");
t = iscsymf(c);
if (t != 0)
     printf("it  could  even  be  the  first
     character in a c identifier \n");
}
```

PURPOSE Deletes the specified file from the specified drive/directory.

FORM KILL FILESPEC

USAGE **KILL** erases a file from the disk. The **KILL** and **UNLINK** commands are similar to DEL and ERASE in DOS. The exact name of the file including its extension, if any, must be given. Standard MS-DOS filename and pathname conventions apply. (For more information on proper file names see Section 1.)

In the above sample:

```
KILL FILENAME.C
```

will delete the file FILENAME.C, if it exists on the current drive/directory. The command:

```
KILL A:BACKUP\MYDISK\FILE2.C
```

will delete FILE2.C from the specified drive/directory, if it exists.

If the file is not found on the specified drive/directory, the message:

```
Cannot remove filespec
```

will be displayed.

Use of quotation marks around the file name is optional. But a space after **KILL** is mandatory. This is also true for **LOAD, SAVE** and **UNLINK.**

NOTES In RUN/C Professional **KILL** can be called from the executing program using the **runc()** command.

EXAMPLE

```
[1] KILL "NEWTEST.C"
    Ok

[2] kill \cprogs\oldfiles\try1.c
    Ok
```

SEE ALSO UNLINK

PURPOSE Print the current values of active variables after the program has been interrupted.

FORM

```
Break in line n
     Continue, Dump, Ldump, Interactive, Help or
     End (C,D,L,I,H,E)?

L
```

USAGE Ldump, a mini-menu command, allows you to print the value of active variables after you have interrupted program execution by pressing <Ctrl-c> or after a breakpoint has been encountered.

This debugging aid can come in quite handy when you are trying to determine where your program has gone awry. It is available only after a program's execution has been interrupted. Upon interruption the following menu will appear:

```
Break in line n
     Continue, Dump, Ldump, Interactive,
     Help or End (C,D,L,I,H,E)? L
```

If a **L** is selected, the values of all active variables will be printed. Then the mini-menu will reappear, allowing you to continue program execution or end the program.

For more information on the use of <Ctrl-c> and <Ctrl-Break> see Section 1. For more information on the use of the **BREAK** command in conjunction with the **runc()** function see Section 6.

NOTES

* A printer must be installed and on-line for this command to function properly. Otherwise, an error message generated by your operating system will be displayed when an attempt is made to use the L(Ldump) option. For more information on attaching printers to your system, see the hardware manual for your machine or contact your dealer.

* An alternative to using the L(Ldump) mini-menu command is to use the <Ctrl-p> or <Ctrl-PtrSc> (on PC compatibles) key combination in conjunction with D(Dump) to cause the printer to echo screen output.

EXAMPLE See example under **D(Dump)**.

SEE ALSO C(Continue), D(Dump), E(End), H(Help), I(Interactive), Section 6

PURPOSE Displays RUN/C Professional library commands.

FORM LIB

USAGE LIB displays the names of RUN/C Professional Loadable Library commands and provides a brief description of each. When the command is invoked the following is displayed:

```
Loadable Library commands:

LIBDIR [drive:][dir] (list all libraries in current or specified directory)
LIBLOAD libname  (load library)
LIBSHOW [libname]  (list given or all loaded libraries)
LIBUNLOAD  (unload all libraries)
   Abbreviations: LIBD, LIBL, LIBS, LIBU
Ok
```

Each of these commands is described in detail in Section 5 of the RUN/C Professional manual.

PURPOSE Displays portions of the program in memory.

FORM

[1] LIST

[2] LIST *line a*

[3] LIST *lines a-z*

[4] LIST *lines a-*

[5] LIST *lines -z*

USAGE **LIST** displays program lines by their *autoline* numbers either individually or in a specified range. Line numbers are dynamically assigned to a program or program fragment in memory each time it is loaded into memory.

In the above sample:

line a represents the first line to be displayed.
line z represents the last line to be displayed.

thus

LIST
displays all lines in the program in memory.

LIST a
displays the line associated with a single line number.

LIST a-z
displays lines from line a to line z inclusively (both line a and line z are displayed).

LIST a-
displays text from line associated with line a to the end of the program.

LIST -z
displays all lines up to and including autoline z.

If a line number is specified which does not exist in the program, the message:

```
No such line number
```

will appear and no lines will be displayed.

If a range of lines is specified which includes non-existing line numbers, the message:

```
Invalid range
```

will be displayed.

NOTES

* The MS-DOS <Ctrl-s> display start/stop toggle may be used to start and stop the listing of the program in memory. Additionally, the listing may be interrupted with a <Ctrl-c>.

* Printer echo (activated by <Ctrl-p> or on PC compatibles <Ctrl-PrtSc>) can also be active during a LISTing.

* In RUN/C Professional **LIST** can be called from the executing program using the **runc()** command as in:

```
runc("LIST 20-40");
```

LIMITS It is occasionally necessary to press <Ctrl-c> more than once in order to halt the listing of a program.

EXAMPLE

Command	Effect
LIST	Displays entire program.
LIST 20	Displays line 20.
LIST 20-30	Displays lines 20 through 30.
LIST -15	Displays from the beginning of the program up to (and including) line 15.
LIST 20-	Displays from line 20 to the end of the program.

SEE ALSO LLIST, NLIST, NLLIST

PURPOSE Prints portions of the program in memory.

FORM

[1] LLIST

[2] LLIST *line* a

[3] LLIST *lines* a-z

[4] LLIST *lines* a-

[5] LLIST *lines* -z

USAGE LLIST sends program lines according to their *autoline* numbers to the list device (system printer). For a detailed explanation of the above sample forms see LIST.

NOTES

* The MS-DOS <Ctrl-s> start/stop display toggle may be used to start and stop the listing of the program in memory. Additionally, the listing may be interrupted with a <Ctrl-c>.

* A printer must be installed and on-line for this command to function properly. Otherwise, an error message generated by your operating system will be displayed when an attempt is made to LLIST. For more information on attaching printers to your system, see the hardware manual for your machine or contact your dealer.

* An alternative to using LLIST or NLLIST is to use the <Ctrl-p> or <Ctrl-PtrSc>) on PC compatibles key combination in conjunction with LIST or NLIST to cause the printer to echo screen output.

* In RUN/C Professional LLIST can be called from the executing program using the **runc()** command.

LIMITS It is occasionally necessary to press <Ctrl-c> more than once in order to immediately end an LLIST operation.

(Continued next page)

EXAMPLE

Command	Effect
LLIST	Prints the entire program.
LLIST 20	Prints line 20.
LLIST 20-30	Prints lines 20 through 30.
LLIST -15	Prints lines from the beginning of the program up to (and including) line 15.
LLIST 20-	Prints line numbers from line 20 to the end of the program.

SEE ALSO LIST

PURPOSE Reads a program into memory.

FORM [1] LOAD *filename*

[2] LOAD *[drive:] [\directory] filename*

[3] LOAD

USAGE **LOAD** reads a file from the default drive/directory or a specified drive/directory into memory. The **LOAD** command also clears any program currently resident in memory prior to loading the specified file.

As the file is loaded, the message:

Loading line *nnnn*

will be displayed. *n* is a counter displaying every fifth program line number as it is loaded into memory from the disk. When the last line is encountered, that number is also displayed.

Standard naming conventions for filenames, disk and directory pathnames as described in Section 1, apply. If no drive or directory is specified, the current default drive/directory will be assumed.

Similarly to the **SAVE** commands, RUN/C assumes that the extension of the file is .C unless an extension is supplied. Thus:

LOAD TEST

will actually search for and, if found, load the file TEST.C into memory.

Of course you may override this convention by supplying the extension name. Therefore:

LOAD TEST.2

will search for and, if found, load TEST.2 into memory.

The only "got'cha" in this scheme is that you cannot simply load a file with no last name without a little acrobatics. For example, if you have a file on disk named simply MYFILE, the command:

```
LOAD MYFILE
```

would search for something named MYFILE.C, not MYFILE. The way around this is to supply the bare beginning of an extension name for files which have none. Thus the correct way to load MYFILE is with a period at the end of the name, as in:

```
LOAD MYFILE.
```

If a file of the specified name is not found, the error message:

```
File not found
```

will be displayed.

Of course, you can supply the .C extension if you wish. Thus the commands:

```
LOAD TEST
```

and

```
LOAD TEST.C
```

have exactly the same meaning and effect.

LOAD by itself, illustrated in Form [3], will load the special file PROGRAM.C if it exists on the current drive/directory. See **EDITOR** and **SHELL** in this section and in Section 2 for more information on PROGRAM.C.

Use of quotation marks around the file name is optional; but a space after **LOAD** is mandatory. This also holds true for **SAVE, KILL** and **UNLINK.**

(Continued next page)

EXAMPLES

	Command	Effect
[1]	**LOAD TEST**	Reads the program TEST.C into memory.
[2]	**LOAD B:TEST.C**	Reads the program TEST.C on the "B:" drive into memory.
[3]	**LOAD "B:\CPROGS\TEST.C**	Reads the program TEST.C on the "B:" drive in subdirectory "CPROGS" into memory.
[4]	**LOAD**	Reads the default program PROGRAM.C into memory, if it exists on the default drive/directory.

SEE ALSO RUN, MERGE, SAVE

PURPOSE Moves the cursor or determines its location.

FORM

```
int y, x, mode, ret;

ret = locate(y,x,mode);
```

USAGE **locate** provides direct access to the cursor and/or coordinated locations on the screen.

In the above sample:

`int ret;`
declares a variable to hold the returned value.

`int y;`
declares a variable for the y-axis.

`int x;`
declares a variable for the x-axis.

`int mode;`
declares a variable to hold the selected mode.

`ret = locate(y,x,mode);`
locates either the cursor or a position on the screen, depending upon mode selected. The returned value from the function is assigned to **ret.**

The screen coordinates* are defined as:

Coordinate	Definition
Upper left corner	y = 1; x = 1
Lower right corner	y = 25; x = 80

(Continued next page)

* Assumes a 25x80 screen.

The modes (0-3) have the following effects:

Mode	Effect
0	Absolute move. **locate(y,x,0)** will move the cursor to the specified (**y**,**x**) location; if the location (**y**,**x**) is not on the screen, the cursor will not be moved. This mode returns a 0, if successful; -1 is returned if an attempt is made to locate the cursor off the screen.
1	Relative move. **locate(y,x,1)** will incrementally move the cursor vertically by **y** units, horizontally by **x** units (hence **x** and/or **y** may be positive or negative); if the resultant location is off the screen, the cursor will not be moved. This mode returns a 0, if successful; -1 is returned if an attempt is made to locate the cursor off the screen.
2	Current cursor line location. **locate(y,x,2)** will return the row number that the cursor is on (in the range 1-25). **x** and **y** passed values are ignored. However, if an absolute cursor move cannot be performed (i.e., on certain non-IBM PC systems), -1 will be returned, indicating that this mode of **locate** is not available.
3	Current cursor row location. **locate(y,x,3)** will return the column number that the cursor is on (in the range 1-80). **x** and **y** passed values are ignored. However, if an absolute cursor move cannot be performed (i.e., on certain non-IBM PC systems), -1 will be returned, indicating that this mode of **locate** is not available.

NOTES

* For modes 2 and 3 *dummy values* must be supplied for the **x** and **y** coordinates, even though those values are ignored.

* If the RUN/C built-in function **locate** function is referenced in a file which you attempt to compile, an error message will be generated during the compilation process unless you include a clear screen function of the same name. Such a file is included on the RUN/C distribution disk in two forms: RCIBMLIB.C, for IBM PC compatible computers and RCANSBLIB.C for systems supporting the ANSI screen addressing standard.

RETURNS

Mode 0: 0 if successful; -1 if an attempt is made to move the cursor off the screen.

Mode 1: 0 if successful; -1 if an attempt is made to move the cursor off the screen.

Mode 2: Line number where the cursor is located; -1, if the mode is not available on the system in use.

Mode 3: Row number where the cursor is located; -1, if the mode is not available on the system is use.

(Continued next page)

EXAMPLE

```
main()      /* locate.c -- illustrates use of
               locate() */
{

int a,b,c,d;
c = 1;
b = 1;
a=1;

cls();

locate(3,35,0);
puts("RUN/C SAMPLE BOX");
locate(1,1,0);
puts("\t* * * * * * * * * * * * * * * * **** * * *
* * * * * * * * * * * *");
     while(a < 12){
  b *= -1;
     (b > 0) ? locate(0,64,1) : locate(++a,8,0);
     putchar('*');
}

locate(0,64,1);
putchar('*');

locate(12,8,0);
d = 9;
     while(c < 13)
     {
          b *= -1;
          (b  >  0)  ?  locate(12,(80-d),0)  :
          locate(12,d,0);
          putchar('*');
          ++c;
          ++d;
     }
locate(12,23,0);
puts("Ok, but what about windows?");

}
```

PURPOSE Calculates the logarithm of a number.

FORM

```
double result;
double number;

result = log(number);
```

USAGE **log** returns the natural logarithm of a number in base e. The numeric argument must be positive. Both the passed number and the returned value are expressed as doubles.

NOTES In RUN/C Professional this and other transcendental math functions are available in the loadable library MATHLIB.

RETURNS Logarithm of a number, if positive.
0, if number is zero or negative.

EXAMPLE

```
main()

/* log.c -- prints (short) table of natural
logarithms */

{
double n, ln;

printf("  n\t\t\t  ln(n)\n");
for (n = 1.0; n < 1.5; n += 0.01) {
     ln = log(n);
     printf("%.2f\t\t\t%.5f\n",n,ln);
}
puts("Correct last two lines:");
puts("1.48\t\t\t0.39204");
puts("1.49\t\t\t0.39878");
}
```

PURPOSE Calculates the base 10 logarithm of a number.

FORM

```
double result;
double number;

result = log10(number);
```

USAGE **log** returns the logarithm of a number in base 10. The numeric argument must be positive. Both the passed number and the returned value are expressed as doubles.

NOTES In RUN/C Professional this and other transcendental math functions are available in the loadable library MATHLIB.

RETURNS Logarithm base 10 of a number, if positive.
0, if number is zero or negative.

EXAMPLE

```
main()

/* log10.c -- prints (short) table of common
(base 10) logarithms */

{
double n, 110;

printf("  n\t\t\t  log  (n)\n");
printf("   \t\t\t     10\n");
for (n = 1.0; n < 1.5; n += 0.01) {
     110 = log10(n);
     printf("%.2f\t\t\t%.5f\n",n,110);
     }
puts("Correct last two lines:");
puts("1.48\t\t\t0.17026");
puts("1.49\t\t\t0.17319");
}
```

PURPOSE Creates an integer variable 32 bits in length (-2,14,483,648 to 2,147,483,647).

FORM
```
long a;
a = 23293472394L;
```

USAGE A **long** provides twice as much data storage space as an **int**, hence the name. A **long** is a whole number with a range of -2e31 to 2e31-1; no decimal points are allowed. A minus or plus can preceded the number assigned to a **long**. If no sign is given, a positive number will be assumed.

NOTES

* An "L" must follow any constant which is being assigned to a variable of type **long**.

* Various data types used in the **printf** and **scanf** can be converted to their **long** equivalents. For example:

```
printf("%Ld", a);
```

  provides for a long integer. See **printf** and **scanf** for more information on such conversions.

* In C, simple data type names (such as **long**) have four distinct uses, namely:

  (1) in declaring variables and aggregate numbers;

  (2) in casting one type of value to another such as:

```
d = (long) 0;
```

  (3) as an argument to **sizeof**, as in:

```
int i = sizeof(long);
```

  (4) as an element in a **typedef** statement used in creating a synonym for an existing data type such as:

```
typedef long LARGE;
```

* The constructions **long int** and **long float** are not supported.

SEE ALSO char, double, float, int, short, unsigned

PURPOSE Sends formatted output to the printer.

FORM `lprintf("format string", arg1,arg2,arg3..arg n);`

USAGE **lprintf**, like its cousins **printf, sprintf** and **fprintf**, is designed to format and print data. It allows the results of calculations to be mixed with generalized comments and sent to the list device. For a complete discussion of **lprintf** formatting options see **printf.**

In the above sample:

`lprintf("format", arg1, arg2, arg3 .. arg n);`
prints the formatted string to the line printer.

lprintf uses several formatting codes based around the form **%a** where the **%** alerts the function that a special character is coming. The character which follows the **%** tells **lprintf** what to do with the data being passed to it.

The data type options and their meanings are:

Option	Data type
%c	character
%d	signed decimal
%e	exponential floating point
%f	fixed decimal floating point
%g	**e** of **f** format, whichever is shorter
%o	octal
%s	string
%u	unsigned decimal
%x	hexadecimal
%%	prints **%**

Like **printf, lprintf** allows you to optionally place additional instructions between the **%** and the conversion character. This is the *precision field*. It provides further capability to exactly control the form of the printed output.

The form for this can be expressed as:

% - # . # *L a*

(In actual usage there are no spaces between the codes in the precision field; they are included here for readability.)

The conversion options and their meanings are:

Option	Meaning
-	left justify output
#	minimum number of spaces in field
.#	for whole numbers, determines the number of characters which can be passed to the field. For floating point numbers, the second # tell how many decimal places to provide for the formatted number.
l	declares that the **int, float** or **unsigned** shall be treated as a **long.**

NOTES

* When **lprintf** is used, newlines (\n) are expanded to carriage return/line feed (0D/0Ah).

* If a line printer is not properly initialized, an operating system message will appear. Turn on the printer and retry the **lprintf** operation, or abort the operation with a <Ctrl-c> and return to DOS.

* The function **lprintf** is modeled after a similarly-named function in Lattice's C Food Smorgasbord. The object library or source code from this collection of utility function is needed in order to compile a program using **lprintf.**

RETURNS EOF, if an error is detected.
0, if successful.

(Continued next page)

EXAMPLE

```
main() /* lprintf.c -- demonstrates lprintf() */

{
    char c;
    int i;
    char *cp;
    double d;

    c = 'A';
    i = 999;
    cp = "This is an arbitrary string.";
    d = 123.456;
    lprintf("%c %d %s %lf\n", c, i, cp, d);
    puts("Correct answer (on printer):A 999
    This is an arbitrary string. 123.456000");
}
```

SEE ALSO printf

PURPOSE Sends a character to the printer.

FORM

```
int ret;
char c;
ret = lputc(c);
```

USAGE **lputc** sends a character to the currently assigned list device (system printer).

In the above sample:

`int ret;`
declares a variable to hold the returned code.

`char c;`
declares a character variable.

`ret = lputc(c);`
sends character **c** to the printer. The returned value from the function will be assigned to **ret**.

NOTES

* A line feed is automatically expanded (translated) into a carriage return/line feed (0D/0Ah) sequence.

* The function **lputc** is modeled after a similarly named function found in Lattice's C Food Smorgasbord. The object library file or source code from this function utility package is needed in order to compile a program using **lputc**.

* If a line printer is not properly initialized, an operating system message will appear. Turn on the printer and retry the **lprintf** operation, or abort the operation with a <Ctrl-c> and return to DOS.

RETURNS EOF, if error is detected.
Character sent, if successful.

EXAMPLE

```
main()      /* lputc.c   -- typewriter program
               using lputc.c */

{
int c;

puts("I'm a typewriter. When you want to
stop,");
puts("press Ctrl-Z. Please type below:");
while((c = getchar()) != EOF)
    lputc(c);
puts("Correct answer (on printer) is what you
typed.");
}
```

SEE ALSO putc, lprintf

PURPOSE Sends a string of characters to the printer.

FORM

```
char *s;
s = "hello, printer";

lputs(s);
```

USAGE **lputs** sends a character string to the list device (system printer). An embedded newline (\n) will be expanded to a carriage return/line feed. **lputs** automatically provides a carriage return/linefeed (0D/0Ah) sequence at the end of the output string.

In the above sample:

`char *s;`
declares a pointer and sets it to the address where the string to be printed is stored.

`s = "hello, printer";`
assigns the string to the area in memory pointed to by **s**.

`lputs(s);`
sends the string at address **s** to the list device.

NOTES
* The function **lputs** is based on a similarly named function in Lattice's C Food Smorgasbord. The object library file or source code from that utility program is needed in order to compile a program using **lputs**.
* If a line printer is not properly initialized, nothing will print.

RETURNS Nothing.

(Continued next page)

EXAMPLE

```
main()      /* lputs.c -- illustrates lputs() as a
               typewriter program */

{
     char ac[128];

     puts("I'm a typewriter. When you want to
     stop, begin your last");
     puts("line with a $. Please type below:");
     do {
          gets(ac);
          lputs(ac);
          lputc('\n');
     } while (ac[0] != '$');
     puts("Correct answer (on printer) is what
     you typed.");
}
```

SEE ALSO puts, lprintf

PURPOSE Seeks to a specified position in the file.

FORM
```
long pos;
int file;
long offset;
int mode;
     pos = lseek(file, offset, mode);
```

USAGE **lseek**, *level 1* (unbuffered) I/O function shifts the current character position pointer for a disk file. The number of characters of movement (known as the *offset position*) is specified as a **long** and is either changed relative to the beginning of the file (mode 0), the current file position (mode 1), or the end of the file (mode 2).

In the above sample:

`long pos;`
declares a variable to hold the returned file position or an error code.

`int file;`
declares the file number; its value is obtained from **open.**

`long offset;`
sets desired shift amount.

`int mode;`
sets offset mode which determines the starting position of the seek.

`pos = lseek(file, offset, mode);`
the file position pointer is moved in the specified file by the offset value according to the specified mode. The returned value from the function will be assigned to **pos.**

The variable **pos** will either receive the new file position or EOF, in case of an error. The most likely cause of an error is an attempt to move beyond the beginning or end of a file.

NOTES * The primary difference between **lseek** and **fseek** is that **lseek** uses *Level* 1 (unbuffered) I/O with files opened using the "primitive" functions **open** or

creat, while **fseek** is based upon disk access via file pointers. **fseek** is used in *Level* 2 (buffered) file operations.

* The offset value of the seek may be positive or negative.

RETURNS -1L, if an error is detected.
Current file position, if successful.

EXAMPLE
```
main()      /* lseek.c -- demonstrates lseek() */
/* Requires TEST.FIL (a program containing 30
consecutively-numbered lines */

{
     int i, fi;
     char buf[128];
     long posn;

     for (i = 0; i < 25; i++)
          buf[i] = '\0';
     if((fi = open("test.fil",0x8000)) == EOF) {
          printf("Can't open test.fil\n");
          exit();
     }
     i = read(fi,buf,20);
     puts(buf);
     posn = lseek(fi,0L,1);
     printf("Current file position: %ld\n",
     posn);
     close(fi);
     puts("\n ... And here is the correct
     answer:\n1\n2\n3\n4\n5\n6\n7\nCurrent
     file position: 20");
}
```

SEE ALSO open, fseek

PURPOSE Serves as the primary function in a C program; a required component of any executable RUN/C program. Arguments to main are optional.

FORM

```
main(argc,argv)

int argc;
char *argv[];

{
     statement(s);
     functions();
}
```

USAGE **main()** is the point where execution of a C program always begins. Any functions which are called by the program must either be called from **main()*** directly or indirectly through another function or series of functions which were initially called from **main()**.

It is also through **main()** that command line arguments may be passed to the program. These take the form of data separated by spaces. The arguments -- or parameters -- can come to RUN/C in two ways: 1) from the DOS command line, supplied when RUN/C is invoked, or 2) from the RUN/C command line, supplied following **RUN** or **EXEC** (in RUN/C Professional).

In the above sample:

`main(argc,argv)`
: **main()** is declared with the optional arguments.

`int argc;`
: an integer is declared to hold the number of command line arguments.

`char *argv[];`
: a pointer to an array of character strings which contain the arguments, one per string,

---

* Traditionally, C function names are followed by parentheses, as we are doing in this discussion of **main()**. However, we generally do not denote function names with the parentheses in this manual because of the potential confusion between functions which require arguments (such as **printf**) and those which do not (such as **exit**).

is declared. Alternative this can be expressed as:

```
char **argv;
```

which is a pointer to a pointer to **char** to reference arguments.

{

the opening bracket signals the beginning of the **main()** function.

statement(s);

symbolizes any number of statements that may be contained in the **main()** function.

functions();

symbolizes any number of function calls that may be contained in the **main()** function.

}

the closing bracket signals the end of the **main()** function.

Arguments to **main()** are handled by RUN/C in essentially the same way as arguments to any other function. The only differences are that the format for passing command line arguments to **main()** is somewhat more rigid. The first argument (traditionally called **argc**) is automatically assigned the number of strings separated by *white space**, supplied when the program was invoked, of which the command RUN or the program name (the *DOS invocation*) counts as one. The second argument (traditionally called **argv**) points to a series of pointers, each of which points to one of the command line argument strings. Of course you may call the arguments to **main()** whatever you like as long as you give them reasonable data type names.

Once arguments have been passed to **main()** and are assigned to the proper data type variable, they behave exactly as arguments passed between functions.

* RUN/C also permits strings enclosed in double-quotation marks to be passed as single arguments to **main()**, even if they contain white space.

An example

Suppose we want our computer to say hello to us once in a while. (Around here we have some really slow days!) Here is SIGN-ON.C, a program that will allow you to give our name and get a greeting.

We want to call SIGN-ON from the DOS command line:

```
A>RC SIGN-ON STEVE
```

From these instructions we know that RUN/C is expected to be accessible from the default drive/directory, that RUN/C will attempt to load a program named SIGN-ON.C and try to pass one character string to **main()**. Looking at it another way, we are passing two arguments to **main()** -- the name of the program and the string itself.

Here is our program:

`main(argc,argv)`

main() is declared with the optional arguments using traditional C names.

`int argc;`

an integer is declared to hold the number of command line arguments.

`char **argv;`

a pointer to a pointer to a **char** is declared. (**argv** points to the first of an array of character pointers each of which points to a command line argument.)

`{`

(now the fun begins)

`printf("Hi ya, %s!\n",argv[1]);`

we print the string contained in argument No. 1 passed from the command line.

`}`

signals the end of **main()**.

When RUN/C is invoked with the above message:

```
A>RUN SIGN-ON STEVE
```

The message:

```
Hi ya Steve!
```

will appear on the screen.

NOTES

* It is important for newcomers to C to recognize that arguments to **main()** are an advanced feature of C. It is not necessary to declare arguments to main() or to use command line arguments when creating programs. The only RUN/C example program which uses arguments to **main()** is the second example in this discussion.

* When a program is called from the DOS command line, control will return to DOS -- not to the RUN/C **Ok** prompt -- if and only if the program successfully executes. If an error occurs, appropriate error messages will be generated within RUN/C and program control remains in RUN/C.

* In the previous discussion we declared the variable **argv** in two different ways:

```
char argv[];
```

and

```
char **argv;
```

These declarations mean the same thing in C: that **argv** points to a pointer which in turn points to a character (**argv** points to the first of a series of pointers each of which points to a word on the command line). We use the first method simply as a reminder to ourselves that **argv** will be used as an array to access each of the character pointers by indexing:

```
char *argv[];
puts(argv[2]; /* prints out the 3rd
word on the command line */
```

Using method No. 2:

```
char **argv;
puts(*(argv+2)); /* same as above */
```

The only real difference is that in the second method, **argv** can actually be changed (although it is not recommended):

```
char **argv;
argv = argv + 2;
puts(argv); /* same as above */
```

RETURNS Nothing.

EXAMPLE [1]

```
main()

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

[2]

```
/*  main.c  --  prints  argument  count,
arguments from RUN line */

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

    printf("argc = %d\n",argc);
    for (i = 0; i < argc; i++)
        printf("argv[%d]: |%s|\n",i,argv[i]);

/* Result: each argument passed to main is
counted and listed within vertical brackets */

}
```

SEE ALSO **RUN** for a discussion of I/O redirection from the RUN command line; **EXEC** in RUN/C Professional.

PURPOSE Dynamically allocates a block of memory.

FORM

```
char *p;
unsigned nbytes;

p = malloc(nbytes);
```

USAGE **malloc**, standing for "memory allocation," sections off a contiguous group of bytes in memory.

In the above sample:

`char *p;`
declares a character pointer.

`unsigned nbytes;`
declares an unsigned integer to hold the number of bytes to be allocated.

`p = malloc(nbytes);`
nbytes of memory will be allocated. The **p** pointer will be assigned the address of the first byte of the contiguous allocated memory.

Memory allocations are automatically released at the end of program execution (at the end of a **RUN**), whether the program ends normally, is stopped by an error or is interrupted with a <Ctrl-c>.

NOTES

* Unlike memory obtained with **calloc**, bytes in the area of memory allocated with **malloc** are not initially set to zero.

* Be certain to retain the pointer value returned by **malloc** if you plan to later release the allocated memory by using **free**.

* A *cast* must be used if the block is to store objects other than **chars**.

RETURNS NULL, if insufficient memory is available.
Address of the first byte of the memory block, if successful.

EXAMPLE

```
main()      /* malloc.c -- demonstrates malloc()
            and free() */
{
char *memloc;
int i;
long l;

     memloc = malloc(1000);
     l = (long) memloc;
     printf("Allocated memory starts at %ld\n",
     l);
     i = free(memloc);
     if (i == 0)
          puts("Successful release");
     else
          puts("Release failure");
     memloc = malloc(1000);
     l = (long) memloc;
     printf("New allocation starts at %ld\n",
     l);
     puts("Correct output: numbers are system
     dependent but should be equal");
}
```

SEE ALSO calloc, free

PURPOSE Copy a block of memory to a target area.

FORM
```
char *source;
char *destin;
unsigned num;
movmem(source, destin, num);
```

USAGE **movmem**, standing for "move (in) memory", copies a fixed number of bytes from one location in memory to another. In cases where there is an overlapping of the source and destination blocks, the data is copied in such a way as to insure a complete copy at the destination, even if part of the source block must ultimately be overwritten.

In the above sample:

`char *source;`
declares a character pointer to the beginning address of the source block.

`char *destin;`
declares a character pointer to the beginning address of the destination of the block to be copied.

`unsigned num;`
declares an unsigned variable to hold the number of bytes to be moved, up to 65,535.

`movmem(source, destin, num);`
moves a block of **num** bytes to the **destin** address, overwriting any existing data in that memory block.

NOTES
* A cast operator should be used when arrays or pointers of types other than pointers to characters are used for the source and destination arguments.

* To further illustrate **movmem** we have included on the RUN/C distribution disk a program called VIDEO.C. This program moves several thousand bytes from the first position in memory to the area of memory known as *video RAM*. The screen memory addresses provided are for the IBM PC, but they may be changed to accomodate other *memory mapped video system*.

This program illustrates a technique often used to boost the display speed of text on video-mapped screens. Note: If you attempt to use this program be prepared to have garbage displayed on the screen. However, RUN/C should still be available; try pressing <Ctrl-c> a few times. Another note: RUN/C does no checking to see that the destination pointer in a **movmem** operation points to an appropriate destination. If the pointer is NULL (a very likely condition if you have not assigned a value to it) and you **movmem** to it, you will write memory right over operating system interrupt vectors, effectively bombing your system and requiring a cold reboot. As written, VIDEO.C is harmless. Modify it with care.

RETURNS Nothing.

EXAMPLE

```
main()

/* movmem.c -- moves data from one
element (source) to another (destin) */
{
char a[80], b[80], c[80], d[80];
int i;

struct SAMPLE {
     char first[80];
     char second[80];
     } source[20], destin[20];

for (i = 0; i < 4; ++i ){
     printf("Type string %c followed by return: ", i+65);
     if(i == 0)
          gets(a);
     if(i == 1)
          gets(b);
     if(i == 2)
          gets(c);
     if(i == 3)
          gets(d);
}
```

```
/*
     We get four strings assigning them to char arrays a, b,
     c and d. No string should exceed the 80 byte maximum.
*/

strcpy(source[0].first, a);
strcpy(source[0].second, b);
strcpy(source[19].first, c);
strcpy(source[19].second, d);

/*
     This routine copies the supplied strings into either
     the first or second 80-element array of either the
     first or last (20th) data structure of type source.
*/

movmem((char *) source, (char *) destin, sizeof(source));

/*
     Here we first determine the number of bytes in source
     using the sizeof function. This becomes the number of
     bytes we will move within memory. Then we move that
     many bytes beginning with the first byte of source to
     the destin data structure. Since destin has the exact
     same structure and is therefore the same size as
     source, we do not need to worry about writing over
     valuable data.
*/

printf("%s\n%s\n", destin[0].first, destin[0].second);
printf("%s\n%s\n", destin[19].first, destin[19].second);

/*
     Here we print out the contents of two destin arrays.
     The fact that destin now contains the strings we
     assigned to source indicates that the movmem
     operation succeeded.
*/
}
```

SEE ALSO calloc, malloc, free

PURPOSE Deletes the program currently in memory and clears the screen.

FORM **NEW**

USAGE **NEW** clears the portion of memory allocated to RUN/C programs and the files and data structures created by those programs. It addition, a clear screen is executed.

NOTES

* When a **NEW** is executed, RUN/C performs some necessary tidying up of memory at a rate of approximately 150 lines per second. On larger programs a **WAIT** message will appear at the top of the screen until the previous program is cleared.

* RUN/C Professional loaded function libraries are not affected by a **NEW**. The only way to remove such a library from memory is to use the **LIBUNLOAD** command, described in Section 5.

EXAMPLE

```
Ok

LIST

1:main()
2:{
3:   printf("Hiya!\n");
4:}

NEW

Ok

LIST

Ok
```

(Result: nothing is listed because the program in memory has been deleted.)

SEE ALSO Section 5 (LIBUNLOAD)

PURPOSE Displays specified portions of the program in memory without *autoline* numbers.

FORM

[1] NLIST

[2] NLIST *line* a

[3] NLIST *lines* a-z

[4] NLIST *lines* a-

[5] NLIST *lines* -z

USAGE NLIST displays program lines by their *autoline* numbers either individually or in a specified range. NLIST displays no line numbers. Thus the listing on the screen will be exactly the same as the program which would be saved to the disk. For a detailed explanation of the above sample forms see **LIST**.

NOTES

* The MS-DOS <Ctrl-s> display toggle may be used to start and stop the listing of the program in memory. Additionally, the listing may be interrupted with a <Ctrl-c>.

* Printer echo (activated by <Ctrl-p> or on PC compatibles <Ctrl-PrtSc>) can also be active during a LISTing.

* In RUN/C Professional **NLIST** can be called from the executing program using the **runc()** command.

EXAMPLE

Command	Effect
NLIST	Lists without autoline numbers all lines in the program.
NLIST 20	List with no line number line 20.
NLIST 20-30	List without autoline numbers lines 20 through 30 inclusively.
NLIST -15	List without autoline numbers from the beginning of the program up to and including line 15.
NLIST 20-	List with no line numbers from line 20 to the end of the program.

SEE ALSO LIST

PURPOSE Prints specified portions of the program in memory without *autoline* numbers.

FORM

[1] NLLIST

[2] NLLIST *line* a

[3] NLLIST *lines* a-z

[4] NLLIST *lines* a-

[5] NLLIST *lines* -z

USAGE **NLLIST** prints on the list device (system printer) program lines by their *autoline* numbers; however, the line numbers themselves are not printed. Thus the listing will be exactly the same as the program which would be saved to disk. For a detailed explanation of the above sample forms see **LIST**.

NOTES

* A printer must be installed and on-line for this command to function properly. Otherwise, an error message generated by your operating system will be displayed when an attempt is made to **NLLIST**. For more information on attaching printers to your system, see the hardware manual for your machine or your dealer.

* The MS-DOS <Ctrl-s> display toggle may be used to start and stop the listing of the program in memory. Also, the listing may be interrupted with <Ctrl-c>.

* An alternative to using **NLLIST** or **LLIST** is to use the <Ctrl-p> (or <Ctrl-PtrSc> on PC compatibles) key combination in conjunction with **LIST** or **NLIST** to cause the printer to echo screen output.

* In RUN/C Professional **NLLIST** can be called from the executing program using the **runc()** command.

LIMITS

* It is sometimes necessary to press <Ctrl-c> several times in order to immediately end an **NLLIST** operation.

EXAMPLES

Command	Effect
NLLIST	Prints the entire program without autoline numbers.
NLLIST 20	Prints line 20 without a line number.
NLLIST 20-30	Prints without autoline numbers lines 20 through 30 inclusively.
NLLIST -15	Prints without autoline numbers lines from the beginning of the program up to (and including) line 15.
NLLIST 20-	Prints without line numbers from line 20 to the end of the program.

SEE ALSO LIST

PURPOSE Opens a disk file for unbuffered read or write operations.

FORM

```
int file;
char *filename;
int mode;

file = open(name, mode);
```

USAGE **open**, as the name implies, opens a file. Until a file is opened, a C program cannot have access to the data contained in it. **open** is used to initiate what are commonly called *Level* 1 I/O or *primitive* I/O operations. Primitive disk operations access the disk directly, without automatically buffering blocks of data in memory.

open and its counterpart **close** are called file *primitives* because they serve as the building blocks for the creation of more complex and sophisticated buffered (*Level 2*) functions like **fopen.**

The major difference between **open** and **fopen** is that each time a file which has been opened with **open** (sorry about having to confuse the function name with the verb) is accessed, it will involve a disk operation. Files which are accessed with *Level* 2 I/O take advantage of available memory to reduce the number of read/write disk operations.

In the above sample:

`int file;`
declares a variable to hold the file number or error code.

`char *filename;`
declares a pointer to the filename.

`int mode;`
declares a variable to hold the mode number.

`file = open(filename, mode);`
opens the named file for a particular mode of operations. The returned value from the function will be assigned to **file.**

The following values (given in both decimal and hexadecimal) may be used for mode:

Decimal	Hexadecimal	Meaning
0	0x0	Read only, with text-file translation.
1	0x1	Write, with text-file translation.
2	0x2	Read/write, with text-file translation.
-32768	0x8000	Read only, without text-file translation.
-32767	0x8001	Write, without text-file translation.
-32766	0x8002	Read/Write, without text-file translation.

Text-file translation refers to the replacement of newlines with carriage return/linefeed sequences (0D0Ah) on "writes" and vice-versa on "reads".

A returned value of 0 or greater means success. An unsuccessful effort will result in an EOF being returned.

The file position is set to zero if the file is opened successfully. If the attempt fails, -1 is returned. Closing the file releases the file number for use with another file.

NOTES

It is rarely necessary to resort to unbuffered file I/O since the **fopen** family of buffered I/O functions is available. The file primitives **open, close** and **creat** and the other *Level* 1 functions are included in RUN/C for those who wish to experiment with direct I/O and/or who are interested in creating their own buffered file I/O routines (i.e., writing a replacement for **fopen**) or to have control over buffer size and thus frequency of disk access on reads and writes.

RETURNS

-1, if an error is detected.
file number, if successful.

EXAMPLE

```
main()

  /* close.c -- illustrates use of close() */

{char name[20];
char buf[1];
int fin;
int i;

     printf("Name please: ");
     gets(name);
     if ((fin = open(name,0)) == -1)
     {     printf("No such file: %s\n",name);
           exit();
     }

     while (0 != (i = read(fin, buf, 1)))
           {
           if (i <0){
                 printf("error on read\n");
                 close(fin);
                 exit();
                 }
           else putchar(buf[0]);
           }

     close(fin);
}
```

SEE ALSO fopen, close, creat, lseek, read, write, unlink

PURPOSE Sends a value to a port.

FORM
```
unsigned p;
int v;
outp(p,v);
```

USAGE **outp** sends a numeric value to the specified I/O port.

In the above sample:

`unsigned p;`
declares a variable to hold the port number.

`int v;`
declares a variable to hold the value to be sent to the port.

`outp(p,v);`
sends the value assigned to **v** to the port number assigned to **p.**

Port numbers range from 0 to 65535; values from 0 to 255 may be sent to a port.

outp is the complement of **inp.**

NOTES
* Many critical elements involved with a PC can be accessed through port numbers. In addition to erasing and otherwise damaging data, it is possible to physically damage a system by outputting random values to casually selected ports.

* Port assignments vary from computer to computer, even in situations which claim complete compatibility. See the technical documentation for your computer for information on valid port numbers (I/O addresses) and the values which are relevant to them.

RETURNS Nothing.

(Continued next page)

EXAMPLE

```
main()      /* outp.c -- illustrates outp() */
{
int i,j,k;
j = 0;
     puts("A few notes from our electronic
     clarinet");
     outp(0x43,0xB7);
     for(i = 0; i < 100; ++i, ++j){
     outp(0x42,j);
     outp(0x61,0x4f);
     }

j = 100;
     puts("I yearn for the high notes");
     for(k = 100; k > 1; --k, --j){
          outp(0x42,j);
          outp(0x61,0x4f);
     }

     outp(0x61,0x4d);   /*  disables  speaker;
     otherwise noise continues */
}
```

SEE ALSO inp, poke

PURPOSE Reads a byte in a location in memory.

FORM

```
long address;
int data;

data = peek(addr);
```

USAGE **peek** looks at an absolute address in memory and returns its contents. The values are read in 8-bit bytes.

In the above sample:

`long addr;`
declares a variable to hold the target memory address.

`int data;`
declares a variable to hold the value currently held in that memory location.

`data = peek(addr);`
reads the contents of memory at **addr**, assigning the returned value to **data.**

NOTES

* The data returned value will be in the range 0 to 255.
* Be sure to use an **L** (denoting **long**) after the value assigned to the memory address element if it is a constant.

RETURNS Contents of memory location, if successful.

(Continued next page)

EXAMPLE

```
main()
{
/* peek.c -- illustrates use of peek (a la a
dump) */
int i, j, v;
long a, a2;

cls();
puts("Hex/ASCII dump of 256 bytes of memory");
printf("Absolute decimal address? ");
scanf("%ld",&a);
for (i = 0; i < 256; i += 16, a += 16) {
     printf("%05lx: ",a);
     a2 = a;
     for (j = 0; j < 16; j++, a2++)
          printf("%02x ",peek(a2));
     a2 = a;
     for (j = 0; j < 16; j++, a2++) {
          v = peek(a2);
          if ( (v < 32) || (v > 127) )
               printf(".");
          else
               printf("%c",v);
          }
     printf("\n");
     }
}
```

SEE ALSO poke, inp

PURPOSE Writes a byte into a location in memory.

FORM

```
long address;
int data;

poke(address, data);
```

USAGE **poke** sends a numeric value in the range of 0-255 to an absolute memory address in the system.

In the above sample:

`long address;`
declares a variable to hold the target memory address.

`int data;`
declares a variable to hold the integer number being sent to the memory address.

`poke(address,data);`
puts the value assigned to **data** into the specified address in memory.

NOTES

* The example given below will only work on the IBM PC and close compatibles. The significance of particular addresses may vary from computer to computer, even in situations which claim complete PC compatibility. See the technical documentation for your computer for information on key memory addresses and the values which they are designed to hold.

* When you send a value to a memory location using **poke**, you are changing the contents of memory. With **poke** you can change any memory location in your system -- and quite possibly crash the system in the process. Critical elements of the PC -- the screen, program memory, system memory, etc. -- can be accessed via **poke**. RUN/C does not check the address you are using for safety or correctness. We quote from the IBM BASIC manual: "Don't go POKEing around" where you don't belong.

* Be sure to use an **L** (denoting **long**) after the value given for the memory address if it is a constant.

* RUN/C's poke is special in that it allows you to write to any address in memory. This is done through automatic conversion from 64k record blocks.

RETURNS Nothing.

EXAMPLE

```
main() /* poke.c -- demonstrates use of poke */
            /* monochrome screen attributes*/
{
     int i;
/* color use 753665L; monochrome use 720897L */
     int attribute_byte;
     int c;
     long a=720897L;
     cls();
     puts("\n\nSelect your mode:");
     puts("\t1 Blinking");
     puts("\t2 Blinking and highlighted");
     puts("\t3 Blinking and underlined");
     puts("\t4 Blinking and reversed");
     puts("\t5 Reversed video");
     puts("\t6 Underlined");
     puts("\t7 Highlighted");
     puts("\t8 Highlighted and underlined");
     puts("\tEsc to exit gracefully");
```

(Continued next page)

```
        if(( c = getchar()) != EOF)
              switch(c)
              {
                    case '1': attribute_byte = 212;
                          break;
                    case '2': attribute_byte = 222;
                          break;
                    case '3': attribute_byte = 129;
                          break;
                    case '4': attribute_byte = 240;
                          break;
                    case '5': attribute_byte = 112;
                          break;
                    case '6': attribute_byte = 1;
                          break;
                    case '7': attribute_byte = 95;
                          break;
                    case '8': attribute_byte = 121;
                          break;
                    case 27: exit();
                    default:
                          puts("\nNot an acceptable
                          answer. Run me again.");
                                exit();
              }

        cls();
        puts("Why don't word processing programs
        use these features?");
        for (i = 0; i < 53; i++, a += 2)
              poke(a, attribute_byte);
  }
```

SEE ALSO peek, outp

PURPOSE Calculates the power of a number.

FORM
```
double result;
double number;
double n;

result = pow(n,number);
```

USAGE **pow** computes the power of a number. The function raises a real number (possibly assigned to a variable) to the power of a second number. Both numbers should be double precision.

NOTES In RUN/C Professional this and other transcendental math functions are available in the loadable library MATHLIB.

RETURNS result of raising one number to the power of another, if second number is positive;
0, if second number is negative or non-integral.

EXAMPLE
```
main()      /* pow.c -- illustrates pow() */
{
     double a,b,c;

     a = 2;
     b = 5;
     c = pow(a,b);
     printf("%g\n",c);
     printf("Correct answer is 32.000000\n");

/* Result: pow yields a to the b */
}
```

PURPOSE Displays the C operator precedence table.

FORM PREC

USAGE PREC displays table of precedence given to the various C operators by Kernighan and Ritchie in The C Programming Language. This command is useful both in program creation and debugging mode, particularly if a C reference manual is not immediately available.

NOTES The precedence table is listed at the end of Section 3.

PURPOSE Prints formatted data to the standard output device (usually the screen).

FORMS

[1]
```
printf("string\n");
```

[2]
```
int arg1;
char arg2;
char arg3[80];

printf("format %d %c %s\n", arg1, arg2,
arg3);
```

[3]
```
int arg1;
int arg2;

printf("total: %d", arg1 + arg2);
```

USAGE **printf**, and its cousins **fprintf**, **lprintf**, and **sprintf**, provides C with sophisticated text editing capabilities. **printf** allows you to easily incorporate the results of program calculations into printed output.

Form [1] simply prints the character string "string" to the terminal screen. The *backslash*-n (**\n**) indicates a *newline* is to follow the printed string.

Form [2] provides for the integration of various constant and calculated values into the text which is then printed. In Form [2]:

`int arg1;`
declares an integer variable named **arg1**.

`char arg2;`
declares a character variable named **arg2**.

`char arg3[80];`
declares an 80-character array named **args**.

`printf("format %d %c %s\n", arg1, arg2, arg3);`
prints a string of characters beginning with "format" followed by the decimal value of **arg1**, the character value of **arg2** and the string which has been loaded into the **arg3** array.

Form [3] prints a value which is derived from calculations performed inside the **printf** function:

`int arg 1;`
declares an integer variable named **arg1.**

`int arg 2;`
declared an integer variable named **arg2.**

`printf("total: %d", arg1 + arg2);`
prints a string of characters beginning with "total:" followed by the decimal value sum of **arg1 + arg2.**

This last example hints at the power of **printf;** complex manipulation of data developed by other functions can be performed. See example 5, below.

Another example
Suppose you wish to have your program calculate the average number of students in each class in a college at any given hour. There may be thousands of lines of data and many lines of code necessary to calculate the answer, but the results can be stated in a single sentence:

At _ _.m. there were an average of _ students in each classroom.

The blanks to be filled in specify the hour, a.m. or p.m., and the number of students.

printf allows you to write a single line of code which does all the work. This is done by using several formatting codes based around the structure **%a** where the **%** signals that a special character is coming. The character which follows the **%** tells **printf** what to do with the data being passed by variables into the program.

There are nine variable types, also called *conversion characters,* available to **printf.** The choice of conversion characters determines the form the data is printed in.

The conversion options are:

Option	Datatype
%c	single character
%d	signed decimal
%e	exponential floating point
%f	fixed decimal floating point
%g	e of f format, whichever is shorter
%o	octal
%s	string
%u	unsigned decimal
%x	hexadecimal
%%	prints % (percent sign itself)

The format type must be specified in lower case.

Taking these one at a time:

%c prints the character value, for example:

```
main()
{
        char a;
        a = 65;
        printf("%c", a);
}
```

prints:

```
A
```

%d Prints the signed decimal or integer value of an expression, for example:

```
main()
{
        char a;
        a = 65;
        printf("%d", a);
}
```

prints:

```
65
```

(The maximum signed value is 32,767.)

%o prints the octal value of an expression. For example:

```
main()
{
        char a;
        a = 65;
        printf("%o", a);
}
```

prints:

```
101
```

%s prints the value of an expression as a pointer to a string and prints the string. For example:

```
main()
{
        char *p;
        p = "This string is mine.";
        printf("%s",p);
}
```

prints:

```
This string is mine.
```

%u prints an unsigned decimal character. For example:

```
main()
{
        int a;
        unsigned b;
        a = 40000;
        b = 40000;
        printf("%d %u", a, b);
}
```

prints:

```
-25536, 40000
```

(The maximum unsigned value is 65,535.)

%x Prints a hexadecimal value of an expression. For example:

```
main()
{
        int a;
        a = 65;
        printf("%x", a);
}
```

prints:

```
41
```

%e Prints the value of an expression in exponential floating point or scientific notation, in the form *i.ffffffEeee*. The precision field (described below) specifies the number of decimal places in the output, the default being 6. For example:

```
main()
{
        float n;
        n = 3333.0 / 6666;
        printf("%e", n);
}
```

prints:

```
5.000000E-01
```

%f Prints a fixed decimal floating point value which is output without an exponent field in the form *i.f*. The precision field (described below) specifies the number of places after the decimal point. The default is 6 places. A precision of zero suppresses the decimal point. For example:

```
main()
{
        float n;
        n = 3333.0 / 6666;
        printf("%f", n);
}
```

prints:

```
0.500000
```

%g uses either the e or f format, whichever is shorter. For example:

```
main()
{
        float n;
        n = 3333.0 / 6666;
        printf("%e %f %g", n, n, n);
}
```

prints:

```
5.000000E-01      0.500000     0.500000
```

%% Prints "%". For example:

```
main()
{
        char *s;
        s = "This is your string.";
        printf("%% %s", s);
}
```

prints:

```
% This is your string.
```

Precision field

To make things somewhat more complicated, quite a few additional specifications can be placed between the % and the conversion character. This is the *precision field*, mentioned above, and it provides further control of printed output.

The form for this is often expressed as:

% - # . # *l a*

(In actual usage there should be no *white space* within the precision field; it is included here for the convenience of the reader.)

Taking these options one at a time:

- The minus sign (-) indicates that the variable or string will occupy the left side of the space allocated to it, rather than the default condition

which is right justified (flush right). This option is considered "left justified" since spaces will be added on the right to fill in the allotted space. See the example below.

The first number (symbolized by *#*) tells how many spaces to allocate for the variable in question -- the minimum field width. For example, **%20s** allocates 20 spaces for a string of characters being passed to it from the program. The field width cannot be used to truncate a string or constant passed by **printf**. Even if the number of characters used by **printf** exceeds the number of spaces allocated in the precision field, the entire string or constant will be printed.

%-20s also allocates 20 spaces, but the padding will be on the right instead of the left. For example:

```
main()
{
        char *n;
        n = "This is a string.";
        printf("%20s\n %-20s", n,n);
        printf("*%24s*%-24s", n,n);
}
```

prints:

```
This is a string.
 This is a string.
*          This is a string.*This is a string.
```

.# The second number, which must be preceded by a period, limits the number of characters which can be passed from a string. It is also used to specify the number of digits to be printed to the right of the decimal point of a **float** or **double** variable. For example:

```
main()
{
        char *a;
        float f;
        a = "2501";
        f = 18.0832;
        printf("%.2s    %.2f", a, f);
}
```

prints:

```
25    18.08
```

l The letter **l** (el) specifies that the data item is a **long** integer (**%ld**) or **double** (**%lf**). Another example:

```
main()
{
        long a;
        long b;
        a = 232501L;
        b = 180832L;
        printf("%ld", a + b);
}
```

prints:

```
413333
```

In formatting text it is sometimes necessary to be able to give special instructions to the printer or the screen such as: 'begin a new line', 'return the carriage (or cursor or printer mechanism) to the leftmost position', 'insert horizontal tabs', 'do a page break or form feed', 'backspace', etc.

These formatting codes -- often called *escape sequences* -- can be anywhere in a **printf*** statement where text can appear. In all cases the escape sequences are preceded by a backslash:

* Escape formatting codes work equally well in simpler output functions such as **puts** and **fputs**, as well as with **fprintf**, **lprintf** and **sprintf**.

Escape code	Meaning
\n	newline
\t	horizontal tab
\b	backspace
\r	carriage return
\f	formfeed
\0	zero character
nnn	octal bit pattern (positive or negative)
****	print as a backslash

The octal bit pattern is useful for placing special characters in a formatted string. For example, to place a double quotation mark within a **printf** string constant one might use:

```
printf("Putting an \042 in is easy!\n");
```

which will output:

```
Putting an " in is easy!
```

NOTES

* The number of arguments supplied must be at least equal to the number of conversions in the format string or unexpected results are likely to occur. Excess arguments are ignored.

* **printf** can be used to convert arguments into the specified conversion type, but there is no guarantee of success. Here are some common conversions and their likely result:

Conversion	Result
int to char	yes, if **int** is less than 256
char to int	yes
float to char	no
char to float	no

* In Form [1] above, a better choice might have been the **puts** function. It's entire job is to output a string constant followed by an automatically supplied *newline*.

RETURNS Nothing.

EXAMPLES [1] Effect of various format strings on the integer 123:

Format string	Result
%d	123
%04d	0123
%-02d	123
%o	173
%x	7B
%u	123
%-12d	123_____*
%12d	_____123

[2] Effects of various formatting commands on the floating point number 12.34:

Format string	Result
%f	12.340000
%.1f	12.3
%06.1f	0012.3
%-6.1f	12.3

[3] Effects of various formatting commands on this 16-character string:

ABCDEFGHIJKLMNOP

format string	Result
%s	ABCDEFGHIJKLMNOP
%12s	ABCDEFGHIJKLMNOP
%-12s	ABCDEFGHIJKLMNOP
%.12s	ABCDEFGHIJKL
%20s	ABCDEFGHIJKLMNOP
%20.12s	_____ABCDEFGHIJKL
%-20.12s	ABCDEFGHIJKL_____

(Continued next page)

* Underline represents padding blanks or spaces.

[4] Routine to convert decimal numbers to hexadecimal and octal numbers:

```
main()      /* printf.c -- illustrates a bit
               of the power of printf() */
{
int a;
printf("Type a character to see its
decimal, octal and hex values\n");
a = getchar();
printf("\n%c equals %d in decimal, %o in
octal and %x in hexadecimal.\n", a, a, a,
a);
}
```

[5] Demonstrates the inclusion of another function within a **printf** statement:

```
main()
{
char c;

while(1)
      printf("%c ", c = (getchar() != 13));

/* Result: any key pressed is echoed to the
screen. */
}
```

SEE ALSO fprintf, sprintf, lprintf, scanf, puts

PURPOSE Turns the profiler (**PRON**) off.

FORM **PROFF**

USAGE **PROFF** turns off the profiler.

NOTES In RUN/C Professional, the profiler can be turned off from within a program using the special **runc()** function as in:

```
runc("PROFF");
```

EXAMPLE

```
Ok

PROFF
Ok
```

SEE ALSO **PRON**

PURPOSE Activates a profiler which reports the number of times each line in your program was interpreted during a RUN. The profiler report appears when the program is subsequently listed.

FORM PRON

USAGE PRON, standing for PRofiler ON, turns on the profiler which in conjunction with **LIST** (or **NLIST, LLIST** or **NLLIST**) reports the number of times a line has been executed during a program run. **PRON** is deactivated with **PROFF.**

NOTES

* Profile information is cleared and re-collected with each program RUN, so that you need not issue the **PRON** command before running your program. The sequence:

```
RUN
 .
 .
 .
PRON

Ok
LIST

Ok
```

will show correct data unless a **NEW** has been issued, clearing memory of the program.

You can, however, get inaccurate profile data by running one program, loading another and then issuing PRON and LIST commands.

* In RUN/C Professional, the profiler can be turned off from within a program using the special **runc()** function as in:

```
runc("PRON");
```

(Continued next page)

EXAMPLE LIST

```
1:/* counts up one variable and down another */
2:
3:main()
4:{
5:   int i,a;
6:   i = 1;
7:   a = 10;
8:
9:   do
10:        printf("i = %d; a = %d\n", i++, a--);
11:  while (i <= 5);
12:}

Ok
PRON

Ok
RUN

i = 1; a = 10
i = 2; a = 9
i = 3; a = 8
i = 4; a = 7
i = 5; a = 6

Ok
LIST

1:<1>/* counts up one variable and down another
*/
2:<1>
3:<1>main()
4:<1>{
5:<1>int i,a;
6:<1>i = 1;
7:<1>a = 10;
8:<1>
9:<1>do
10:<6> printf("i = %d; a = %d\n", i++, a--);
11:<6>while (i <= 5);
12:<1>}
```

```
Ok
PROFF
Ok
```

In the above example, the number of times each line is accessed by the program is shown by a number enclosed in angle brackets.

SEE ALSO PROFF, TRON, TRACE

PURPOSE Send a character to the specified file.

FORM
```
int ret;
char c;
FILE *fp;

ret = putc(c, fp);
```

USAGE **putc** outputs a character to the indicated disk file.

In the above sample:

`int ret;`
declares the variable **ret** to contain returned value.

`char c;`
declares a variable to hold the character to be sent to the file.

`FILE *fp;`
declares the file pointer.

`ret = putc(c, fp);`
puts the character **c** into the file pointed to by **fp**. The returned value will be assigned to **ret**.

The returned value will hold the character (an "echo" effect), if successful, or -1, if an error has occurred.

NOTES In RUN/C there is no difference between **putc** and **fputc**. However some compilers implement **putc** as a macro, so beware of side effects when compiling programs using this function.

RETURNS -1, if an error is detected.
Character, if successful.

(Continued next page)

EXAMPLE

```
main() /* putc.c -- illustrates use of putc() */
{
     FILE *fi, *fo;
     int i;
     char c, dest[25];
     char ibuf[255];

     printf("Output filename? ");
     gets(dest);
     fo = fopen(dest, "w");
     puts("Type something:");
     gets(ibuf);
     puts(ibuf);
     for (i = 0; ibuf[i] != 0; i++)
          putc(ibuf[i], fo);
     fclose(fo);
     puts("Done");
}
```

SEE ALSO fputs, putchar, printf

PURPOSE Sends a character directly to the console.

FORM

```
char c;

putch(c);
```

USAGE **putch** sends a character to the screen.

In the above sample:

`char c;`
declares a variable to hold the value of the character to be sent to the screen.

`putch(c);`
sends that character to the screen.

NOTES The primary difference between **putch** and **putchar** is that the former will send characters to the console screen even if other input has been redirected to another output device such as the printer or a communication port.

RETURNS Nothing.

EXAMPLE See example under **getch.**

SEE ALSO putchar, printf

PURPOSE Sends a character to standard output device (usually the screen).

FORM

```
int r;
char c;

r = putchar(c);
```

USAGE **putchar** sends a character to the screen or currently assigned output device (**stdout**).

In the above sample:

`int ret;`
declares an integer variable.

`char c;`
declares a character variable.

`ret = putchar(c);`
places the character in **c** to the currently active output device. The returned value from the function will be assigned to **ret**.

NOTES Some compilers implement **putchar** as a macro, so beware of side effects when compiling programs using this function.

RETURNS -1, if error.
Character, if successful.

EXAMPLE

```
main()      /* putchar.c -- illustrates simple use
               of putchar() */

{
     puts("Type a character and get it back.");
     char i;
     i=getchar();
     printf("\n\n");
     putchar(i);
}
```

SEE ALSO getchar, putc, putchar, printf, stdout

PURPOSE Outputs a string to the standard output device (usually the screen).

FORM

```
int r;
char *s;
s = "my words";
r = puts(s);
```

USAGE **puts** sends a string of characters to the standard output device (**stdout**), generally the video screen. **puts** automatically outputs a terminating newline (**\n**) after sending all the characters in the string.

In the above sample:

`int ret;`
declares an integer to hold the returned value.

`char *s;`
declares a pointer to the string.

`s = "my words";`
assigns the string constant "my words" to the address in memory pointed to by the **s** pointer.

`ret = puts(s);`
sends string **s** to the screen. The returned value from the function will be assigned to **ret**.

RETURNS -1, if an error or end-of-file is detected.
0, if successful.

(Continued next page)

EXAMPLE

```
main()

/* puts.c -- illustrates simple use of puts() */
{
     char x[80];
     puts("Type a string up to 80 characters and
     get it back.");
     gets(x);
     printf("\n\n");
     puts(x);
}
```

SEE ALSO fputs, gets, fprintf, stdout

PURPOSE Generates a random number.

FORM

```
int result;

result = rand();
```

USAGE **rand** returns a pseudo random integer in the range 0 to 32,767.

NOTES In RUN/C Professional this and other transcendental math functions are available in the loadable library MATHLIB.

RETURNS Random number.

EXAMPLE

```
main()      /* rand.c -- obtains random positive
               integers from rand() */

{
     int i, n;

     for (i = 0; i < 5; i++) {
          n = rand();
          printf("%d ",n);
     }
     puts("\nThere are no correct answers.");
}
```

SEE ALSO srand

PURPOSE Reads data from a file.

FORM

```
int ret;
int file;
char *buffer;
int length;

return = read(file, buffer, length);
```

USAGE **read** is a *Level 1* (unbuffered) I/O function, upon which many other functions (such as **fread** are based). **read** is used to access disk data and data in other devices. The function reads the next group of bytes from a file. After the operation is performed, the file's character position pointer is advanced the number of bytes read. **read** can be used only with a file which has been opened with **open** or **creat.**

In the above sample:

`int ret;`
declares a variable to hold the returned code.

`int file;`
declares an integer to hold the file number (obtained with **open**).

`char *buffer;`
declares a pointer to an input buffer which will hold the data read.

`int n;`
declares an integer to hold the number of bytes to be read. This must be a positive number.

`ret = read(file, buffer, n);`
reads **n** bytes from **file**, placing the characters read in **buffer.** The returned value from the function will be assigned to **ret.**

The return count is always equal to the actual number of bytes placed in the buffer and will not vary from the **n** parameter unless an EOF or other error is encountered, in which case a 0 or -1 is returned.

NOTES * The number of bytes by which the file position is advanced may not equal the number of bytes transferred if *text mode translation* is active.

RETURNS -1, if error is detected.
0, if end-of-file.
number of bytes read, if successful.

EXAMPLE

```
main()

/* read.c -- reads a file and echoes it on
screen */

{char name[20];
char buf[1];
int fin;
int i;

     printf("Name please: ");
     gets(name);
     if ((fin = open(name,0)) == -1)
     {     printf("No such file: %s\n",name);
           exit();
     }

     while (0 != (i = read(fin, buf, 1)))
           {
           if (i <0){
                 printf("error on read\n");
                 close(fin);
                 exit();
                 }
           else putchar(buf[0]);
           }

     close(fin);
}
```

SEE ALSO open, write

PURPOSE Allows the declaration of a variable as a **register** variable.

FORM

```
register int x, y;
```

USAGE **register** in a compiled C environment means the compiler will, where possible, use machine registers for storage and manipulation of automatic variables. In some environments use of **register** variables provides distinct speed advantages over other forms of variable storage. However in the Intel 8086/8088 environment the number of hardware registers is so limited that many compilers have chosen not to support **register** variables.

It is important to note that in RUN/C the **register** keyword is merely accepted by the interpreter. Hardware registers are not at present used for data manipulation and no speed advantage will result from use of this keyword. It is provided for the convenience of those using C compilers which support **register** variables.

NOTES

* The **register** declaration may only be used inside functions; never globally.

* Similarly to **register**, the storage class identifier **extern** is permitted in interpreted programs, but has no direct effect on the running program.

EXAMPLE

```
main()

{
register int c = 4;
     printf("register int c = %d", c);
}
```

PURPOSE Renames a disk file.

FORM

```
int ret;
char *old_name;
char *new_name;

ret = rename(old_name, new_name);
```

USAGE rename changes the name of a disk file after making certain that the "old name" file exists and that there is no file with the same name as the "new name" on the same drive/directory.

In the above sample:

`int ret;`
declares a variable to hold the returned code.

`char *old_name;`
declares a character pointer to the file name assigned to **old_name.**

`char *new_name;`
declares a character pointer to the file name assigned to **new_name.**

`ret = rename(old_name, new_name);`
renames the file attached to the first argument to the name assigned to the second argument. The returned value from the function is assigned to **ret.**

If a 0 value is returned, the operation has been successful. Otherwise an error has occurred. An error code will be returned -- and no name change will take place -- if the new name chosen already exists or if the old file cannot be located.

Standard naming conventions for filenames, disk and directory pathnames, as described in Section 1, apply. If no drive or directory is specified, the current default drive/directory will be assumed.

NOTES If a drive/directory pathname is used, it must be for both the old and new file names.

RETURNS Non-zero, if an error is detected.
0, if successful.

EXAMPLE

```
main()

/* rename.c -- renames hello.c to hiya.c */

{
int error;
char *old;
char *new;

     old = "hello.c";
     new = "hiya.c";

     error =  rename(old,new);
     if(error != 0)
          puts("Error detected; new file name
          may exist or old name not.");
}
```

SEE ALSO SAVE, KILL, UNLINK

PURPOSE Terminates the execution of the current function, returning control to the calling function, and, if an expression is present, passing its value back to the calling function.

FORM
```
[1] return;

[2] return(expression);
```

USAGE **return** causes program control to return to the calling function. If an expression is carried by the **return** statement, it is returned as well. It is up to the function to which control is returned to determine how returned expressions or values are used, if at all.

Calculations can be performed within **return.** To give a simple example:

```
return(3+2);
```

will return a value of 5 to the calling function.

NOTES
* If an expression is returned, it should be the same data type as is called for in the calling function. For example, if the calling function expects a **double** to be returned, the value should be returned to the function as a **double.**
* The parentheses around the expression are generally optional.
* Unless an expression is specified as an argument, any value returned by the function should either be considered 'garbage' and/or a suitable value for a lottery entry.
* If the value to be returned does not match the returned-data type with which the function was defined (default is **int**), the value will be coerced to that type. (And the result may not be what you wanted.) See the example for a illustration of the use of casting to prevent this type of problem.

EXAMPLE

```
main()
{    /* return.c -- converts a value in test(),
     illustrating return */
int a;
int b;
a = 27;
b = 3;
a = test(a,b);
printf("But in main() a is equal to %d and b to
%d.\n", a, b);
puts("Correct second line: But in main() a is
equal to 9 and b to 3.");
}

test(a,b)
int a;
int b;

{
a = a / 3;
b = b * 3;
printf("In test() function a = %d; b = %d.\n",
a, b);
return(a);
puts("This should not print");
}
```

SEE ALSO void

PURPOSE Resets a file position pointer to the first byte of the file.

FORM
```
FILE *fp;
rewind(fp);
```

USAGE **rewind**, a Level 2 (buffered) I/O function resets the file position pointer of the specified file to the beginning of the file. This function is often used in conjunction with **fgetc** and **fputc** which read or write bytes from a file, leaving the file's byte pointer where the **read** or **write** ended.

In the above sample:

`FILE *fp;`
declares a pointer to file **fp.**

`rewind(fp);`
resets **fp**'s file pointer.

RETURNS Nothing.

(Continued next page)

EXAMPLE

```
main()      /* rewind.c -- demos rewind(); requires
               a text file on disk */

{
     FILE *fi;
     char c, source[25];
     int i;

     printf("Name of file to read? ");
     gets(source);
     if ((fi = fopen(source,"r")) == NULL) {
          printf("Can't open %s\n",source);
          exit();
     }
     puts("It begins:");
     for (i = 0; i < 128; i++) {
          c = getc(fi);
          putchar(c);
     }
     puts("\nDa capo:"); /* this may be Latin,
     but it's all Greek to me. */
     rewind(fi);
     for (i = 0; i < 128; i++) {
          c = getc(fi);
          putchar(c);
     }
     fclose(fi);
     puts("\nCorrect answer is the same thing
     twice.");
}
```

SEE ALSO fopen, fseek

PURPOSE Executes the program in memory.

FORM

```
[1] RUN

[2] RUN send 5 words to main

[3] A>RC MYPROG

[4] RUN > CAPTURE

[5] RUN < KEYSTROKE
```

USAGE **RUN** executes -- or at least attempts to execute -- the program currently in memory.

If the program name is invoked when RUN/C is first loaded, control will return to DOS after the program has successfully **RUN.** If an error or <Ctrl-c> interrupts execution of the program, control will revert to the RUN/C Command level, rather than DOS.

Arguments may be passed to the **main()** function through **RUN.** Form [2], which simply passes five character strings to **main(),** illustrates this technique, as does the example, below.

Input and output can be redirected from the **RUN** command line in two ways. Program output -- the stuff you normally see on the screen when your program runs -- can be sent to a file using the > (greater than) symbol followed by the name of the DOS file you wish created. Program input -- the stuff you normally type in response to prompts from your C program -- can be obtained from a previously created disk file using the < (less than) symbol followed by the name of the file which contains the specified data.

For example:

```
RUN FTOC > CAPTURE
```

will place the output from the FTOC.C program into a file named CAPTURE on the default drive/directory.

Input to a program can be obtained by a command line similar to:

```
RUN < DATA
```

which will supply the items in data to the program as if they had been typed on the command line.

NOTES

* Redirected I/O is independent of any arguments to **main()**.
* Redirected I/O should not be attempted when RUN/C and a C program are invoked together from the DOS command line.
* Program **LOAD** and **RUN** operations can be combined using the **EXEC** command.

EXAMPLE [1]

```
Ok

NLIST

main()

{
        printf("Hello, world!\n");
}
RUN

Hello, world!
Ok
```

[2]

```
/* main.c -- prints argument count, arguments
from RUN line */

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

        printf("argc = %d\n",argc);
        for (i = 0; i < argc; i++)
                printf("argv[%d]: |%s|\n",i,argv[i]);

/* Result: each argument passed to main is
counted and listed within vertical brackets */

}
```

SEE ALSO LOAD, EXEC, main()

Built-in debugging function

PURPOSE Permits execution of RUN/C commands from within interpreted programs.

FORM

```
runc("BREAK");
```

USAGE **run()** is a special debugging function which can be used in conjunction with any C program being interpreted under RUN/C Professional. Its use is fully described in Section 6 of the Professional's manual; portions of that discussion will not be repeated here.

In the above sample:

```
runc("BREAK");
```

is the equivilent of placing a breakpoint in the program which, in turn, is the equivalent of pressing <Ctrl-c>. Like other functions, execution of **runc()** can be triggered by various software events such as:

```
if(i < 0)
      runc("SAVE MYFILE");
```

All RUN/C commands are available to **runc()** except:

*	RUN	*	NEW
*	LOAD	*	SYSTEM
*	EDIT or E	*	EDITOR
*	HELP		

NOTES * You can compile programs that use the **runc()** function in several ways. One way is to define a dummy function such as:

```
runc(cp)
char *cp;
{ }
```

which will negate any effects of the function call.

EXAMPLE See above.

SEE ALSO BREAK, Section 6

PURPOSE Saves a program in memory to the specified drive/directory.

FORM [1] SAVE *filename*

[2] SAVE drive: *directory**filename*

[3] SAVE

USAGE SAVE writes the program currently in memory to a file on the default drive/directory or the specified drive/directory. The program remains resident in memory following the SAVE. While the file is being saved the message:

Saving *program.nam*

will be displayed.

When a file is saved the .C extension is assumed unless another extension is supplied. See LOAD for a full discussion of RUN/C file naming conventions.

If SAVE is used without a filename (Form [3]), the program currently in memory will be saved to the default file name PROGRAM.C.

Use of quotation marks around the file name is optional. But a space after SAVE is mandatory. This also holds true for LOAD, KILL and UNLINK.

NOTES

* If there is currently a file of the same name on the drive/directory as the name you have selected, the older file will be renamed with an extension ".BAK". If a ".BAK" file with the same primary name exists, it will be deleted.

* RUN/C saves programs at a rate of approximately 150 lines per second. Thus when saving larger program a slight delay may occur, particularly if a .BAK file must be deleted and re-created.

EXAMPLES

	Command	Effect
[1]	**SAVE TEST**	Program TEST.C is saved to the default drive/directory.
[2]	**SAVE B:TST.C**	Program TEST.C is saved to the current directory on the "B:" drive.
[3]	**SAVE "B:\CPROGS\TEST.C**	Program TEST.C is saved to the pathname \CPROGS and the filename TEST.C.
[4]	**SAVE**	Saves program in memory to the filename PROGRAM.C.

SEE ALSO AUTOSAVE, LOAD

PURPOSE Captures data input via the standard input device (usually the console).

FORM

```
int ret;
int arg1;
char arg2;
char arg3[80];
ret = scanf("%d %c %s", &arg1, arg2, arg3);
```

USAGE **scanf**, like its cousins **fscanf** and **sscanf**, use a syntax similar to **printf** to read input data. It performs formatted input conversions on text obtained from the keyboard or other standard input devices. The **scanf** control string may contain format specifiers and/or characters to be matched from the input; the list of pointer arguments specifies where the results of the conversions are to go.

In the above sample:

`int ret;`
declares a variable to hold the returned value.

`int arg1;`
declares integer variable **arg1**.

`char *arg2;`
declares a character pointer variable **arg2**.

`char arg3[80];`
declares an 80-character array named **arg3**.

`ret = scanf("%d%c%s", &arg1, arg2, arg3);`
waits for keyboard input in the form of one decimal number, one character and one string of characters, and places the data in **arg1**, ***arg2** and **arg3[0]** to **arg3[n]** respectively. The returned value is assigned to **ret**.

An example
Let us consider a programmer whose program is designed to obtain information from a computer operator. S/he needs a function similar to **printf** which can be used to obtain the data necessary to fill in a "blank".

Our hypothetical programmer who you may remember from our exposition on **printf** now wants his/her program to calculate the average number of students in each class in a college at any given hour, but it is the operator who is to specify for which hour the calculations should be made. The operator will see the following statement:

```
Which hour would you like me to calculate?
__ Type your answer using a 24-hour clock.
```

scanf will allow you to write a single line of code to handle the operator's response.

As with **printf, scanf**'s formatting codes are based on a character string enclosed in double quotes in which conversion characters control the types of data assigned to data spaces pointed to by arguments. The conversion character format **%a** found in **printf** is also used in **scanf.** The **%** indicates that a special character is coming. The character which follows the **%** tells **scanf** how to handle the data being passed from the operator.

The data type options and their meanings are:

Option	Data type
%c	single character
%d	signed decimal
%f, %e, %E	decimal floating point
%o	octal
%s	string
%u	unsigned decimal
%x	hexadecimal
%%	percent sign expected

Taking these data conversion options individually:

%c captures the character value of the first character typed from the keyboard. For example:

```
main()
{
      printf("Type any character:\n")
      char a;
      scanf("%c", &a);
      printf("%c", a);
}
```

obtains the next character typed from the keyboard and prints the character captured. If you type 3, the character 3 will be assigned to **a** and printed.

%d captures the value of a decimal number typed from the keyboard. For example:

```
main()
{
      int a;
      printf("Type any integer\n");
      scanf("%d", &a);
      printf("%d", a);
}
```

obtains the decimal value of the next integer number. If you type 45, 45 will be assigned to a and printed.

%o captures the value of a string of octal digits. For example:

```
main()
{
      int a;
      scanf("%o", &a);
      printf("%o", a);
}
```

obtains the value of the next valid sequence of octal digits. For example, if you type 27, the value 23 (23 equals 27 octal) will be assigned to **a**. If you type 9, an invalid number in the octal system, the value of **a** will remain unchanged. Similarly, if you type 29, only the 2 will be recognized.

%s captures a string typed at the keyboard. The string must be assigned to a pointer or an array. For example:

```
main()
{
      char p[80];
      scanf("%s", p);
      printf("%s", p);
}
```

obtains the string typed at the keyboard and prints it out. If you type:

This string is mine.

that sentence will be assigned to the character array p. Note: the ampersand (&) is omitted because an array name without a subscript is a pointer to the array.

%x captures the value of a string of hexadecimal digits. For example:

```
main()
{
      int a;
      scanf("%x", &a);
      printf("%x", a);
}
```

obtains the value of the next valid sequence of hexadecimal digits typed at the keyboard. The leading 0x (signifying a hexadecimal number) is optional. However, the first character input must be a valid digit in the hexadecimal number system or the input will be ignored.

%f captures a fixed decimal floating point value in the form i.f. The precision field (described below) specifies the number of places after the decimal point. The default is 6 decimal places. A precision of zero suppresses the period in the floating point number. For example:

```
main()
{
      float n;
      scanf("%f", &n);
      printf("%f", n);
}
```

obtains in floating point format the next number input from the keyboard. If the number 23.234200 is typed, the value 23.2342 will be assigned to n and printed.

Precision field
As with **printf**, additional instructions can be placed between the % and the conversion character. This is often called the precision field. The form of the precision field can be expressed as:

*% * # l a*

(In actual use there should be no white space within the precision field; it is included here for the convenience of the reader.)

Taking these options individually:

* An asterisk (*) indicates that the conversion is to be performed, but the resulting value is not to be returned by **scanf** to the object pointed to by an argument. For example:

```
scanf("%*d");
```

will ignore the next decimal number. This is primarily of use in the **fscanf** function which extracts data from a file. See also, example [2] below.

The number (symbolized by **#**) specifies the maximum field width for the input governed by a variable value. For example, **%20s** allocates 20 spaces for a string of characters being passed to it from the operator. Thus:

```
main()
{
        char n[80];

        scanf("%20s\n", n);
        printf("%20s\n", n);
}
```

Captures the first 20 characters input at the terminal to the **n** character array. **printf** prints these characters in a right-justified format.

l The letter **l** (el) indicates that the data item is a **long** (long integer). For example:

```
main()
{
        long a;
        int i;

        scanf("%d%ld",&i,&a);
        printf("%d %ld", i, a);
}
```

Captures two decimal numbers, the second of which may be a **long. printf** prints the results.

Data matching
Another way to ignore scanned material is to provide an exact match for the material in the **scanf** formatted command. For example:

```
int a, b;
scanf("Cows: %d Cowboys: %d", &a,&b)
```

would correctly extract the numbers 25 and 50 from the following input:

Cows: 25 Cowboys: 50

Like the asterisk formatting option, data matching is primarily useful in **fscanf** where it is necessary to extract information from a file with a fixed data format.

NOTES

* The format type must be specified in lower case. Spaces, tabs and newline characters (collectively known as white space) are generally ignored. Input is scanned through white space to the next input item in all cases except the single byte **char** specifier, which will capture the next input character, whitespace or otherwise.

* **scanf** will not return control to the calling statement until either it has filled all its "blanks" or an error has been detected.

* Since **scanf** treats groups of one or more whitespace characters as field separators, **scanf** is not appropriate for capturing a line of text. The function **gets** should be used instead.

* Note the value returned by **scanf**. It will tell you how many items were successfully scanned, if any.

* **scanf** is quite particular about the type of input it is expecting. If you specify a **%d** decimal number **scanf** will ignore input until it finds a decimal number. The same is true for the other types of objects: octal, strings, etc.

* Notice that **scanf** lacks the **.#** option which in **printf** allows you to control the number of whole numbers or decimal digits printed.

RETURNS -1, if an error is detected.
number of input data items successfully matched, (i.e., for which valid data was found; this includes all single character items in the control string), if successful.

EXAMPLE [1]

```
main() /* scanf.c -- example of scanf() */
{
        int ret;
        int i;
        float x;
        char name[50];

        puts("Scanf is looking for a decimal
        number, a floating point, and a
        string no greater than 50 characters
        -- remember, no white spaces in
        string.");

        ret = scanf("%d %f %50s", &i, &x,
        name);
        printf("%d returned values: %d %f
        %s", ret, i, x, name);
}
```

The printed string will be:

```
4 returned values: 56   789.000000 45
```

(Continued next page)

[2]
```
main()
{
        int ret;
        int i;
        float x;
        char name[50]
        ret = scanf("%2d %f %*d %2s", &i,
        &x, name);
}
```

with an input of:

```
56789 0123 45a72
```

will assign 56 to **i**, 789.0 to **x**, skip over 0123 and place the string "45" in **name**. The number assigned to **ret** will be 4, the number of separate items processed by **scanf**.

SEE ALSO fscanf, sscanf, printf, gets

PURPOSE Displays the RUN/C options which you can modify using the SET convention.

FORM [1] SET

[2] SET *command [on] [off] [parameter]*

USAGE SET lists the user options which have been assigned to the SET command in RUN/C. After the SET command is given the program will respond either telling you the newly SET parameter or asking for additional information. In some cases a Y or N answer is appropriate (such as the SET TRUST ON -- see below) while in other cases a name or number may be supplied.

NOTES Over time much of the debugging and other facilities which will be added to RUN/C will be accessible using the SET command format.

EXAMPLE SET

```
Valid commands are:
      SET EDITOR name
      SET PRINT ON/OFF
      SET QUIET ON/OFF
      SET STEP ON/OFF
      TRUST ON/OFF

Ok
```

SEE ALSO Other SET commands

PURPOSE Establishes an internal 'alias' name for the EDITOR command.

FORM [1] SET EDITOR *filename*

[2] SET EDITOR *drive: filename*

USAGE SET EDITOR provides an alternative to actually renaming a screen editor to either EDITOR.COM or EDITOR.EXE, which requires that a separate copy of the program be kept on an active disk.

The filename should be the same as that which you would use to invoke the screen editor or word processing program. For example if you want to use IBM's Personal Editor, the command:

```
SET EDITOR PE
```

will search for an executable program PE.COM or PE.EXE on the DOS path (see your DOS manual for information on setting a PATH for executable programs) whenever the EDITOR command is invoked. If the file is not found the error message:

```
Can't find PE
```

will be displayed. If PE is found the message:

```
EDITOR now set to PE
```

will be displayed.

Standard naming conventions for filenames, disk and directory pathnames as described in Section 1 apply. If no drive or directory is specified, the current default drive/directory will be assumed, as illustrated in Form [1]. If a drive/directory pathname is supplied, RUN/C will call that program using the specified pathname.

NOTES * Although pathnames may be used, EDITOR will not support an alias with a pathname unless 1) any additional files related to the editor are on the default drive/directory, and 2) the screen editor or word processing program can be called from DOS to your default drive/directory.

* **SET EDITOR** must be invoked each time RUN/C is loaded. It is not at present possibly to permanently assign an alias for **EDITOR** other than to create an EDITOR.COM or EDITOR.EXE or to set up an EDITOR.BAT file as described in this section under **EDITOR**.

EXAMPLES Ok

```
SET EDITOR WS

EDITOR set to WS

Ok
```

SEE ALSO SET, EDITOR; Section 2

PURPOSE Sets the single-step mode which allows the user to control program execution with keyboard input.

FORM [1] SET STEP ON

[2] SET STEP OFF

USAGE **SET STEP ON** activates the single stepping debugging mode. When active, programs will execute statement-by-statement only after a key has been pressed. Single stepping is best used in conjunction with other debugging commands such as **TRACE** or **TRON**.

When single stepping is active program output and execution will cease until and unless a key is hit.

SET STEP is deactivated with the **SET STEP OFF** command.

When single stepping is first activated the message:

```
Single-step mode is now ON
```

will appear. When deactivated the following message will be generated:

```
Single-step mode is now OFF
```

NOTES
* Single stepping may be turned on or off from the command line, through a **runc()** function statement or from the I (Interactive) mode.
* Unless single stepping is combined with other debugging commands it may be necessary to press a key several -- or even many -- times before some activity is evident. If in doubt about the status of your program press <Ctrl-c> and deactivate single stepping using the I(Interactive) command mode.
* The **SET STEP** command is only available in RUN/C Professional.

SEE ALSO SET, TRACE, TRON, I(Interactive), Section 7

PURPOSE Sets the "Trust Me" mode which potentially controls whether program pointers can be assigned to any location in memory.

FORM [1] SET TRUST ON

[2] SET TRUST OFF

USAGE **SET TRUST ON** disables an interpreter safety feature which prevents user program pointers from being assigned to an address outside the memory space normally reserved for user programs and data.

The pointer trap is designed specifically to prevent pointers from passing random or erroneous values to specific memory locations which could damage system data, including such relevant items as the interpreter which contains and manages your program, the source code of your program and the File Allocation Table (FAT) which maintains your disk directory.

Whenever you give the command:

SET TRUST ON

the following message will appear:

```
With trust mode ON, you may point to any location
in memory.
The potential for data loss and other problems
is real.

    Do you want trust mode ON (Y/N)?
```

If the answer is Y, the message:

```
Trust mode is now ON
```

will appear. If you answer N the message:

```
Trust mode remains OFF
```

will be displayed.

If trust mode is on, you can switch it off with:

```
SET TRUST OFF
```

And the following acknowledgment will appear:

```
Trust mode is now OFF
```

In the example below an address in memory is cast as character pointer **p**. In other words, the pointer is set to an absolute address in memory. Since the address you choose is probably outside the allocated memory for your RUN/C program and data, the attempt will be trapped unless you have the TRUST flag set to on.

NOTES

In RUN/C Professional the TRUST flag may be turned on or off using the **runc()** function with either:

```
runc("SET TRUST ON");
```

or

```
runc("SET TRUST OFF");
```

When the TRUST mode is invoked from the executing program in this manner, the user will not be double queried regarding his/her true intentions about turning the TRUST flag on.

(Continued next page)

EXAMPLE

```
LOAD SETTRUST

NLIST

main()     /* settrust.c -- illustrates SET
           TRUST command */
{
     char *p;
     int i;
     long a;
     printf("Starting address? ");
     scanf("%ld",&a);
     p = (char *) a; /* address is cast as a char
     pointer */
     for (i = 0; i < 25; i++, p++)
          printf("%02x ", *p); /* and the value
          pointed to listed */
}

Ok
RUN

Starting address? 2500
*** Error # 65 at line 9
Pointer value outside user data space.

Ok
SET TRUST ON

     With TRUST mode on, you may point to any
     location in memory. The potential for data
     loss and other problems is real.

     Should we trust you (Y/N)? Y

Trust mode is now ON

Ok
RUN

Starting address? 2500
02 00 26 8A 05 47 32 E4 E8 1D 00 75 05 E2 F0 E9
64 FE 4F 4B 75 EC E9 48 FE
Ok
```

SEE ALSO SET

PURPOSE Initialize an asynchronous communication port for various communication parameters.

FORM

```
char port;
char mode;

setcom(port,mode);
```

USAGE **setcom** sets the selected communication port for allowable character length (7 or 8 bits), number of stop bits (1 or 2), parity (none, odd or even) and baud rate (110, 150, 300, 600, 1200, 2400, 4800 or 9600).

The IBM PC has four communication ports numbered 0 through 3.

In the above sample:

`char port;`
declares a variable to hold the port number.

`char mode;`
declares a variable to hold the mode code.

`setcom(port,mode);`
sets the communication port number assigned to **port** to the communication mode parameters encoded in the byte assigned to **mode.**

The IBM PC and most other microcomputers employ what is known as a binary bit pattern to set communication parameters. When it comes to communications, the PC uses an "8 bit" pattern which can be depicted as:

(Continued next page)

bit pattern							
7	6	5	4	3	2	1	0
b	b	b	p	p	s	c	c
a	a	a	a	a	t	h	h
u	u	u	r	r	o	a	a
d	d	d	i	i	p	r	r
			t	t		a	a
r	r	r	y	y	b	c	c
a	a	a			i	t	t
t	t	t			t	e	e
e	e	e			s	r	r
						l	l
						e	e
						n	n
						g	g
						t	t
						h	h

Although there are eight places in the bit pattern, keep in mind that each place represents a single digit binary number which can only be 0 or 1. Thus, the number of possible of combinations of these eight binary numbers is only 256.

As the above chart shows, various bit values control the communication mode setting. For example, bit 2 -- the third bit from the right -- controls the number of stop bits sent with each item of information to the communication port. If bit 2 is set to 0, 1 stop bit will be sent. If bit 2 is set to 1, 2 stop bits will be sent. If bit 2 is set to 2 -- hey, we're talking binary here! No bit can ever be greater than 1 or less than 0.

As noted in the above chart:

Bits 0 and 1 control character length:

10 = 7 data bits
11 = 8 data bits
00 = illegal
01 = illegal

Bit 2 controls the number of stop bits:

0 = 1 stop bit
1 = 2 stop bits

Bits 3 and 4 controls the parity setting:

00 = no parity (none)
01 = odd parity
11 = even parity
10 = no parity

Bits 5 - 7 control the baud rate:

000 = 110 baud
001 = 150 baud
010 = 300 baud
011 = 600 baud
100 = 1200 baud
101 = 2400 baud
110 = 4800 baud
111 = 9600 baud

Picking from our binary menu, if we want our communications mode to be set for 300 baud, 8 data bits, 1 stop bit, even parity (the equivalent of the MS-DOS command **MODE 300,E,8,1**) we would use the binary number:

bit position	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
bit value	0	1	0	1	1	0	1	1

How did we get this number?

Positions 0 and 1 control *character length.* 11 (binary number 3) sets the number of data bits in each serial transfer group at 8.

Position 2 controls the number of *stop bits.* Our table shows that a 0 in that position means 1 stop bit will be sent after every group of 8 data bits.

Positions 3 and 4 control the *parity setting.* 11 (binary 3) in those positions means that the parity will be set to EVEN.

Positions 5 through 7 set the *baud rate.* From our chart we know that 010 means 300 baud.

Unfortunately, binary numbers are not currently part of the C language. So we must convert the binary number 01011011 to a decimal number (91). The best way to do

this is to use a binary-to-decimal calculator, but it is possible to calculate the number by hand using a binary to decimal conversion chart or an abacus.

Now that we have a decimal number we can use the command:

```
setcom(port,mode);
```

where mode has previously be set equal to 91.

NOTES

* **setcom** should be expected to work properly only on the IBM PC and close compatibles.

* If a port is specified which is not equipped for asynchronous communications (for example, if there were no modem properly hooked up to the line, no serial port, improper cabling or if the equipment was faulty or not plugged in or turned on or not initialized properly, **setcom** will not provide communications to the world at large.

* The function **setcom** is similar to the "pcasm" function in Lattice's C Food Smorgasbord. The object library file or source code from this collection of utility functions is needed in order to compile a program using **setcom.** In addition, the name **setcom** must be changed to **pcasm** prior to compilation.

RETURNS Nothing.

EXAMPLE

```
main() /* setcom.c -- initializes a port */
{
     setcom(0,91);
     aput('H');
/*
Initializes communications port #0 to 300 baud,
even parity, 1 stop bit, 8 data bits, and
sends the character 'H' out to port #0.
*/
}
```

SEE ALSO aput, aget

PURPOSE Transfers control of the computer to the operating system or directly to another program.

FORM [1] SHELL

[2] SHELL drive:*dir**filespec*

[3] SHELL *DOS command*

USAGE **SHELL** allows you to temporarily leave RUN/C to perform operating system activities (such as changing the logged-in drive/directory) or to transfer control to another program such as a C compiler.

In Form [1], above:

SHELL

simply calls the operating system. The message:

```
Type 'exit' when done with external
operation(s).
```

will be displayed to remind you how to get back to RUN/C.

Form [2], above, illustrates the use of **SHELL** with a command string. Using this technique it is possible to call an external program from within RUN/C and to pass a command line to that program.

One useful application of this technique is to use **SHELL** to call a C compiler to compile your program. See example below for an illustration of this technique.

NOTES

* There must be sufficient memory for COMMAND.COM or the program you invoke and RUN/C and any program in RUN/C's memory to be co-resident. For a fuller discussion on this subject, see EDITOR and SHELL in Section 2.

* If you have a program in memory when you invoke **SHELL**, your program will remain in memory and be available to you when and if you return from **SHELL**. This is a convenient feature, but it is still prudent to save your program frequently including prior to executing **SHELL** operations.

* The key to successfully using **SHELL** to compile C programs or with other memory-hungry applications is having sufficient memory. Depending on the compiler and the complexity of the program you are trying to compile, you will need anywhere from 128k to 256k of memory in addition to that used by RUN/C.

* In RUN/C Professional **SHELL** can be called from the executing program using the **runc()** function. This allows the executing program to perform limited batch processing at the DOS level as part of its task.

EXAMPLES:[1] SHELL

```
Type 'exit' when done with external
operation(s).
```

Then the DOS sign-on message will appear. At the DOS level you might perform the following operation:

```
A>DATE

Current date is Wed 1-09-1985
Enter new date:

12-5-85

A>EXIT

Press any key to return to RUN/C ...

Ok
```

[2] LOAD HAP-PROG

```
Ok
NLIST

main()
{
      printf("I'm a happy program.");
}
Ok
RUN
      I'm a happy program.
```

```
Ok
EDIT

        (Insert: #include "stdio.h" at the
      beginning of the file.)

Ok
SHELL LC1 PROGRAM

Lattice C Compiler (Phase 1) V2.13
Copyright (C) 1982 Lattice, Inc.

Press any key to return to RUN/C...
(Back in RUN/C)

Ok
SHELL LC2 PROGRAM

Lattice 8086 Code Generator V2.13
Copyright (C) 1982 Lattice, Inc.
Module size P=000E D=0015

Press any key to return to RUN/C...

(Back in RUN/C)

Ok
SHELL LINK CS+PROGRAM,PROGRAM.EXE,,LCS

 Microsoft Object Linker V1.10

Press any key to return to RUN/C...

(Back in RUN/C)

Ok
SHELL PROGRAM

I'm a happy program.

Press any key to return to RUN/C...

(Back in RUN/C)
Ok
```

SEE ALSO EDITOR, AUTOSAVE

PURPOSE Creates a variable limited to a value in the range -32,768 to 32,767.

FORM `short a = 9273;`

USAGE **short** is practically identical to **int** in this implementation of RUN/C and on most C systems running on the Intel 8086/8088 chip.

NOTES

* In C, simple data type names (such as **short**) have four distinct uses, namely:

  (1) in declaring variables and aggregate numbers;

  (2) in casting one type of value to another such as:

  ```
  d = (short) 0;
  ```

  (3) as an argument to **sizeof**, as in:

  ```
  int i = sizeof(short);
  ```

  (4) as an element in a **typedef** statement used in creating a synonym for an existing data type such as:

  ```
  typedef short SMALL;
  ```

* The data type **short int** is unsupported.

SEE ALSO `char, double, int, float, long, unsigned`

PURPOSE Calculates the sine of a number.

FORM

```
double result;
double number;

result = sin(number);
```

USAGE **sin** computes the trigonometric sine of a number expressed in radians. Both the passed number and the returned value are expressed as **doubles.**

NOTES In RUN/C Professional this and other transcendental math functions are available in the loadable library MATHLIB.

RETURNS Sine of a number.

EXAMPLE

```
main() /* sin.c -- computes sine of a number */

{

double x;
double result;

x = 30;

result = sin(x);
printf("%f", result);

/* Correct result: -0.988059 */
}
```

PURPOSE Calculates the hyperbolic sine of a number.

FORM

```
double result;
double number;

result = sinh(number);
```

USAGE **sinh** computes the hyperbolic sine of a number. Both the passed number and the returned value are expressed as **doubles.**

This can be expressed algebraically as:

$$\sinh(x) = \frac{e^{x} - e^{-x}}{2}$$

NOTES In RUN/C Professional this and other transcendental math functions are available in the loadable library MATHLIB.

RETURNS Hyperbolic sine of a number.

EXAMPLE

```
main()      /* sinh.c -- computes hyperbolic sine
               of a number */

{

double x;
double result;

x = 6;

result = sinh(x);
printf("%f", result);

/* Correct answer: 201.713157 */
}
```

PURPOSE Calculates the size in bytes of a data type or data object.

FORM `printf("%d\n" sizeof(char));`

USAGE **sizeof** computes the size of an operator or declared variable.

RETURNS Nothing.

EXAMPLE

```
main()

/* sizeof.c -- illustrates size of built-in data types */

{
int a = 420;
float b = 23.2342;
char c = 'A';
double d = 23423423.234234423;
unsigned e = 64000;

     printf(" a (%d) occupies %d bytes\n", a,sizeof(a));
     printf(" b (%f) occupies %d bytes\n", b,sizeof(b));
     printf(" c (%c) occupies %d byte\n", c,sizeof(c));
     printf(" d (%e) occupies %d bytes\n", d,sizeof(d));
     printf(" e (%u) occupies %d bytes\n", e,sizeof(e));

/* correct answers: a (2 bytes), b (4 bytes) */
/* c (1 byte), d (8 bytes), and e (2 bytes). */
}
```

SEE ALSO `calloc` (for another example of the use of **sizeof**).

PURPOSE Places a formatted string in memory.

FORM

```
int ret;
char m[128];
char *n;
n = "this string is my string";

ret = sprintf(m, "%s", n);
```

USAGE **sprintf**, formats and sends data to a string in memory which can be referenced by a pointer. Instead of output to the standard output device (**printf**) or a file (**fprintf**) or the printer (**lprintf**), **sprintf** writes its output to a location in memory. See **printf** for a complete discussion of **sprintf** formatting options.

In the above sample:

`char m[128];`
declares an array of 128 characters.

`char *n;`
declares a character pointer.

`n = "this string to memory";`
assigns string to pointer **n**.

`ret = sprintf(m, "%s",n);`
formats string **n** and sends it to array **m** for later manipulation. The value returned is assigned to **ret**.

The number of characters placed in the array will be returned.

Because **sprintf** is printing to an array in memory, a zero byte is automatically appended to the end of the formatted string.

Like **printf**, **sprintf** uses several formatting codes of the form **%a** where the **%** alerts the function that a special character is coming. The character which follows the **%** tells **sprintf** what to do with the data being passed by variables into the program.

Option	Data type
%c	single character
%d	signed decimal
%e	exponential floating point
%f	fixed decimal floating point
%g	e or f format, whichever is shorter
%o	octal
%s	string
%u	unsigned decimal
%x	hexadecimal
%%	prints %

You may also optionally place additional instructions between the % and the conversion character. This is the *precision field.* It provides further ability to control the form of the printed output.

The form for this can be expressed as:

% - # . # l a

(In actual usage there are no spaces between the codes in the precision field; they are included here for readability.)

The conversion options and their meanings are:

Option	Meaning
-	left justify output
#	minimum number of spaces in field
.#	for whole numbers, determines the number of characters which can be passed to the field. For floating point numbers, the second # tell how many decimal places to provide for the formatted number.
l	declares that the **int, float** or **unsigned** shall be treated as a **long.**

NOTES * Be certain any created array will be large enough to hold all the formatted characters created by **sprintf.**

RETURNS Number of characters in the formatted string (including the NULL byte), if successful.

EXAMPLE

```
main() /* sprintf.c -- illustrates sprintf() */

{
     int a,b;
     char c[80];
     sprintf(c,"%s","this is a string");
     puts(c);

     puts("\nCorrect answer: 'this is a string'
     is displayed.");
}
```

SEE ALSO printf, sscanf

PURPOSE Calculates the square root of a number.

FORM

```
double result;
double number;

result = sqrt(number);
```

USAGE sqrt computes the square root of a number. Both the called number and the returned value are expressed as doubles.

NOTES

* You cannot take the square root of negative numbers.

* In RUN/C Professional this and other transcendental math functions are available in the loadable library MATHLIB.

RETURNS 0, if the number is 0.
Square root, if the number is positive.

EXAMPLE

```
main()     /* sqrt.c -- squares two numbers */
{
double x;
double result;

     x = 9;

     result = sqrt(x);
          printf("%.f is the square root of
          %.f\n", result, x);
     result = sqrt(result);
          printf("%.17f is the square root of
          %.f", result, sqrt(x));
}
```

PURPOSE Resets the random number seed used by **rand.**

FORM

```
unsigned seed;

srand(seed);
```

USAGE **srand** resets the pseudo-random number generator to a new starting point. The default initial seed value is 1 but a seed value up to 65,535 can be used.

NOTES In RUN/C Professional this and other transcendental math functions are available in the loadable library MATHLIB.

RETURNS Nothing.

EXAMPLE

```
main()

/* srand.c -- illustrates srand() seeds random
number generator for repeatable sequence */

{
     int i, j, n;
     unsigned u;

     u = 392;
     for (j = 0; j < 2; j++) {
          srand(u);
          for (i = 0; i < 5; i++) {
               n = rand();
               printf("%d ",n);
          }
          puts("");
     }
     puts("Both sequences should be the same.");
}
```

SEE ALSO rand

PURPOSE Performs formatted input conversions from a string in memory.

FORM
```
int ret;
char *ss;
double aa, bb;

ret = sscanf(ss, %g %g, &aa, &bb);
```

USAGE **sscanf**, like its cousins **fscanf**, and **sscanf**, scans input and performs formatted conversions on data. In the case of **sscanf** the data is obtained from a string in memory. The data is captured in a form specified by the conversion string and the formatted data is then placed in the pointer arguments listed.

In the above sample:

`int ret;`
declares a variable to contain the number of input items matched, or EOF.

`char *cs;`
declares a pointer to the format control string.

`double aa, bb;`
declares two variables of type **double.**

`ret = sscanf(ss, %g %g, &aa, &bb);`
scans the specified string (**ss**) for input, assigning the first two valid data items to the doubles **aa** and **bb.** The returned value from the function will be assigned to **ret.**

The returned value is the number of items successfully scanned, unless an error is detected, in which case -1 will be returned.

Discussion
sscanf provides for the extraction of information from a string in memory in sequential order using the appropriate conversion characters of the format **%a** where the **%** alerts the computer that a special character is coming. The lowercase alphabetic character tells **sscanf** how to handle the data being captured.

The conversion characters are:

Option	Data type
%c	single character
%d	signed decimal
%f,%e,%E	fixed decimal floating point
%o	octal
%s	string
%u	unsigned decimal
%x	hexadecimal
%%	a percent sign is expected

For a fuller explanation of the use and effects of these variable types, see **scanf.**

Like **scanf, sscanf** allows you to place additional formatting instructions between the % and the conversion character. This is referred to as the *precision field.* It provides further control over the form of the printed output. This form can be expressed as:

% * # l a

(In actual usage there should be no white space in the precision field; it is included here for the convenience of the reader.)

The conversion options are:

Option	Effect
*	indicates that the conversion is to be performed, but the resulting value is not to be returned.
#	specifies the maximum field width for the input data item.
l	(el) indicates that the data item is a **long** (long integer) or **double.**

Input is scanned through white space to locate the next input item in all cases except the **%c** char specifier, where the next input character is returned regardless.

Data matching
An alternative to insuring that scanned material is properly ignored to provide an exact match for the material in the **sscanf** formatted command. For

example, the program fragment:

```
int a,b;
   sscanf("Cows: 25 Cowboys: 50", "Cows: %d
   Cowboys: %d", &a,&b)
```

would correctly extract the numbers 25 and 50.

NOTES All of the input values to **sscanf** must be pointers. However, if the assignment suppression feature (*) is used, a pointer must not be supplied for that specifier since pointers are matched to data on a one-to-one basis. Since the asterisk suppresses returned data, **sscanf** would lose its place if a pointer was available for data which was never sent.

RETURNS -1, if an error is detected.
number of input items successfully matched, i.e., for which valid text data was found; this includes all single character items in the control string.

EXAMPLE

```
/* demonstrates sscanf() (adapted from K&R's C
Prog. Lang., p. 149) */

main()    /* sscanf.c -- illustrates sscanf()
          */
{
     char *input;
     int i;
     float x;
     char name[50];

     input = "25    54.32E-1    Thompson";
     sscanf(input,"%d %f %s", &i, &x, name);
     printf("%d %f %s ", i, x, name);
     input = "56789 0123 45a72";
     sscanf(input,"%2d %f %*d %2s", &i, &x,
     name);
     printf("%d %f %s\n", i, x, name);
     puts("Correct answer: 25 5.432000 Thompson
     56 789.000000 45");
}
```

SEE ALSO scanf, fscanf, gets

PURPOSE Creates a variable of the specified data type which will hold its current value even after program control has passed to another function.

FORM

```
static int a = n;
static char b = n;
```

USAGE **static** provides a means of assigning a value to a variable which will be retained, even when the function is not actively being executed. In other words -- those of K&R, in fact -- a "**static** variable remains in existence rather than coming and going each time the function is activated. This ... provides private, permanent storage in a function."

In the above sample, integer **a** is assigned a numeric value and declared to be a **static** variable. Similarly, **static char** sets up a character variable **b** and initializes in to a numeric value.

There are two types of **statics** -- local and global (external), the type being determined by the position of the **static** in the program. If the variable is declared **static** within a function, the variable value will be retained locally. Thus a program can enter a function repeatedly without having to reinitialize a variable's value.

static may be used with any data type: **char, double, float, int, long, unsigned,** and structures and unions.

NOTES

* Although a variable can be declared as a **static** globally -- outside a function -- this will have little effect in an interpreted environment. In compiled situations external **statics** are used to "hide" information from other program modules.

* It is important to recognize that **statics** can only retain the value of a variable by "permanently" taking up memory space during program execution. For this reason **statics** are used only when needed.

* The default storage class **auto** is, in a sense, the opposite of **static.** In RUN/C it is not necessary to declare variables **auto,** although such a declaration is legal within functions.

EXAMPLE

```
main()      /* demonstrates use of static */
{
     int i;

     for( i= 0; i < 10; ++i)
          static_demo();
}

static_demo()
     {
          int variable = 0;
          static int static_variable = 0;

          printf("automatic = %d, static =
          %d\n", variable,
static_variable);
          ++variable;
          ++static_variable;

/*
Result: the static variable will be incremented
each pass through the function up to the value
10; but the auto variable will be reset to 0
each time the function is entered.
*/
}
```

SEE ALSO auto

PURPOSE Provides an error reporting facility independent of whatever is the currently assigned standard error output device (stderr).

FORM
```
if (fopen("data", "r") == NULL)
    fprintf(stderr, "Couldn't open file.\n");
```

USAGE **stderr** complements **stdout** by providing a convenient method of alerting the operator to error conditions when program output is otherwise being directed to another output device such as a file, a printer or a communication port.

Actually, **stderr** is a file pointer which is automatically opened when RUN/C is active to provide output to the console screen when an error condition occurs. In this implementation the error output will go to the screen regardless on the device which is currently assigned the output.

In the above sample:

`if (fopen("data", "r") == NULL)`
if **fopen** returns a NULL indicating that the file "data" cannot be read . . .

`fprintf(stderr, "Couldn't open file.\n");`
. . . then the error message will be printed on the screen.

The example below illustrates a variation on this technique. When the program is **RUN** output taken from the text file TEST.FIL via redirected I/O is sent to a new file named CAPTURE. (TEST.FIL is supplied on the RUN/C distribution disk.)

NOTES
* The file pointers **stdin, stdout** and **stderr** are automatically opened under RUN/C. Thus it is possible to do limited file I/O without having to "manually" open and close particular files, while **stderr** remains available as a channel to the console for prompts and other messages.

* See the note under **stdin** regarding the number of files automatically opened by RUN/C and means of increasing the maximum number of files which may be open simultaneously.

EXAMPLE

```
main()

/* std stuff -- demonstrates stdin, stdout,
stderr */
/* file copier with console echo */
{
FILE *fi;
char filename[20];
int c;
char *source, dest[25];

printf("Input filename: ");
fscanf(stdin, "%s", filename);


if((fi = fopen(filename, "r")) == NULL) {
 fprintf(stderr, "\tCan't open %s\n",filename);
  exit();
}

while((c = fgetc(fi)) != EOF) {
     fputc(c, stdout);
     }
fclose(fi);
puts("\n{DONE}");
```

SEE ALSO stdin, stdout, getchar, fopen

PURPOSE Provides an input facility for the running program.

FORM

```
int i;
fscanf(stdin, "%d", &i);
printf("%d", i);
```

USAGE **stdin**, an abbreviation for standard input, is a file pointer which is held open to provide input to the program. Unless otherwise directed, **stdin** is obtained from the console keyboard; however I/O may be redirected to other input devices in which case stdin would be obtained from the assigned device such as a communication port or a disk file.

In the above sample:

`int i;`
declares integer variable **i**.

`fscanf(stdin, "%d", &i);`
scans the standard input device for a single decimal value, placing the value found at the address of **i**.

`printf("%d", i);`
prints the value obtained from the input device.

stdin can be used to take input from a data file which might otherwise be supplied from the keyboard. **stdin** might be taken from a file which contains the names of a series of files which are to be processed. The example at the end of this discussion illustrates a variation on this technique. When the program is RUN with redirected I/O, input is taken from the file TEST.FIL which contains a series of numbers. (TEST.FIL is supplied on the RUN/C distribution disk.)

Another use of redirected I/O in conjunction with **stdin** is to use DOS's device names. For example:

```
RUN < CON:
```

will redirect the program's input to the operator's console. Thus program input can be redirected to a testing device (or whatever) without having to modify

the program. The input-oriented device names available for DOS are:

Name	Function	Direction
CON:	Console	Input and output
AUX:	Com. port 1	Input and output
COM1:	Com. port 1	Input and output
COM2:	Com. port 2	Input and output
NUL:	Test device	Neither input nor output

See your DOS manual for details on using these and other device names.

NOTES

* In C standard I/O functions that perform input and do not take a file pointer as an argument get their input from **stdin.**

* The file pointers **stdin and stdout** are automatically held open when RUN/C is active. This makes it possible to do limited file I/O without having to "manually" open and close particular files.

* The existence of **stdin, stdout** and **stderr** increases the number of files open by three in addition to RUN/C itself. It is important to keep this in mind in situations where your program holds several files open. In general it is recommended that you increase the number of files DOS can hold open simultaneously from the default 8 to 16. For most implementations of DOS the line:

```
FILES = 16
```

placed in the CONFIG.SYS boot-up file will accomplish this goal at a very minimal memory cost. See your DOS manual for complete details regarding DOS's **FILES** setting and CONFIG.SYS.

* Some C functions are commonly implemented as macros using the **stdin** file pointer. For example, many compilers implement the function **getchar** as a macro with the following line in the stdio.h file:

```
#define getchar() getc(stdin)
```

which uses the *Level 2* file input function **getc** to get characters from standard output. Thus the source of the characters obtained by **getchar** can change unexpectedly depending upon what is currently the standard input device.

EXAMPLE

```
main()
{
int i;
while(( i = getc(stdin)) != EOF )
     putc(i,stdout);
     puts("Done!");
}
```

SEE ALSO stdout, stderr, RUN, getchar, getc, fopen

PURPOSE Provides an output facility for the running program.

FORM

```
char *string_ptr="Hello standard output device";

fprintf(stdout, "%s", string_ptr);
```

USAGE **stdout**, an abbreviation for standard output, is a file pointer which is automatically opened when RUN/C is active to accept output from the program. Unless otherwise directed, **stdout** is sent to the console screen; however redirected I/O can cause output to be sent to a file or another output device.

In the above sample:

`char *string_ptr="Hello standard output device";`
initializes a character pointer to the string in quotes.

`fprintf(stdout, "%s", string_ptr);`
sends the string pointed to by **string_ptr** to the standard output device.

stdout is used to send output which might otherwise be displayed on the screen to a data file. Another frequent use of redirected I/O in conjunction with **stdout** is to use DOS's device names. For example:

```
RUN > LPT1:
```

will redirect the program's output to the first parallel printer port. Thus program output can be easily captured at the printer without having to modify a single line of the program. The DOS device names available for output are:

(Continued next page)

Name	Function	Direction
CON:	Console	Input and output
AUX:	Com. port 1	Input and output
COM1:	Com. port 1	Input and output
COM2:	Com. port 2	Input and output
LPT1:	Parallel 1	Output only
PRN	Parallel 1	Output only
LPT2:	Parallel 2	Output only
LPT3:	Parallel 3	Output only
NUL:	Test device	Neither input nor output

See your DOS manual for details on the meanings of these system device names.

The example below illustrates a variation on this technique. When the program is **RUN** output taken from the text file TEST.FIL is sent to a new file named CAPTURE. (TEST.FIL is contained on the RUN/C distribution disk.)

NOTES

* In C, standard I/O functions that perform output and do not take a file pointer as an argument send their output through **stdout.**

* The file pointers **stdin, stdout** and **stderr** are automatically held open whenever RUN/C is active. Thus it is possible to do some file I/O without having to "manually" open and close particular files.

* Some C functions are commonly implemented as macros using the **stdout** file pointer. For example, many compilers implement the function **putchar** as a macro with the following line in the stdio.h file:

```
#define putchar(c) putc(c,stdout)
```

which uses the *Level* 2 file input function **putc** to get characters from standard output. Thus the destination of characters output with **putchar** can change unexpectedly depending upon what is

currently the standard output device.

* See the note under **stdin** regarding the number of files automatically opened by RUN/C and method of increasing the maximum number of files which may be simultaneously held open.

EXAMPLE

```
main()      /* illustrates stdout -- stdout.c */
{
int i;
for( i = 40; i < 80; ++i){
fprintf(stdout, "hello there.\n");
printf("hello there\n");
}
```

SEE ALSO stdin, stderr, RUN, putchar, putc, fopen

PURPOSE Concatenates two strings.

FORM

```
char *to;
char *from;

strcat(to, from);
```

USAGE **strcat** concatenates two strings each of which is accessed by a pointer. In the above sample, the character string **from** is appended to the end of character string **to.** The resulting combined string is automatically NULL-terminated.

In the above sample:

`char *to;`
declares a pointer to the target string.

`char *from;`
declares a pointer to the source string.

`strcat(to, from);`
appends NULL-terminated string at address **from** to the end of string at address **to.**

NOTES

* There is no checking to prevent the length of the source string from being longer than the destination string; therefore, be sure that the destination string is large enough to receive the combined result. If this is not done, the data will overwrite an area of memory, with unpredictable and probably unwanted results.

* Use **strlen** to help determine in advance the maximum size of the combined string.

RETURNS Nothing.

EXAMPLE

```
main()      /* strcat.c -- illustrates strcat() */

{
char result[20], *first, *second;
first = "How now";
second = " brown cow?";

     strcpy(result, first);
     strcat(result, second);
     puts(result);

/* prints "how now brown cow" */
}
```

SEE ALSO strcmp, strcpy, strlen

PURPOSE Compares contents of two strings.

FORM

```
int status;
char *a;
char *b;

status = strcmp(a, b);
```

USAGE **strcmp** compares two NULL-terminated strings, byte for byte, returning a 0 value if the strings are identical, up to and including the terminating zero byte.

In the above sample:

`char *a;`
declares a pointer to the first string.

`char *b;`
declares a pointer to the second string.

`int status;`
declares a variable to hold the returned value of the comparison.

`status = strcmp(a, b);`
a byte-by-byte comparison of string **a** with string **b** is made. If the NULL-terminated byte of both strings is successfully reached, the strings are indeed identical. The returned value from the function will be assigned to **status.**

Identical strings are defined as those which have the same number of characters and matching characters, in order, up to and including the terminating zero byte.

If the returned value is positive, it means that the first non-identical byte in the comparison had a higher ASCII value in the first string. If the returned value is negative, the first non-identical byte in the comparison had a higher ASCII value in the second string. If the returned value is 0, the strings were identical.

NOTES

* The byte comparisons are unsigned.
* Both strings must be NULL-terminated.

* RUN/C keeps track of strings in memory via pointers which are maintained internally. A string begins at the first byte pointed to and ends when a NULL bytes (ASCII 0) is encountered.

RETURNS 0, if strings match.

Greater than 0, if first non-matching byte in the *a* string is greater than byte in same position in the *b* string.

Less than 0 (a negative number), if first non-matching byte in the *a* string is less than byte in the same position in the *b* string.

EXAMPLE

```
main()      /* strcmp.c -- illustrates string
               compare strcmp() function */
{
     char s[200], t[200];
     int i,j,k;

     puts("Please type:");
     gets(s);
     puts("& some more:");
     gets(t);
     puts("Thanks");
     i = strcmp(s, t);
     if (i == 0)
     puts("You typed the same thing twice!");
}
```

SEE ALSO strcat, strcpy, strlen

PURPOSE Copies one string to another.

FORM
```
char *to;
char *from;

strcpy(to, from);
```

USAGE **strcpy** copies a zero-terminated source string to a destination string. In the function call to **strcpy**, the first argument is the pointer to the destination string. The second points to the source string. **strcpy** does not get a length parameter for either the source or target string, therefore the source string up to its zero-byte terminator is copied unconditionally. The target string is automatically NULL-terminated.

In the above sample:

`char *to;`
declares a pointer to the target string.

`char *from;`
declares a pointer to the source string.

`strcpy(to, from);`
copies the source string to the target string.

NOTES There is no checking to prevent the source string from overwriting the target string, in which case the extra data from the source string will overwrite whatever in memory follows the destination string. Use **strlen** to avoid such an unhappy event.

RETURNS Nothing.

EXAMPLE
```
main()      /* strcpy -- demos string copy
               function: strcpy() */

    {
    char str1[20];

    strcpy(str1, "Copy cat, copies cat");
    printf("%s", str1);
    }
```

SEE ALSO strcat, strcmp, strlen

PURPOSE Measures length of a string.

FORM
```
char *s;
int length;
length = strlen(s);
```

USAGE **strlen** counts the number of bytes in a character string up to but not including the NULL terminator.

RETURNS Number of bytes in string before encountering a zero-valued byte.

EXAMPLE
```
main()      /* strlen.c -- illustrates strlen(),
               string length function */
     {
     int length;
     char *string;
     string = "12345678984823467823847628";
     length = strlen(string);

     printf("There are %d digits in my number --
     count 'em!\n", length);
     }
```

SEE ALSO strcat, strcmp, strcpy, sprintf, sscanf

PURPOSE Creates a definition for several variables or collections of variables which can then be referenced and manipulated as a group.

FORM

[1]
```
struct bills_due{
        float AmEx;
        float MC;
        float Visa;
        } acct1, acct2, acct3;
```

[2]
```
struct {
        int a;
        char b;
        float c; } v1, v2, *v3;
```

[3]
```
struct royalty_schedule{
char author[30];
char *address;
struct person manager;
float royalty_due;
}

(statement(s);)

struct royalty_schedule royalties[80];
```

USAGE **struct** initializes a data structure which holds an image or definition of a grouping of data items. In a sense, **struct** can be thought of as a file drawer containing potentially different sized letters and envelopes which are physically related to each other by virtue of the fact that they are all in the same drawer. Structures are comparable to the "records" of Pascal.

structs and **unions** are collectively known as aggregates.

In Form [1] above:

`struct bills_due{`
declares a structure definition, giving it the tag name **bills_due.**

`float AmEx;`
declares the floating point variable **AmEx** to be a member of the structure.

`float MC;`
declares **MC** to be a member.

`float Visa;`
declares **Visa** to be a member.

`} acct1, acct2, acct3;`
declares variables **acct1, acct2, acct3** of type **bills_due.**

In Form [2] above:

`struct {`
declares a structure definition.

`int a;`
declares integer **a** to be in the structure.

`char b;`
declares **char b** to be in the structure.

`float c;} v1, v2, *v3;`
declares floating point **c** to be in the structure. Variables **v1 and v2** are declared to be of the data type of the **struct** just defined and **v3** is declared to be a pointer which can be used to point to a data object of the same **struct** type. Because no *tag name* is given to the structure, the structure definition must be used immediately.

In Form [3] above:

`struct royalty_schedule{`
declares a structure template, giving it the tag name **royalty_schedule.**

`char author[30];`
declares an array 30 character lengths long called **author** to be a member of the structure.

`char *address;`
declares a character pointer (ultimately to point to some address string) to be a member of the structure.

`struct person manager;`
declares the item **manager**, which is of type **struct person** (the structure **struct person** must be defined beforehand), to also be a member of this structure.

`float royalty_due; };`
declares the floating point item **royalty_due** to be a member of the structure.

..
additional statements may be executed.

`struct royalty_schedule royalties[80];`
declares an array of 80 elements each of type:

`struct royalty_schedule.`

Each element will contain the information on one author.

Anatomy of structures
Following the keyword **struct** a *tag name* for the structure is often given. In fact, a tag must be given if you plan to use the structure definition later in the program. The components which make up the structure's definition or *template* are then declared within curly brackets. These are the *members* of the structure. Following the closing curly bracket there may be one or several variable names separated by a comma(s). Whether variables are declared or not, the definition of the structure will not be considered complete until a semi-colon is encountered following the closing curly bracket.

It may be useful to think of a structure as a data type which through some genetic engineering grew big enough to hold other data type -- even other structures. From the inside, a **struct** can be viewed as a collection of variables of (possibly) different data types, but from the outside, a **struct** appears to be a single unit whose parts can be referenced.

A key concept in understanding a structure lies in the fact that once a structure is created, it becomes a new type of data element. Structures may be identified by their *tag names* and/or by the items, examples or

instances of the **struct**, which are declared immediately following the closing brace -- or later with references to the **struct** *tag*.

Variables are not really *assigned* to structures any more than variables are assigned to **ints**, **longs** or **floats**. Instead, each variable attached to a structure causes the structure template to be "cloned" within memory.

This helps explain why the example of the employee record is used so often in describing structures. Each employee's record will contain various items of information of varying type and size (age and social security number are just two items). However, the structure becomes useful only when variables are assigned to this newly created data type, creating definitions or records for each employee.

NOTES

* Structures are among the most advanced features of C. Experimentation with an interpreter in conjunction with one or several good books on C should be useful.

* The closing semicolon after the closing brace (}) is required, even if there are no instances declared at the end of the **struct** definition.

* Keep in mind the distrinction between a *definition* and a *declaration*. An aggregate's *definition* establishes a template which can be used subsequently in the *declaration* of objects which will be instances of the aggregate. Note also that the declaration may be part of the same statement as the definition, as with:

```
struct COMPLEX{
        float real;
        float imaginary;} complex_a ;
```

where **COMPLEX** is defined and **complex_a** is declared.

Space is allocated for a structure only when variables are declared which depend on the **struct** definition. Otherwise the **struct** merely defines the form of the data structure. Of course, space can be claimed later if a tag name was used when the structure was declared. An example of this is

shown in Form [3], above.

* Structure members may be referenced and altered directly (using the . operator) or with pointers (using the -> operator). Both these operators are used in the example below.

* Recursive use of definitions is allowed only in that a structure may contain as a member a pointer to (or pointer to pointer to) an example of the structure.

LIMITS RUN/C provides a limited amount of storage (3000 bytes as of this writing) for structure and union definitions. How quickly this space is consumed depends not only on the total number of aggregates but also on the complexity of the average aggregate used. if you encounter this limit (error #272) with real -- not fantastic -- programs, please let us know.

EXAMPLE

```
/* Demonstrates getting data into and out of
structure members using . and -> operators. */

struct house {          /* struct.c */
     int foundation;
     long walls;
     double roof;
     char address[20];      };

main()
{
struct house h, *hp;
h.foundation = 999;
h.walls = 111111L;
h.roof = 23.21e3;
strcpy(h.address,"223 Sunny Place");
printf("Foundation:  %d,  walls:  %ld,  roof:
%.2f\n",h.foundation,h.walls,h.roof);
printf("Address: %s\n",h.address);
hp = &h;
printf("Foundation:  %d,  walls:  %ld,  roof:
%.2f\n",hp->foundation,hp->walls,hp->roof);
strcpy(hp->address,"225 Sunny Place");
printf("Address: %s\n",hp->address);
}
```

SEE ALSO union

PURPOSE Transfers control to one of several parallel **case** statements if there is an exact match between the **switch** value and a **case** constant, or to a **default** statement, if one exists.

FORM

```
char c;
c = 'z';

{
switch(c) {
     case '0':
          (expression1;);
          break;
     case 'z':
          (expression2;);
          break;
     default:
          (expression3;);
     }
```

USAGE **switch** compares an integer-type (**char**, **short** or **int**) value with a series of **case** statements. If an exact match is found, the statement following the matched case is executed. In addition, all subsequent statements within **switch** will be executed unless and until a **break** statement is encountered (see example below for an illustration of this concept). If no match is found, **switch** will either execute the **default** statement, if there is one, or leave the **switch** statement without taking any action.

In the above sample:

`char c;`
: declares a global character variable.

`c = 'z';`
: assigns 'z' to that variable.

`switch(c) {`
: calls the switch statement keyed to the value of **c**.

`case '0':`
`(expression1);`
`break;`

if **c** equals 0, execute *statement1*, then break out of **switch.** If **c** does not equal 0, fall to **case 'z'.**

```
case 'z':
(expression2);
      break;
```

if **c** equals **'z'**, execute **statement2,** then break out of switch. If **c** does not equal **'z'**, fall to **default.**

```
default:
    (expression3);
          }
```

since **c** is not equal to any of the previous cases and since there has been no intervening **break** between a matched case and the default case, execute **statement3.**

NOTES

* The statements **switch, case** and **default** are used only in conjunction with one another.

* statements other than **case** and **default** such as **if, else, while** and **for** can be used within individual **case** statements.

RETURNS Nothing.

(Continued next page)

EXAMPLE

```
main()     /* switch.c -- illustrates use of switch, case,
           default and break statements */
{
char c;
puts("Type in 1, 2 or 3 or any other character");
c = getchar();
switch (c)
     {case '1':
     puts("\nYou're in case 1");
     case '2':
     puts("\nYou're in case 2");
     break;
     case '3':
     puts("\nYou're in case 3");
     break;
     default:
     puts("\nYou pressed something other than 1, 2 or 3");}
puts("End of switch");
}
```

SEE ALSO case, break, default

PURPOSE Unconditionally returns control to the operating system.

FORM SYSTEM

USAGE SYSTEM clears the screen before it returns to the operating system. Any program in memory will be lost unless AUTOSAVE is active, in which case the program in memory will automatically be saved to file PROGRAM.C on the default drive/directory before RUN/C is terminated.

NOTES In RUN/C Professional SYSTEM can also be used to return from the I(Interactive) mode to the mini-menu.

EXAMPLE

```
SYSTEM

A>
```

SEE ALSO AUTOSAVE, SHELL, I(Interactive), Section 6

PURPOSE Sets the number of columns taken by each tabular space in the program listing.

FORM TAB n

USAGE TAB sets the number of spaces the cursor will move forward when the <tab> (-->) key is pressed. TAB affects text being written or edited in either the Insert or Overwrite text editing modes.*

After the tab spacing is changed, the next listing of your program will show tabs adjusted for the new tab setting.

NOTES

* Changing the TAB setting will not effect program output; i.e., tab positions within the running program.

* Tabs may range from every 2 to every 12 columns. The default tab setting is 5 column spaces. TAB with no argument will set tabs every 2 spaces.

EXAMPLE

```
TAB 3

Tabs now set every 3 columns

Ok
```

SEE ALSO EDIT

* In Overwrite mode (described in detail in Section 2), <tab> will move the cursor forward over the logical line. In Insert Mode <tab> will insert n spaces up to the next tab stop. This system is harder to explain than it is to learn to use. After a few minutes practice using the <Ins> key toggle in combination with <tab>, all will (we hope) become clear.

PURPOSE Calculates the tangent of a number.

FORM

```
double result;
double number;

result = tan(number);
```

USAGE **tan** returns the trigonometric tangent of a number expressed in radians. Both the called number and the returned value are expressed as **doubles.**

NOTES In RUN/C Professional this and other transcendental math functions are available in the loadable library MATHLIB.

RETURNS Tangent of a number.

EXAMPLE

```
main()      /* tan.c -- illustrates tan(), the
               tangent function */
{
double x;
double result;

    x = 85;

    result = tan(x);
    printf("%f", result);

/* correct answer is 0.178870 */
}
```

PURPOSE Calculates the hyperbolic tangent of a number.

FORM

```
double result;
double number;

result = tanh(number);
```

USAGE **tanh** returns the hyperbolic tangent of a number. Both the called number and the returned value are expressed as doubles.

This can be expressed algebraically as:

$$\tanh(x) = \frac{e_x - e_{-x}}{e_x + e_{-x}}$$

NOTES In RUN/C Professional this and other transcendental math functions are available in the loadable library MATHLIB.

RETURNS Hyperbolic tangent of a number.

EXAMPLE

```
main()

/* tanh.c -- illustrates hyperbolic tangent
function, tanh() */

{

double x;
double result;

     x = 6;

     result = tanh(x);
     printf("%.8f", result);
/* correct answer is: 0.99998771 */
}
```

PURPOSE Converts an uppercase character to lowercase.

FORM

```
char c = 'A';
char u;

u = tolower(c);
```

USAGE **tolower** converts to lowercase any uppercase alphabetic character. If **c** is not an uppercase alphabetic character, the character's original value is returned.

RETURNS Value of lowercase character, if the character is a letter.
Value of input character, otherwise.

EXAMPLE

```
main()
/* tolower.c -- illustrates tolower() */
{

    int a, c;
    printf("Type any upper case character: ");
    a = getchar();
    c = tolower(a);

    if(a > 'Z' && a <= 'z')
    printf("\nFor your information, %c in
    lower case is %c.", a, c);
    else
     printf("\nHey bud, that wasn't
    uppercase!\n");
}
```

SEE ALSO toupper

PURPOSE Converts a lowercase character to uppercase.

FORM

```
char c = 'a';
char u;

u = toupper(c);
```

USAGE **toupper** converts to uppercase any lowercase alphabetic character. If **c** is not a lowercase alphabetic character, the input value of the character is returned unchanged.

RETURNS Value of uppercase character, if the character is a letter.
Value of input character, otherwise.

EXAMPLE

```
main()
/* toupper.c -- illustrates toupper() */
{

     int a, c;
     printf("Type any lower case character: ");
     a = getchar();
     c = toupper(a);

     if(a > 'Z' && a <= 'z')
     printf("\nFor your information, %c in upper
     case is %c.", a, c);
     else
      printf("\nBeg pardon, but that was not
     lowercase!\n");
}
```

SEE ALSO tolower

PURPOSE Traces all variable values during the execution of a program.

FORM TRACE

USAGE TRACE follows the current values of all declared variables through the program as it runs. TRACE is a toggle. When TRACE is first called the message:

```
Trace ON
```

will be displayed. The second call to TRACE will display the message:

```
Trace OFF
```

NOTES

* TRACE's displays may be stopped and restarted with the <Ctrl-s> system toggle or may be viewed one step at a time when single stepping (SET STEP ON) is active.

* <Ctrl-c> remains active to interrupt program execution.

* TRACE prints the value of each variable when it is referred to in the program. However, any value modified by the program itself will not be shown by TRACE until that variable is referred to again. The first reference to any variable will probably display a garbage value.

* In RUN/C Professional TRACE may be turned on or off as often as desired within the executing program using the **runc()** function with:

```
runc("TRACE");
```

In is necessary to keep track of the active or inactive status of TRACE prior to the execution of the command, since TRACE is a toggle command.

EXAMPLE LIST

```
main()

{
int i,a;
i = 1;
a = 10;

do
     printf("i = %d a = %d\n", i++, a--);
while (i <= 5);
}

Ok

TRACE

Trace ON

Ok

RUN

          i: int 0

          a: int 0

          i: int 1

          a: int 10
     i = 1 a = 10

          i: int 2

          i: int 2

          a: int 9
     i = 2 a = 9

          i: int 3

          i: int 3

          a: int 8
     i = 3 a = 8
```

(Continued next page)

```
                    i: int 4

                    i: int 4

                    a: int 7
              i = 4 a = 7

                    i: int 5

                    i: int 5

                    a: int 6
              i = 5 a = 6

                    i: int 6

Ok

TRACE

Trace OFF

Ok
```

SEE ALSO PRON, TRON, Dump, Ldump

PURPOSE Turns program line trace (TRON) off.

FORM TROFF

USAGE TROFF turns off the TRON program line trace facility.

NOTES In RUN/C Professional TROFF can be called from the executing program using the runc() function.

EXAMPLE

```
Ok

TROFF
Ok
```

SEE ALSO TRON

PURPOSE Displays the execution of program statements combined with program output, if any.

FORM TRON

USAGE TRON is used to analyze or "debug" programs. When active, TRON prints each line and autoline number of the program as it is called. When the program calls for screen output (i.e., a **puts** statement), the output is displayed just below the source code line that called it. TRON is useful in analyzing program loops which repeatedly call the same program lines. It is also good for debugging C source code containing macros, since when the program line is listed, any macros are shown in their *expanded* form. The TRON command is turned off by the TROFF command.

NOTES In RUN/C Professional **TRON** can be called from the executing program using the **runc()** function as in:

```
runc("TRON");
```

EXAMPLE

```
LIST

1:
2:#define 10 MAX
3:main()
4:{
5:   int i,a;
6:   i = 1;
7:   a = MAX;
8:
9:   do
10:        printf("i = %d a = %d\n", i++, a--);
11:  while (i <= 5);
12:}

Ok

TRON

Ok
```

(Continued next page)

RUN

```
   1:
   2:
   3: main()
   4: {
   5: int i,a;
   6: i = 1;
   7: a = 10;
   8:
   9: do
  10:   printf("i = %d a = %d\n", i++, a--);
  11: while (i <= 5);
  12: }
  10:   printf("i = %d a = %d\n", i++, a--);
i = 1 a = 10
  11: while (i <= 5);
  10:   printf("i = %d a = %d\n", i++, a--);
i = 2 a = 9
  11: while (i <= 5);
  10:   printf("i = %d a = %d\n", i++, a--);
i = 3 a = 8
  11: while (i <= 5);
  10:   printf("i = %d a = %d\n", i++, a--);
i = 4 a = 7
  11: while (i <= 5);
  10:   printf("i = %d a = %d\n", i++, a--);
i = 5 a = 6
  11: while (i <= 5);
```

Ok

(In the above program, which increments one variable as it decrements another, screen output is printed on the far left, while the lines which are being traced are slightly indented. Notice how frequently lines 10 and 11 are executed. Also notice how the macro MAX is expanded to its value 10 when the program in run under TRON control.)

SEE ALSO TROFF, PRON, TRACE, Dump, Ldump, #define

PURPOSE Displays on the screen contents of a disk file without modifying or removing the program currently in memory.

FORM TYPE *filename*

USAGE **TYPE** allows the listing of a program on disk without having to clear the program currently in memory or to execute a **SHELL** command. The primary purpose of **TYPE** is to allow the immediate examination of files which are not within the RUN/C system.

NOTES In RUN/C Professional **TYPE** can be called from the executing program using the **runc()** function.

EXAMPLE

```
Ok

LIST

1:main()
2:{
3:   printf("Hiya, kids!\n");
4:}

TYPE TEST.FIL

1
2
3
 ..
30
Done!

Ok
```

SEE ALSO NLIST, SHELL

PURPOSE Provides a means for creating new names for existing date types, including aggregates.

FORM

```
[1]  typedef int COUNTER;

[2]  typedef char LINE[80];

[3]  typedef char *LINE_PTR, *COLUMN_PTR;

[4]  typedef struct
          {int monday;
          int tuesday;
          int wednesday;
          int thursday;
          int friday;
          }     DAY;
```

USAGE **typedef** is a keyword which allows the creation of aliases or synonyms for *data types*. No new data types are created (as they are by **union** and **struct** aggregates), however the redefinition facilities are powerful enough to put a new perspective on uses of data types, particularly data type pointers and arrays.

typedef can be expressed in the form:

directive [data type] [new_name]

where the *directive* is **typedef**, the *data type* is either a simple C data type (**char**, **int**, etc.) or an aggregate data type (a **union** or a **struct**). The *new_name* can be any new name, but should not conflict with existing keywords, tagnames or functions within the program.

The form of **typedef** is the same as that of a variable declaration, merely preceded by the keyword **typedef**. The new data type occupies the same position in a **typedef** statement that the variable name does in a declaration.

typedef names (those entities created by **typedef** statements) may be either global or local in scope, and will be "eclipsed" by local variables that duplicate their names.

In the above samples:

[1]
```
typedef int COUNTER;
```
Creates an alias for **int** named **COUNTER.** Subsequently integer variables can be declared such as:

```
COUNTER outer = 100;
```

[2]
```
typedef char LINE[80];
```
Creates a synonym for an 80-element character array named **LINE.**

[3]
```
typedef char *LINE_PTR, COLUMN;
```
Creates an alias for a character pointer named **LINE_POINTER,** and an alias for **char** named **COLUMN.**

[4]
```
typedef struct PERIOD
          {int monday;
          int tuesday;
          int wednesday;
          int thursday;
          int friday;
          }     WEEK;
```
Creates an alias for a **struct** template. In this case both **WEEK** and **PERIOD** are nearly interchangeable. This permits constructions such as:

```
WEEK workers[50];
```

as an alternative to:

```
struct PERIOD workers[50];
```

Discussion

It is most basic and most common usage, **typedef** has many similarities to the **#define** preprocessor in that its primary purpose is to create more "readable" and potentially more "portable" C programs. Also like **#define, typedef** makes it possible to create new names for data types which are easier for the programmer to input and for the reader of the program

to recognize.

However, **typedef** is not a preprocessor command; it is capable of, in the words of K&R, coping "with textual substitutions that are beyond the capabilities of the C macro preprocessor." To illustrate the shortcomings of preprocessor definitions (and the reason for **typedef**), you can:

```
#define METACHAR char *
```

if you use **METACHAR aaa;** later in the program the preprocessor will replace it with **char *** and you will get the character pointer:

```
char *aaa;
```

through normal macro expansion. This seems to indicate that the **#define** mechanism is sufficient for declaring character pointers. However if you try to do two declarations on the same line you will see the fatal flaw of the system:

```
METACHAR aaa, bbb;
```

which will expand to:

```
char *aaa, bbb;
```

which is not what is wanted. Enter **typedef. typedef** was designed specifically and only for creating new names for data types and it solves this problem in the following way:

```
typedef char *charptr;
```

now you may say:

```
charptr aaa, bbb;
```

and **aaa** and **bbb** will both be character pointers.

NOTES

* If you have ever used file I/O in a C program you have already run across a name that was created with **typedef: FILE.** In most compiler systems the **typedef FILE** is defined in the stdio.h file and is a synonym for a structure containing all sorts of internal information about currently open files.

* With **typedef**, multiple aliases for particular data types can be created simply by separating the aliases by commas on the definition line. This is illustrated in sample No. 3, above.

* In C, **typedefs** are often used to reduce complicated type declarations which are used repeatedly to single, immediately recognizable terms.

* If a variable and a **typedef** have the same name, an error message will be generated only if 1) both are globals (outside any function), or 2) both are declared within the same block (framed by curly brackets).

* By purely optional convention, terms created with **typedef** are often capitalized.

LIMITS

* Currently **typedefs** cannot be "nested" when working with simple scalar data types. For example:

```
typedef int i;
typedef i fred;
```

is not supported. However is is possible to use a **typedef** in the **typedef**'ing of a **struct** or a **union.** Thus the following is legal:

```
typedef i int;
typedef struct {
      i aa, bb;
      } FRED ;
```

* **typedefs** that are arrays cannot be used in forming new objects that are arrays of those **typedefs.** For example:

```
typedef int[10] SOMEINTS;
```

is fine, and so is:

```
SOMEINTS a10ofint;
```

but:

```
SOMEINTS a10by10ofint[10];
```

will yield an (admittedly gratuitous) declaration error.

EXAMPLE

```
typedef char units;
typedef int *TIMEPTR;
typedef int MINUTES;

main()     /* typedef.c -- just a sample */
{
TIMEPTR mp;
MINUTES time = 30;

units a, b;
a = 'a';
b = 'z';
     printf("a is %c and b is %c\n", a,b);

mp = &time;
     printf("%d minutes to go\n", *mp);
}
```

SEE ALSO #define, struct, union, TRON, LIST

PURPOSE Pushes a character back onto the file.

FORM

```
int ret;

char c;
FILE *fp;

ret = ungetc(c, fp);
```

USAGE **ungetc** pushes back a character onto the specified input file. The **ungetc** function also decrements the byte pointer to the file so that it points to the byte that was just written back.

In the above sample:

`int ret;`
declares a variable to hold the return value.

`char c;`
declares the character to be pushed back to the file.

`FILE *fp;`
declares the file pointer.

`ret = ungetc(c, fp);`
pushes the character **c** back onto the file identified by **fp**. The value returned will be assigned to **ret**.

The returned value will be equal to the value of the character pushed back, if successful. Otherwise a -1 will be returned.

The **ungetc** function can only "put back" one character. That is a read operation must be performed on the file before another call to **ungetc** can be made.

NOTES * It may not be immediately clear why a function like **ungetc** is needed. According to K&R (p. 79) "It is often the case that a program reading input cannot determine that is has read enough until it has read too much. One instance is collecting the characters that make up a number; until the first non-digit is seen, the number is not complete. But then the program has read one character too far, a

character that it is not prepared for. The problem would be solved if it were possible to 'un-read' the unwanted character. Then, every time the program reads one character too many, it could push it back on the input, so the rest of the code could behave as if it had never been read."

* To make matters a bit more complicated, **ungetc** does not actually return the character to the file. Instead, it is pushed back onto the input stream which is maintained for a file which has been opened with **fopen** under *Level 2* (buffered) I/O and for which data has been read. In addition, **ungetc** will not be successful after the file position pointer is moved by **fseek**. Generally speaking, an **ungetc** either should be done immediately after one too many characters has been read from the file's input buffer or not at all.

RETURNS EOF, if an error has been detected.
Character, if successful.

EXAMPLE

```
main()     /* ungetc.c -- illustrates ungetc() */
/* test.fil contains 30 consecutively-numbered lines */

{
FILE *fi;
int c, r;

if((fi = fopen("test.fil","r")) == NULL) {
     printf("Can't open test.fil\n");
     exit();
     }
while ((c = getc(fi)) != EOF) {
     putchar(c);
     if (c == '0' && r != '0')
          r = ungetc(c,fi);
     else
          r = EOF;
     }
fclose(fi);
printf("\n...Correct answer using TEST.FIL: ");
printf("\n1\n2\n3\n4\n5\n6\n7\n8\n9\n29\netc.");
}
```

SEE ALSO ungetch, getc

PURPOSE Sends a character just captured back to the console keyboard buffer.

FORM

```
int r;
char c;

r = ungetch(c);
```

USAGE **ungetch** returns a character obtained by **getch** to the **getch** keyboard buffer. The next time **getch** is invoked, the **c** will again be captured.

This function is useful in testing for a particular character (such as <Ctrl-c>) and, if it is not found, returning it to the keyboard buffer to be "reused" later.

NOTES As with **ungetc**, only one character may be pushed back.

RETURNS **EOF**, if the character has already been pushed back. Character just captured, if successful.

EXAMPLE

```
#define MAXLEN 50
main()      /* ungetch.c -- illustrates un-get
            console function ungetch() */
{
     char c, a[MAXLEN];
     int i;
     puts("Type something, please:");
     i = 0;
     while ((c = getch()) != '\n' && i < (MAXLEN
     - 1)) {
          a[i++] = c;
          if (c == 'e')
               ungetch(c);
     }
     a[i] = '\0';
     puts("");
     puts(a);
     puts("Correct output: Your input up to the
     first e, then all e's");
}
```

SEE ALSO getch, ungetc

PURPOSE Provides a single data area which can hold multiple members. As such, a **union** allows manipulation of different kinds of data within a single storage area.

FORM

```
union data_masseuse{
        int int_value;
        float float_value;
        char char_value;
        char array_tst[80];
        } univ_value;
```

USAGE **unions** are used to conserve memory by allowing the same memory space to hold a variety of data types. When a **union** is defined it will be large enough to accommodate the largest member in the **union**; it follows that *members* of a **union** share a common storage area. **structs** and **unions** are collectively known as aggregates.

In the above sample:

`union data_masseuse{`
declares a union with the tag name **data_masseuse.**

`int int_value;`
declares a member of size **int.**

`float float_value;`
declares a member of size **float.**

`char char_value;`
declares a member of size **char.**

`char array_tst[80];`
declares an array of 80 **chars.**

`} univ_value;`
The variable **univ_value** is declared to be an instance of the **union** just defined.

In this case it is easy to tell that the size of the **union** in the sample is 80 **chars** (80 bytes) since the character array is by far the largest number, but in case of any doubt the **sizeof** function could be used:

```
size = sizeof(union data_masseuse);
```

or

```
size = sizeof(univ_value);
```

where **size** is an unsigned integer (recall that in RUN/C a single data object cannot exceed 64k).

Following the declaration of a **union**, a *tag name* for the **union** may be given. A tag must be given if a **union** *definition* is to be used again elsewhere in the program. The components which make up the **union**'s definition are then declared within curly brackets. These are the *members* of the **union.** Following the closing curly bracket there may be one or several variable names separated by a comma(s). Whether variables are declared or not, the declaration of the **union** will not be considered complete until a semi-colon is encountered following the closing curly bracket.

It is perhaps useful to think of a **union** as a shoe box designed to accomodate the largest object likely to be put in it -- perhaps a pair of size 15 EEEEE shoes.

A key concept in understanding a **union** lies in the realization that once a **union** is created, it, like a **struct**, becomes a new type of data element. **unions** may be identified by their tag and/or by the examples which are declared immediately following the closing brace -- or later, with reference to the **union** definition.

Variable names are not really "assigned" to **unions** any more than variable names are assigned to **ints, longs** or **floats.** Instead, each variable attached to a **union** causes the **union** be "cloned" within memory.

Assigning values to union members
Items within **union** definitions can be assigned values directly, as in (from the sample):

```
data_masseuse.float_value = 2.23;
```

which will assign the floating point number 2.23 to the **union** with the tag name **data_masseuse.**

NOTES

* **unions** are among the most advanced features of C. Experimentation with an interpreter in conjunction with one or several good books on C is recommended.

* The semicolon after the closing brace (}) is required, even if there are no examples declared at the end of the **union** definition.

* It is up to the programmer to keep track of which member is currently occupying the **union's** data space.

* Theoretically space is allocated for a **union** only when variables are declared which depend on the **union** definition to define the amount of data space they will occupy. Otherwise the **union** merely defines the size of the data structure. Of course, space can be claimed later if a *tag name* was used when the **union** was declared.

* **union** members may be referenced and altered directly (using the aggregate member-operater (.)) or with pointers (using the -> operator).

* Recursive use of definitions is allowed only in that a **union** may contain as a member a pointer to (or pointer to pointer to, or ...) an example of the **union**.

LIMITS

RUN/C provides a limited amount of storage (3000 bytes as of this writing) for **struct** and **union** definitions. How quickly this space is consumed depends not only on the total number of aggregates, but also on the complexity of the average aggregate used. if you encounter this limit (error #272) with real -- not fantastic -- programs, please let us know.

(Continued next page)

EXAMPLE

```
/* union.c -- Shows how union members interact.
*/

union {    char delaware;
           int  iowa;
           long california;
           float texas;
           double alaska;
      } USA;

main()
{     USA.alaska = -999.0;
      USA.delaware = 'D';
      printf("%c\n",USA.delaware);
      USA.iowa = 99;
      printf("%d\n",USA.iowa);
      USA.california = 1111111L;
      printf("%ld\n",USA.california);
      USA.texas = 1.23456e9;
      printf("%f\n",USA.texas);
      printf("%lf\n",USA.alaska);
      printf("%c\n",USA.delaware);
      printf("%d\n",USA.iowa);
      printf("%ld\n",USA.california);
}
```

SEE ALSO struct

PURPOSE Deletes a specified file from the specified drive/directory.

FORM UNLINK *FILENAME.EXT*

USAGE **UNLINK** deletes a file from the disk. The **KILL** and **UNLINK** commands are similar to **DEL** and **ERASE** in DOS. The exact name of the file (primary name and extension) must be supplied. Standard MS-DOS file and path naming conventions apply. (For more information on filenames and pathnames see Section 1.)

If the file is not found on the specified drive/directory the message:

```
Cannot remove filespec
```

will appear.

Use of quotation marks around the file name is optional. But a space after **UNLINK** is mandatory, likewise with **LOAD, SAVE** and **KILL.**

NOTES In RUN/C Professional **UNLINK** can be called from the executing program using the **runc()** function as in:

```
runc("unlink");
```

EXAMPLES [1] UNLINK \cprogs\oldfiles\try1.c

```
Ok
```

[2] UNLINK "GLASSES"

```
Cannot remove glasses
```

SEE ALSO KILL, unlink

PURPOSE Deletes a file from the specified drive/directory from within a program.

FORM

```
int ret;
char *name;

ret = unlink(name);
```

USAGE **unlink** erases the named file from a disk. The specified file must not be currently open. All data in the file is lost.

In the above sample:

`int ret;`
declares an integer to hold the returned value.

`char *name;`
declares an array to hold the name of file to be erased.

`ret = unlink(name);`
erases the named file, assigning any returned value to **ret**.

If the operation is successful, a 0 will be returned; -1 indicates a failed effort. The most likely reason for an error is that the named file does not exist on the specified drive/directory.

NOTES **unlink** should be used with care since the file, once removed, is generally not retrievable.

RETURNS -1, if an error or end-of-file is detected.
0, if successful.

EXAMPLE

```
main()      /* unlink.c -- demonstrates file
               deleting program */

{
char victim[25];

printf("Name of disposable file to erase? ");
gets(victim);
printf("%d\n", unlink(victim));
puts("Correct output: 0 if file existed, -1 if it
did not");
}
```

SEE ALSO fopen, exit, fclose, UNLINK

PURPOSE Creates a positive integer variable 16 bits in length (range 0 to 65,535).

FORM `unsigned a = 60000;`

USAGE **unsigned** allows use of the "high-bit" for a numeric value, rather than to hold a sign (positive or negative) indicator. Thus all numbers declared **unsigned** must be positive. **unsigned** is only used with whole numbers.

NOTES

* **unsigned** data types are often used to keep track of such housekeeping items as available memory since such items are generally best expressed as positive numbers.

* In C, simple data type names (such as **unsigned**) have four distinct uses, namely:

  1) in declaring variables and aggregate numbers;

  2) in casting one type of value to another such as:

```
d = (unsigned) 0;
```

  3) as an argument to **sizeof**, as in:

```
int i = sizeof(unsigned);
```

  4) as an element in a **typedef** statement used in creating a new data type or as an alias for an existing data type such as:

```
typedef unsigned POSITIVE;
```

* The construct **unsigned int** is not supported.

EXAMPLE

```
main()     /* unsigned.c illustrates %u */
     {
     int a = 40000;
     unsigned b = 40000;
     printf("a equals %d; b equals %u", a, b);
}
```

SEE ALSO `char, double, float, int, long, short`

Interactive mode debugging command

PURPOSE Provides a means for determining the current value of any variable.

FORM

[1] >> VAL a

[2] runc("val *variable*");

USAGE VAL is a RUN/C Professional debugging command which enables the determination of the value of any variable at any point in program execution. (VAL is also described in Section 6; portions of that discussion may not be repeated here.) It can be used with the **runc()** function as in:

```
runc("val aa");
```

where it will print the pertinent information about **aa** as the program runs. Or VAL can be used interactively after the I(Interactive) mode has been invoked from the mini-menu, as shown in sample 1.

Generally speaking if a:

```
runc("BREAK");
```

command has been executed and the I(Interactive) mode selected from the mini-menu, the value of any active variable can be determined. For example, if there is an **aa** integer variable in the system the following interactive mode commands will provide information about it while the program is in an interrupted state:

```
>>VAL aa
Automatic variable of current function: aa: int 10
>>
```

will reveal the type, location and value of the variable.

If you want to test what your program would do if a suddenly developed a case of negativitis you could artificially change its value with the command:

```
>>@ a = -1;
>>
```

And then continue with program execution by pressing <enter> again and selecting the C(Continue) option from the mini-menu. The program will resume at the breakpoint, but the value of a will be -1.

As shown in example 2, below, VAL can provide information about a variable as the program goes through its paces. Information about **struct** and **unions** members, pointers, arrays, file pointers, etc. can be obtained.

EXAMPLE [1]

```
>>

val upper

Automatic variable of current function: upper: int 300
>>@ upper = 30;

>>val upper
Automatic variable of current function: upper int 30
>>
```

(Continued next page)

[2]

```
main() /* VAL + runc() demonstration program */
{
int a, b, c;
for(a=1, b=10; a < 10; ++a, --b){
        c = mult(a,b);
        printf("C = %d\n", c);
        }
}

mult(a,b)
int a, b;
{
int c=0;
c = a*b;
printf("info from called function: "); runc("val c");
}
```

SEE ALSO runc(), BREAK, Interactive, Section 6

PURPOSE Displays the RUN/C version information and creation date.

FORM VER

USAGE VER displays the version number of the currently running RUN/C and the date the version was created. This is helpful in case you get an update and want to make sure you are running the right version of RUN/C.

When VER is invoked a message similar to:

RUN/C Version n.nn. Created Month nn, 198n.

will appear.

NOTES In RUN/C Professional **VER** can be called from the executing program using the **runc()** function.

EXAMPLE VER

```
RUN/C Version 1.20. Created April 18, 1985.
```

PURPOSE Declares a function which does not return a value.

FORM void funcl(x,y);

USAGE The primary use for **void** is as a reminder to the programmer that a referenced function is not expected to return a value.

EXAMPLE

```
main()
{
int n;
     testeven(5);
     n = testeven(14);
     printf("%d", n);
}

void testeven(n)
int n;
{
     if(n % 2 == 0)
          printf("%d is even\n",n);
     else
          printf("%d is odd\n",n);
     return(n);
}

/*
the error message, stack underflow, results
from the fact that a function of type void
cannot return a value, thus the attempt to
initialize n with a function which returns
nothing results in an error indicating an
attempt to return a value from a void function
 */
```

SEE ALSO: return

PURPOSE Executes a statement or series of statements as long as the test expression is true.

FORM

```
while(condition(s))
     statement(s);
```

USAGE **while** sets an operation in motion which will continue as long as a specified condition(s) is true. The condition is evaluated before the statement or statements controlled by the **while** construction are executed. Thus, if the condition is initially false, none of the statements will be executed. "True" has a numeric value of non-zero in C; "false" is 0.

NOTES One common error is not providing an opportunity for the **while** to ever get a false value. In such a case the loop will continue indefinitely or until you type a <Ctrl-c> or reboot the system.

RETURNS Nothing.

EXAMPLE

```
main()     /* while.c -- illustrates use of while
              statement */

{
int a = 0;

while(a < 10){
     ++a;
     printf("%d ", a);
     }
putchar(7);
puts("\nYou have reached your goal!");
}
```

SEE ALSO do

PURPOSE Writes data to a file.

FORM

```
int ret;
int file;
char *buffer;
int n;
return = write(file, buffer, n);
```

USAGE **write** is a *Level 1* (unbuffered) I/O function, used by other functions such as **putchar** to save data to a disk or other devices. **write** can be used only on a file which has been opened with **open** or **creat.**

In the above sample:

`int ret;`
declares a variable to hold the return code.

`int file;`
declares a file number.

`char *buffer;`
declares an output buffer.

`int n;`
declares an integer to hold the number of bytes requested.

`ret = write(file, buffer, n);`
writes the number of bytes specified by **n** from **buffer** to **file.** The value returned will be assigned to **ret.**

The return count will be equal to the number of bytes written to the file, which may be less that the number called for by **n** if an EOF is encountered. The file position is advanced to the end of the data written to it. -1 will be returned if an error is detected.

NOTES

* The number of bytes by which the file position is advanced may not equal the number of bytes transferred if *text mode translation* is active.

* n may be a positive or negative number. If no minus sign is present, positive will be assumed.

RETURNS -1, if end-of-file or other error has been detected.
Number of bytes actually written, if successful.

EXAMPLE

```
main()
     /* write.c -- reads, then writes contents
     of file */

{char in_name[20];
char out_name[20];
char buf[1];
int fin, fout;
int i;

     printf("Input filename please: ");
     gets(in_name);
     printf("Output filename please: ");
     gets(out_name);
     if ((fin = open(in_name,0)) == -1)
     {    printf("No such file: %s\n",in_name);
          exit();
     }
     if ((fout = creat(out_name,1)) == -1)
     {    printf("Cannot open: %s\n", out_name)
          exit();
     }

     while (0 != (i = read(fin, buf, 1)))
          {
          if (i <0) {
               printf("error on read\n");
               close(fin);
               exit();
               }
          else if (i != write(fout, buf, i)) {
               printf("error on write\n");
               close(fout);
               exit();
               }
          }
     close(fout)
     close(fin);
     printf("Success!\n");
}
```

SEE ALSO open, read

SECTION 5. USING RUN/C LOADED LIBRARIES

Summary

A RUN/C Loadable Library is a set of files containing functions that can be used by the RUN/C interpreter just like RUN/C built-in functions (e.g. **printf**). One loadable library consists of three files: an .EXE file containing the compiled code for the functions; a .MAP file automatically generated by the MS-DOS LINK program at the time that the .EXE file is created; and an .RCL file, which is a text file created by the user, and which serves as a guide for both RUN/C and the user as to the contents (list of functions, etc.) of the compiled .EXE file.

When the user wishes to use any or all of the functions in the .EXE file, he simply loads the library (see below for the LIBLOAD command) and subsequently can use the library's functions from within his interpreted program or from RUN/C's Immediate mode (see the beginning of Section 4 for a description of the '@' command). These functions will execute at compiled speeds (just like RUN/C built-in functions).

The user can load more than one library if he/she has enough memory space for them, and can remove all libraries from memory and load others. All of these operations can be performed either from RUN/C's Command mode or from within a RUN/C interpreted program (see the **runc()** function described in Section 6 of this manual).

The user can create a loadable library from his own compiled functions, or can extract functions from a .LIB library of routines.

As a consequence, functions can be created and debugged in the interpreter. And, when ready, embedded in a library to run at compiled speeds.

Loadable Library Capabilities

1) Up to 256 functions can be embedded in a single loadable library.

2) As many libraries can be loaded as available memory permits.

3) Any RUN/C function can call any library function. Furthermore, library functions can be called via RUN/C's immediate execution mode.

4) Assembly language routines can be made into loadable libraries.

Loadable Library Limitations

1) Functions and subroutines must conform to the Lattice large model calling conventions (see assembly interface below). We are working to expand this to other large model compilers.

2) Library functions cannot call RUN/C interpreted functions, nor can the functions in one library call function in another library.

3) The global variables or *externals* in a library have no relationship to variables inside RUN/C or variables in another library.

4) While RUN/C attempts to protect the user by checking the validity of pointers to make sure that they do not trash valuable data or code, library functions cannot do this.

5) It is not possible to smoothly interrupt executing loaded library functions. If a <Ctrl-c> is pressed, the usual ^C will appear. The only way get out of an endless loop inside a loaded library function is do a warm reboot of the system. Of course if control is eventually passed back to the interpreted portion of the program, <Ctrl-c> will be available at the point.

Background

One of the virtues of the C compiler environment is its ability to access and combine function libraries developed at different times and quite possibly from different sources. While this is useful to the individual developer -- who can solve a programming problem and "reuse" the "solution" again in a somewhat different context -- it also opens the way to accessing the programming expertise of others.

In a sense, external function libraries are C "expert systems", black boxes which provide programming expertise and save weeks if not months or years of programming effort.

Even the most independent C programmer uses functions developed by others, since common C library functions such as

malloc, fopen and **printf** are not, strictly speaking, part of the C language, but of C's I/O library which in a compiler environment is only made available when the program modules are linked together to create an executable program.

Due to the increasing popularity of C, the number and diversity of commercial C function libraries for the MS-DOS environment is expanding rapidly. Now it is possible to make full use of functions contained in many of these link-time libraries within the RUN/C Professional interpreter. Or, looked at another way, library functions can now be used in real time. By real time we mean that the pre-compiled function, loaded as a library into RUN/C, will execute at or near compiled code speed. For example, the Sieve of Eratosthenes, everyone's favorite benchmark for no particular reason, runs in about 10 seconds as a RUN/C loaded library function -- at or near the speed of the best C compilers.

In describing the RUN/C loadable library system there are six major areas to be discussed:

1) Using a loaded library function.

2) Creating a loadable function library.

3) Extracting functions from .LIB or .OBJ files.

5) Assembly language interface.

6) Using the verification program CHECKRCL.EXE.

Using a Loadable Library
There are five RUN/C Professional commands which directly affect loadable libraries:

* LIB
* LIBLOAD (or LIBL)
* LIBSHOW (or LIBS)
* LIBDIR (or LIBD)
* LIBUNLOAD (LIBU)

On the following pages, using the same form we used to describe RUN/C keywords in Section 4, are descriptions of these commands.

PURPOSE Displays library command options.

FORM LIB

USAGE LIB displays the names of loadable library commands and provide a brief description of each. When the command is invoked the following is displayed:

```
Loadable Library commands:

LIBDIR [drive:][dir] (list all libraries in current or specified directory)
LIBLOAD libname          (load library)
LIBSHOW [libname]   (list given or all loaded libraries)
LIBUNLOAD          (unload all libraries)
   Abbreviations: LIBD, LIBL, LIBS, LIBU
Ok
```

SEE ALSO LIBDIR, LIBLOAD, LIBSHOW, LIBUNLOAD

PURPOSE Displays the names of the Loadable Libraries on the default or a specified drive/ directory.

FORM

```
[1] LIBDIR

[2] LIBD

[3] LIBD \subdir\sub\
```

USAGE LIBDIR, and its shorter form **LIBD**, displays the names of loadable libraries which can be accessed by RUN/C. In addition to displaying the names and sizes of all available libraries, it also displays a descriptive phrase taken from the library's .RCL file. Lastly, the amount of memory available for additional libraries is reported.

NOTES

* It is also possible to get a listing of available libraries from within a RUN/C program using the special **runc()** function as in:

```
runc("libdir");
```

* When getting a list of valid loadable libraries on directories other than the default drive directory, the listing of the libraries can take a few moments since all loadable libraries on a specified drive/directory are shown.

EXAMPLE

```
Ok

LIBDIR

Library:        TESTLIB2
incrementation of all types
memory requirements: 24 (K bytes)

Library:        TESTLIB3
type, and sizmem(allmem())
memory requirements: 28 (K bytes)

Library:        DISKSTAT
directory search, return drv & dir, filesz, drv size
memory requirements: 21 (K bytes)

8 (K bytes) left for libraries
Ok
```

SEE ALSO LIB

PURPOSE Loads an available loadable library from the default or a specified drive/directory.

FORM

[1] LIBLOAD TESTLIB4

[2] LIBL TESTLIB4

[3] LIBL \SUB\TESTLIB4

USAGE **LIBLOAD,** and its shorter form **LIBL,** loads into memory the specified loadable library. Each library, whether loaded from the default directory or a specified drive/directory, is accessed and loaded as a system; all three elements of the system must be on the same drive/directory. The three components of a loadable library are:

LIBFILE.RCL
LIBFILE.EXE
LIBFILE.MAP

where *LIBFILE* is the name of the library system which is being loaded.

LIBFILE.RCL is a text file containing descriptive information about the functions in the library and memory management information required by RUN/C. The .RCL file (described in detail below) is generated by either the creator of the library or the user.

LIBFILE.EXE is a special compiled file which can only be executed within RUN/C.

LIBFILE.MAP contains the required extended .MAP file generated by the DOS linker when *LIBFILE.EXE* was created.

If any of these three files is not available on the default or specified drive/directory the message:

```
*** Error # 1002 at line n
Bad library. Please check library with CHECKRCL
```

will appear. (The verification program CHECKRCL.EXE is described later in this section.)

NOTES

* Always run the RUN/C utility program CHECKRCL on a library before it is loaded into RUN/C for the first time. It takes very little time to use and it can save you enormous amounts of time.

* It is possible to load a library from within a RUN/C program using the special **runc()** function as in:

```
runc("libload testlib2");
```

EXAMPLE

```
Ok

LIBLOAD TESTLIB2
Library loaded

Ok

LIBLOAD \SUB\TESTLIB4
Library loaded

Ok
```

SEE ALSO LIB

PURPOSE Displays information about loaded libraries and their functions.

FORM

[1] `LIBSHOW`

[2] `LIBS`

[3] `LIBSHOW libname`

[4] `LIBSHOW function-name`

USAGE **LIBSHOW,** and its shorter form **LIBS,** display information about function library system(s) loaded into the interpreter. That information includes:

* name(s) of the loaded library(ies).
* number of available functions in each library.
* names, types and arguments of each function taken from the .RCL file.
* optional descriptive information taken from the .RCL file.
* number of kilobytes left for additional loadable libraries.

For example, if the **diskstat** library is loaded into the interpreter, the following information will be displayed when **LIBSHOW** is invoked:

```
library diskstat
5 functions
int dirnames(*,int,*)    int dir search (scratch-array, len, filespec)
int drivestr(*)          return drive number and string
void dirstr(*,int)       return dir string
long filesize(*)               return filesize
long diskfree(int)       return available space on drive

81 (K bytes) left for libraries
```

All this information is extracted from the DISKSTAT.RCL file. (This means that if the creator of the .RCL file has made an error in specifying the calling sequence or file type or in the descriptive phrase following, that error will be repeated when **LIBSHOW** is used.)

If you want information about a particular loaded library simply give the name of the library without an extension as in:

```
LIBSHOW TESTLIB
```

DOS pathnames may be used to see the contents of loaded libraries on drive/directories other than the default. If you want information only about a particular loaded function, use **LIBSHOW** in conjunction with the function name:

```
LIBSHOW add

int add(int,int)  /* integer add */
25 (K bytes) left for libraries

Ok
```

If no library is specified, information about all loaded libraries will be displayed. If a specific library is specified, only information about that library will be displayed. If the interpreter is asked to show information about a library named **LIBTEST** which, in fact, is not loaded into the system, the following message will be displayed:

```
Library or function libtest not found

nn (K bytes) left for libraries
```

If **LIBS** is called without an argument and there are no libraries loaded into the interpreter the message:

```
No libraries loaded
```

will be displayed.

NOTES You can also obtain information about loaded libraries from within a RUN/C program using the special **runc()** function as in:

```
runc("libshow");
```

EXAMPLE

```
Ok

LIBSHOW C:\sub\testlib2

Library \sub\testlib2
9 functions
char cincr(char)          /* char increment */
short sincr(short)        /* short increment */
int iincr(int)            /* integer increment */
unsigned uincr(unsigned)  /* unsigned increment */
long lincr(long)          /* long increment */
float fincr(float)        /* float increment */
double dincr(double)      /* double increment */
int add(int,int)          /* integer add */
double dadd(double,double)      /* double add */

77 (K bytes) left for libraries

Ok
```

SEE ALSO LIB

PURPOSE Releases memory which had been occupied and assigned to loaded libraries.

FORM [1] LIBUNLOAD

or

[2] LIBU

USAGE **LIBUNLOAD**, and its shorter form **LIBU** unloads all loaded function libraries, returning any memory assigned to those libraries to the interpreter.

NOTES It is also possible to unload libraries from within a RUN/C program using the special **runc()** function as in:

```
runc("libu");
```

EXAMPLE

```
Ok

LIBUNLOAD

All libraries unloaded

Ok
```

SEE ALSO LIB

Using a Loaded Library Function

After one or several function libraries are loaded the next logical step is to use the newly available functions. By far the simplest means of using a built-in or loaded library function is to use RUN/C's Immediate mode, described in a previous section of this manual. Basically, any built-in or library function can be executed from the command line by preceding the function call with an '@' (at sign) as in:

```
Ok
@ printf("hello, world");*
```

Here is another listing of the DISKSTAT library functions which were shown in the discussion of **LIBSHOW** above:

```
Ok
LIBSHOW

library diskstat
5 functions
int drivestr(*)          return drive number and string
void dirstr(*,int)  return dir string and function of drive number
long filesize(*)         return filesize
long diskfree(int)  return drive space as function of drive number
int dirnames(*,int,*)    int dir search (scratch-array, len, filespec)
```

Using the immediate mode you can easily determine the amount of free space on your default drive with the command:

```
@ printf("%ld free\n", diskfree(0));
```

The program STATS.C on the RUN/C Professional distribution disk exercises these library functions (also on the disk under the name DISKSTAT.C). Let's take one or two examples from that program:

The **filesize** function is shown in the descriptive text to return a **long** data type upon being passed a pointer. The following program fragment determines the file size of STATS.C using this function:

---

* The space after the '@' is mandatory. See the '@', the first command described in Section 4, for a complete explanation of use of RUN/C's Immediate mode; also see a discussion in Section 5 pertaining to use of the Immediate mode with other RUN/C professional debugging commands.

```
/* illustrates filesize */

long ret, filesize();
char *filename;
filename = "stats.c";
ret = filesize(filename);
if(ret == -1)
  printf("file not found\n");
else
printf("size of %s is %ld\n", filename, filesize(filename));
```

Notice how the function **filesize** is declared as returning a **long** in accordance with the type of value it is designed to return. Also note that in keeping with information supplied by the creator of the library we are passing a character pointer to the function. In this case we have set the pointer to the appropriate string. An equally valid call would be

```
ret = filesize("stats.c");
```

The function **dirstr** is described in the .RCL file as returning a **void** data type (i.e., nothing is returned) and being passed a pointer and an integer number:

```
void dirstr(*,int)  return dir string and function of drive number
```

In STATS.C the function **dirstr** is exercised with the following program segment:

```
/* illustrates dirstr */

char drive_string[3];
int drive = 3;
dirstr(drive_string,drive);
printf("Current directory for drive %d is %s\n", drive, drive_string);
```

Since **dirstr** returns a **void,** there is no special data type declaration required when the function is invoked.

In our example above, the drive is set to the "C:" drive (in DOS the 0 drive is the default, 1 is the A: drive and so on) and a character array is declared to hold the string returned from the

function. When the program is run the default directory name or subdirectory name from the specified drive will be printed.

Although the information contained in the function description line is hardly definitive, it does point the way toward understanding the calling conventions of the function. In conjunction with a hard-copy description, making use of loaded library functions should be relatively easy and requires no knowledge of the internal workings of these particular functions (which essentially use system interrupts to get their disk statistics).

Creating your own Loadable Libraries

The tools required to create a RUN/C loadable library consist of the Lattice Compiler (version 2.0 or greater) -- specifically the large model (**-ml**) version of the compiler -- and DOS' LINK.EXE program. We also assumed that you have experience with compiling and linking C programs using the Lattice compiler.

A RUN/C compatible loadable library system consists of three files residing on the same disk/directory, all with the same primary name and the extensions: an .EXE file, a .MAP file and an .RCL file. On the RUN/C Professional distribution disk you will find several examples of this including the loadable library system SAMPLE. It consists of four files: SAMPLE.C, SAMPLE.EXE, SAMPLE.MAP and SAMPLE.RCL. (The .C file was used to create the loadable library but is not required when the created library is loaded into RUN/C.)

The .EXE file is created from source code using the Lattice large-model compiler and the LINK program (which also creates the extended .MAP file). The .RCL file is a text file which contains descriptions of the library functions and specific information about the size and memory requirements of the library. The .RCL file can be created with any standard text-editor.

Five easy steps to loadable libraries

To create of a loadable library you need to:

1) Write or obtain the C source code for the functions that you wish to integrate into RUN/C. Do not write a **main()** function for your program; this is supplied at link time.

2) Successfully compile the source code with Lattice's large model (**-ml**) compiler.

3) Link your object file(s) together with two specially provided object files (RCLMAIN.OBJ and RCL2.OBJ), and produce a complete linker map using the \m option.

4) Create the .RCL file using a text editor.

5) Check the consistency of the library with the specially provided program CHECKRCL.EXE.

Once the program passes the compile, link and CHECKRCL steps successfully, it is a loadable library which can be loaded into RUN/C using **LIBLOAD**, described above. Once loaded, the functions in the library will be accessible to your C programs and

to the RUN/C Immediate mode.

Let's get into specifics. The example below contains two simple functions: **copy** and **countc** which we would like to compile and make available to the program being interpreted by RUN/C. **copy** is similar to the built-in function **strcpy** in that they both copy NULL-terminated strings, although **copy's** calling parameters are in the opposite order to **strcpy's** and **copy** returns a character pointer to the beginning of the destination character array. **countc** opens a file whose name you have provided, counts the characters in it, and returns that character count. These two functions are in a file called EXAMPROG.C. Here, in its entirety, is EXAMPROG.C:

```
#include "stdio.h"  /* we are including Lattice's stdio.h here */
char *copy(from, to)
  /* copies the null-terminated character string pointed to */
  /* by "from", to the string array pointed to by "to"; returns "to" */
char *from, *to;
{char *keepto;
    keepto = to;
    while (*to++ = *from++)
  ;
    return(keepto);
}

long countc(filename)
  /* counts and returns the number of characters in the */
  /* file "filename"; if no file, or cannot open, then returns -1 */
char *filename;
{
int c;
int count = 0;
FILE *fp;
    if ((fp = fopen(filename, "rb")) == NULL)
        return(-1);
    while ((c = getc(fp)) != EOF)
        ++count;
    fclose(fp);
    return(count);
}
```

Forget the main()
Note that the above program contains no **main()** function. As pointed out above, the files you compiled to create loadable library modules do not contain a **main()**. The **main()** is supplied by RUN/C's RCLMAIN.OBJ file at link time.

Compilation and linkage
Compilation and linkage for EXAMPROG.C is accomplished with the following commands:

```
lc1 examprog -ml
lc2 examprog
link c1+rclmain+rcl2+examprog, examprog, examprog/m, lcl
```

Notice the "**/m**" flag in the third part of the LINK command. This directs the linker to produce a full linker-map listing, which is essential to the correct functioning of loadable libraries. Warning: if the "**/m**" is left off, then an incomplete (the so-called 'summary') link map listing will be produced and various error messages will appear if an attempt is made to load the library or to check it with CHECKRCL.

To make it somewhat easier to compile your source code according to our requirements the following batch file is included on the distribution disk under the name LL.BAT:

```
lc1 %1 -ml
lc2 %1
link c1+rclmain+rcl2+%1, %1, %1/m, lcml+lcl
```

Using LL.BAT the command string:

```
ll examprog
```

will created the proper EXAMPROG.EXE and EXAMPROG.MAP files assuming that Lattice's LC1.EXE, LC2.EXE, LCML.LIB (this is their optional math library) and LCL.LIB and our RCLMAIN.OBJ and RCL2.OBJ are all on the default drive/directory.

The ".EXE" file generated with this method is a dummy program in the sense that none of the functions contained in the file can be independently executed. However, running the program does provide us with some important information since it tells us how much space must be allocated for the library when it is loaded

into RUN/C. See below for more information on the use of information contained in an .EXE file generated using this system.

Creating the .RCL file:
Assuming that compilation and linkage were completed without any errors, we now must produce a text file containing the information about the functions which are to be added to RUN/C and the memory requirements of the loaded library. The file will be called EXAMPROG.RCL and will contain:

1) An initial single-line comment identifying the file on the first line.

2) A program size indicator on the second line.

3) A memory size indicator on the third line.

4) The names of all function(s) in the file, the data type(s) they return and a one-line description of each function.

Here is the .RCL file for EXAMPROG:

```
This first line is a comment which is printed in response to LIB-DIR
23=program
1=memory needed
* copy(*,*)    /* like strcpy() but parameters reversed */
int countc(*) /* arg points to filename, returns char count or -1 */
```

The data in the .RCL file will be read by RUN/C when the library is loaded. It is quite important that the information contained in the .RCL file be correct since it is RUN/C's guide to the actual machine code.

Anatomy of an .RCL file
The first three lines of an .RCL file are special.

RCL line 1: should contain a comment which characterizes the nature of the function library. This comment will be printed out when the user asks what libraries are available on the disk with the command LIBDIR, described above.

RCL line 2: indicates how much space the library code needs, including global and static data spaces. It is not necessary to estimate this amount since it is provided to you when you execute EXAMPROG as in:

```
A>EXAMPROG

This is a RUN/C library module.
It is not meant to be executed by the user.
23=program size
```

Just copy the last line into the RCL file exactly.

RCL line 3: indicates how much heap space is required by the library. Heap space is the combined memory requirements of the libraries use of memory allocation functions such as: **malloc(), getmem(), getml(), allmem()** and by file I/O. RUN/C often needs information about required heap space because when it loads the library, it must pre-allocate temporary heap space to it. The library will be loaded only if there is sufficient heap space.

Providing for memory needs of loaded functions
RUN/C loads libraries just above its own code, data and heap and each library owns its own heap. When multiple libraries are loaded, they are loaded one on top of the other. For this reason the memory requirement number is critical when multiple libraries are being loaded. If only one library is loaded, RUN/C automatically allocates all memory above that reserved for the interpreted program (about 40k) to the library.

But if you anticipate ever using your library with another loadable library you will want to accurately estimate memory space needs. How do you estimate the heap requirements for a program? There are two issues: 1) do you use the heap memory management functions **malloc, getmem, getml** or **allmem** directly; 2) and do you use the file I/O function **fopen.** Let's take each issue in turn.

Allocating memory for memory-hungry functions
If you use the memory management functions mentioned above, you should estimate the amount of memory you are likely to allocate and round up to the next higher kilobyte. For example, a two-byte 500-element array would require at least 1k of memory in addition to any other memory requirements.

There is an alternative for programmers who desire all available memory space. For instance, a sort function might want all available memory space for a sort buffer, and furthermore requires a minimum of 15K. In such a case, you can tell RUN/C what the minimum is, and then to allocate all remaining memory above it to the library with the line:

```
15+=memory space
```

The "+" after the 15 effectively tells RUN/C to load the library if there is at least 38K (23K for the EXAMPROG library and at least 15K for heap) in high memory and to allocate any remaining high memory to this library, thus preventing any addition libraries from being loaded.

On a 640k system, memory management would look something like:

337-640	Additional heap available to EXAMPROG functions
321-336	EXAMPROG heap
281-320	EXAMPROG Loaded Library
241-281	Space for the interpreted program to run
40-240k	RUN/C
0- 40k	System memory

Allocating memory for buffered file functions
If you use a buffered file I/O function such as **fopen**, you must estimate how many files will be open simultaneously. Count on 512 bytes (1/2 K bytes) for each open file and round upwards. Remember, this is how many files are open at one time. If you open many files but only one at a time, then all you need is 1/2 K of memory. In that case line 3 of the .RCL file can read:

```
1=memory space
```

if there are no other memory-eating functions in use.

To summarize: Memory space requirements can be expressed by the following C-like equation:

```
if(memory_requirements > 1)
  memory space = files_open/2 + (malloc & calloc needs)
else
  memory space = 1
```

In all cases, a **memory space** setting must appear in the .RCL file.

Note: The number for memory space must be a decimal integer; no floating point or hexadecimal (or octal!) numbers please. If you have not allocated enough heap space and you use buffered file I/O, then one of two things will happen when you try to use a loaded function: 1) if it is the last library loaded and there is adequate memory on top of it, RUN/C will use all available space for your library function and you won't have any problems; or 2)

if not, then various aspects of the file I/O or memory allocation system will fail, possibly requiring a warm reboot of the system.

Function descriptors in .RCL files

Each subsequent line in the .RCL file defines a function, its return type (including the non-returning **void**), and the types of its parameters. To reiterate: the general form of these lines is:

```
return-type func-name(param-type, param-type, ...) /* comment */
```

As for the formal definition which precedes the comment, please keep in mind that:

* Every function must have a return type. For those functions returning nothing, **void** is the return type.
* RUN/C cannot provide complete parameter-type checking for loaded library functions, so care should be taken to make sure that all declared return types are correct.
* It is possible to define a function taking a variable number of parameters (for instance, **printf**) with the special RUN/C parameter type **var**, described below.
* All pointer types are treated identically (see discussion below on * return type and parameter type).

Here, is a list of valid function return types:

```
void
char
short
int
unsigned
long
float
double
*
```

And here is a list of the valid parameter types:

```
char
short
int
unsigned
long
float
double
*
var
```

Special return-type void:

Use this data type for functions returning nothing, even though most compilers and interpreters (including RUN/C) do not require specific void declarations. (If current thinking on the ANSI C committee holds sway, C compilers adhering to the ANSI standard will probably require a **void** return type declarations for functions returning nothing.)

Special return/parameter type "*":
Use the asterisk symbol for any pointer or array type. No distinction is made between character pointers and integer pointers, for instance. Or between pointers and arrays.

Special parameter type "var":
Use this as the last parameter type for functions taking a variable number of arguments. **var** translates to "zero or more parameters of unknown type". As examples of functions needing **var, printf** and **sprintf** would be defined in an RCL file as:

```
void printf(*,var)  /* it must have a pointer first arg */
int sprintf(*,*,var)  /* two initial char pointers, then var */
```

After the function declaration and the declaration of its parameters followed by one or more spaces or tabs, a comment should be composed describing the purpose of the function and of its parameters. The comment may be up to 50 characters of the comment are picked up as "on-line help", and will be given to the user in response to the RUN/C command **LIBSHOW** (described above).

Extracting Functions From .LIB Files

A common use of loadable libraries will be to extract functions from commercially-available C function libraries or from functions you have already compiled and placed into an object file library. How do we extract these functions and create a loadable library file set containing them?

Suppose there are three functions in a .LIB library (called FUNCS.LIB), which you would like: **tom()**, **dick()** and **harriet()**. Create a C source file containing a function like **dummy()** below:

```
/*      EXTRACT.C      */
dummy()
{
    tom();
    dick();
    harriet();
}
```

Then compile the file in the usual fashion:

```
lc1 extract -ml
lc2 extract
```

And LINK it together with RUN/C provided object modules RCLMAIN.OBJ, RCL2.OBJ and include the object library name at the end of the link command line as in:

```
link cl+rclmain+rcl2+extract, extract, extract/m, funcs +lcl
```

The next step is to create an .RCL file counterpart to the .EXE you have created.

Note: Do not define the function **dummy()** in the .RCL file (discussed below) since its only purpose is to tell the linker to extract the desired functions from FUNCS.LIB.

Accessing Global Variables in Loaded Libraries

When an external library is loaded, all the functions and global variables contained in that library are loaded with it. Preprocessor commands such as **#define** and **#include**'s are also processed for that library (in fact, they were processed when the

library was compiled).

In RUN/C and other standard C systems, the words *global* or *external* (we are using these terms interchangeably) refer to variables that are known throughout the entire executable program code. Normally, all C code throughout the executable program can access and change the values of variables that are not local to another function or block.

However, in the RUN/C environment with loadable libraries, global variables are only known within their *domains.* For instance, a global variable inside the library SAMPLE is only known to functions inside SAMPLE and not to any of the interpreted code.

Fortunately it is possible -- with a little work -- to make a library's global variables known, accessible and changeable by the interpreted program. These variables include simple scalars (such as **ints, chars,** pointers of all types), arrays and **structs** and **unions.**

Describing how this might be done will exercise some of what you know about pointers in C. In a nutshell, the address of a library's global variable is passed back to RUN/C by a special function built for that purpose. Once RUN/C knows where in memory that variable is, it can find its value and change it, if necessary.

A simple example: suppose a library has a global variable of type **long** called **number.** This means that somewhere in the library source there is the following declaration:

```
long number;
```

Suppose you also want to manipulate this variable from within RUN/C. Simply write a function for the library which will return the address of **number.** That way the interpreter can know where it is and then it can deal with it. There are several approaches to writing such a function; here is one:

```
long *where_is_number()
{
 return(&number);
}
```

With the function in your loaded library and duly noted in the .RCL file this new function **where_is_number** will be available to and callable by RUN/C. One way to access **number** is:

```
long *pointer_to_number;
long *where_is_number();
  /* must declare function to return pointer-to-long */
          pointer_to_number = where_is_number();

  /* we print out number's current value */
  /* since we know where it is */

  printf("number is: %ld", *pointer_to_number);

  /* and let's change it to, say, 1200045 */

  *pointer_to_number = 1200045L;
```

So all that was required was a new library function to give us the address of the variable so that we could manipulate it.

Now, if there are many such variables that we wish access to, then we can build on the mechanism above to do fancier things.

Let's say that there are two global variables **long fred** and **double george** in the library, declared normally like so:

```
long fred;
double george;
```

Instead of defining two functions, each of which returns the address of its respective variable, let's instead generalize, and define a function that finds the addresses of the global variables we are interested in, puts those addresses into a structure where we can find them from within RUN/C:

```
struct RC_TRANSFER {
        long *point_to_fred;
  double *point_to_george;
} rc_trans;

/* here is the function: */

struct RC_TRANSFER *get_globals()
{
  /* we find the addresses of variables */
  rc_trans.point_to_fred = &fred;
  rc_trans.point_to_george = &george;

  /* and return address of the structure */
  return(&rc_trans);
}
```

Now, inside RUN/C, we will define a like structure:

```
struct RC2_TRANSFER {
  long *point_to_fred;
  double *point_to_george;
} *pointer_to_addresses;
```

And also declare the library function inside RUN/C as is normal C rule requiring the declaration of all functions returning data types other than **ints:**

```
struct RC2_TRANSFER *get_globals();
```

And then, activate **get_globals()** and use the resulting address to manipulate copies of the original **fred** and **george:**

```
pointer_to_addresses = get_globals();

printf("fred is: %ld", *(pointer_to_addresses->point_to_fred));
```

and even create **fred** with the same values:

```
long fred = *(pointer_to_addresses -> point_to_fred);
printf("%ld", fred);
```

etc.

We can also change the variables **fred** and **george** inside the library since we now know their addresses:

```
*(pointer to addresses->point to fred) = 1200045L;
```

etc.

The library must of course call **get_globals()** before **fred** and **george** can be used.

Making RUN/C globals available to library functions

If you merely want your library file to access the values of global variables in RUN/C there is no problem. Simply pass the address of your variable, **struct** or **union** to your library function. Just as with automatic variables, the library function will be able to access these variables through the addresses that you pass.

For example, if you want to pass the value of the global variables **float mary** and **int maude** to a function named **convert** in one of your libraries you simply call the function with the addresses of **mary** and **maude** as in:

```
convert(&mary, &maude);
```

Within the library your function will look something like:

```
convert(p_mary, p_molly)
float *pmary;
int *pmolly;
{
 *pmary = 13.3;
 *p_molly = 3;
}
```

Using the method described you can also pass and modify values between several library files using the interpreter as an intermediary. However, keeping track of the various names and aliases of the variables being passed can become a challenge even using the convenience of transfer structures.

Finally, while a function in RUN/C can call a loaded library function, a loaded library function cannot at present call or execute a RUN/C function.

Assembly Language Interface

Currently, loadable libraries can only be created by users who have a copy of the Lattice C compiler version 2.0+. This is because the two object modules RCLMAIN.OBJ and RCL2.OBJ depend upon Lattice's library. However, it is still possible to create a library out of assembly language routines, provided that you know what the appropriate calling conventions are, and that you link your object modules with Lattice's library.

For those determined to work in assembly language, (or who have covert allegiances to other high-level languages such as Fortran, Pascal or PL/1) and are interested in creating RUN/C compatible loadable libraries using functions written in such languages, herewith is a description of the calling conventions. This interface is consistent with that of the Lattice C compiler, large data and program model.

It is assumed for the discussion below, that you have knowledge of 8088 assembler terms and the 8088 processor register architecture. (Much of the material below is also in the Lattice C compiler manual section on its assembly language interface) Here is an example of an assembly language routine:

Assembly Language Requirements

1) All functions must be FAR Procedures.
2) Functions names must be declared PUBLIC.
3) Arguments to the function are pushed onto the stack in reverse order, so that the first argument is at the bottom of the stack.
4) The calling function's stack frame is in the BP register, and this is the only register that needs to be preserved by the user.
5) The following chart shows the assembly language required to use C data sizes:

Type	Argument on stack	Return register(s)
char	2 bytes (high byte == 0)	AX (AH == 0)
short,int, unsigned		2 bytes AX
float, long	4 bytes (low byte first) float in IEEE (8087) format	(AX,BX)
double	8 bytes (low byte first)	(AX,BX,CX,DX)
pointer	4 bytes (IP first, then CS)	(AX,BX)

Note: pointers have a different representation than longs: pointers consist of two 16-bit words, of which the first is an offset and the second a segment. Frequently the representation is "normalized" meaning that the segment portion is adjusted so that the offset lies in the range (0-15).

Let us take as an example, a routine that is given a character pointer **p**, and an integer **n**, and returns the character at the location **p+n**:

```
; This was assembled by MicroSoft's Macro Assembler

                        EXAMPLE.ASM

------------------------------------------
Title           Example program for loadable libraries

Page 66,80

;***************************************
;
;    Equivalent C function:
;
;    char retchar(where, n)
;    char *where;
;    int n;
;
;    retchar returns *(where + n)
;
;***************************************

pgroup    group _prog
_prog     segment

    assume cs:_prog

    public retchar

retchar         proc  far

    push  bp    ; save caller's stack frame BP

;   stack now looks like:
;   n
;   where (segment)
;   where (offset)
;   return address (segment = CS)
;   return address (offset = IP)
;   BP

    mov   bp,sp       ; set up own stack frame
    mov   ax,[bp+8]   ; get where segment
    mov   es,ax       ; put segment into ES
    mov   si,[bp+6]   ; put where offset into SI
    add   si,[bp+10]  ; add n into offset
    mov   al,es:[si]  ; get char there
    mov   ah,0        ; zero out top byte

    pop   bp          ; restore caller's stack frame
```

```
    ret

retchar   endp
_prog     ends

    end
```

Now it is possible to create a loadable library out of the above file: EXAMPLE.ASM, by assembling, linking and creating the RCL file...

```
masm example;
link cl+rclmain+rcl2+example,example,example/m,lcl
```

And the EXAMPLE.RCL file might look like this:

```
This is an example assembly function
18=program
1=memory
char retchar(*,int)      return nth char in char array given
```

CHECKRCL.EXE -- the Eraser's Edge

RUN/C provides a utility program to check the consistency of library systems before they are loaded. As a time saving and safety measure it is strongly recommended that you use CHECKRCL to verify the correctness of any library you have just created or changed prior to attempting to load the library into RUN/C.

To use CHECKRCL.EXE execute the program with the library name. For instance, to check EXAMPROG, invoke CHECKRCL as follows:

```
A>CHECKRCL EXAMPROG
```

CHECKRCL will look for errors in the .EXE, .MAP and .RCL files that comprise the EXAMPROG library. It will however, only report the first, or first few errors it encounters, so after fixing those, be sure to run CHECKRCL again (and again) until no errors are found.

The error messages generated by CHECKRCL and some possible remedies are listed here in alphabetical order.

Bad param-type in: *line*
In the .RCL file, all function parameters that are declared must come from the set of valid parameter types list in the manual. *Remedy:* Check syntax, also check spelling.

Bad declaration (no return type?): *line*
Syntax error in the declaration of a function in the .RCL file. *Possible cause:* no return type declared for that function (use **void** if function returns nothing).

Bad return-type in: *line*
In the .RCL file, all function return types must come from the valid return-type list found in the manual. *Remedy:* Check syntax, also check spelling.

Filename.exe and *filename.map* were not created together
This message will occur if CHECKRCL detects that the file creation times for the .EXE and .MAP files are not the same (or "close enough"). Since the .EXE and .MAP files are created at essentially the same time by the linker, their file creation times should be the same. *Remedy:* re-link program.

Incorrect parentheses in: *line*
The declared function in the .RCL file has a syntax error related to incorrect number of parentheses. *Remedy:* check syntax.

Line is longer than 80 chars: *line*
RCL file lines should be limited to 80 characters. *Remedy:* check offending line.

Map file does not exist: *ma_filename*
The library .MAP file was not found. *Possible remedy:* Check to make sure the .MAP file is on the same drive/directory as the .RCL and .EXE files of the same primary name.

No closing ')' in: *line*
The declared function in the .RCL file has a syntax error like no closing parenthesis. *Remedy:* Check syntax.

No executable file: *filename*
The executable (.EXE) file of the library was not found. *Possible remedy:* Check to make sure the .EXE file is on the same drive/directory as the .RCL and .MAP files of the same primary name.

No function name(?) in: *line*
CHECKRCL's syntax checker has thrown up its hands on this function declaration in the .RCL file. *Remedy:* Check and repair syntax.

Not a RUN/C executable library file: *filename*
CHECKRCL does not recognize the .EXE file as of the loadable library type. *Possible remedy:* Make sure the object files are linked together with RCLMAIN.OBJ and RCL2.OBJ.

Not enough temporary memory to run CHECKRCL.
Bad news. If you don't have enough memory available to run CHECKRCL then there isn't any chance that you can run RUN/C or any of its libraries. Try eliminating memory resident programs (like Sidekick) or obtaining more memory.

Problem with: *filename*, DOS error code: *error_number*
This message should never occur with MS-DOS 2.0+. Please contact the authors if this message appears.

Program size declared in .RCL file, line 2, does not match program size returned by executing the *library-file.EXE*.
Please execute the library .EXE program and insure that the size printed out matches the one declared in the second line of the .RCL file.

RCL and MAP file mismatch. Possible causes are:
CHECKRCL has found an inconsistency between the functions declared in the .RCL file, and the ones found by the linker and listed in the .MAP file. Further possible explanations are listed

after this error message.

RCL file does not exist: *filename*
The .RCL file of the library cannot be found. *Possible remedy:* Check to make sure the .RCL file is on the same drive/directory as the .MAP and .EXE files of the same primary name.

RCL file must contain 3 special lines, plus one line per declared function
Check and correct format of .RCL file.

Second line should be: nn=program size
Check and correct format of the second line in the .RCL file.

Third line should be:nn=memory needed or nn+=memory needed
Check and correct format of the third line of the .RCL file.

Usage: CHECKRCL library_name
CHECKRCL was called with no arguments or more than one. *Remedy:* execute CHECKRCL with just the library name (one argument).

Section 6. DEBUGGING RUN/C PROGRAMS

Summary

RUN/C Professional provides a wide range of user-selected programming debugging facilities including:

* multiple break points
* program execution tracing (**TRON**)
* variable tracing (**TRACE**)
* program profiling (**PRON**)
* single step tracing (**SET STEP**)
* determination of the value of a variable at any point during program execution
* ability to change the value of any variable and continue with execution of a program
* Interactive mode, which gives access to most RUN/C commands from within an interrupted program
* Immediate mode, which allows execution of C statements or blocks entered on the command line
* execution of most RUN/C commands under interpreted program control (**runc()** function).

The terms in parenthesis -- **SET STEP, TRON, TRACE** and **PRON** -- are debugging commands, described in Section 4 of this manual. This section, which deals with features unique to RUN/C Professional, will describe how these commands interact with each other in relation to:

* the **runc()** function
* the **BREAK** command
* the **VAL** command
* the mini-menu

Each of these features is quite powerful. However the fact that they may be quickly combined in nearly any combination gives them unique power.

The runc() function

In order to do dynamic debugging a mechanism must exist to stop program execution in mid-course and to be able examine and change the contents of various variables and activate or deactivate various interpreter commands.

To accomplish these things and more we have provided a special function called **runc()** which can be used to call various commands that normally are available only within the RUN/C command line. In addition, **runc()** accepts several special commands which are only meaningful when the program is executing.

The form for the **runc()** function is:

```
runc("command");
```

The command can be any of the RUN/C interpreter commands that is not specifically prohibited to the **runc()** function (see below), including the '@' immediate mode command.

To use a simple example, to debug the ubiquitous FTOC.C program you might insert multiple breakpoint statements following by a listing of the program with the profiler statements inside the program (**runc()** function statements are highlighted for the readers' convenience):

```
/*
 print Fahrenheit-Celsius table
       for f = 0, 20, ..., 300
       (This program is from p. 8 of the Kernighan and Ritchie text)
*/
main()
{
runc("PRON");
    int lower, upper, step;
    float fahr, celsius;

    lower = 0;          /* lower limit of temperature table */
    upper = 300;        /* upper limit */
    step = 20;          /* step size */

    fahr = lower;
    runc("break");
    while (fahr <= upper) {
        celsius = (5.0/9.0) * (fahr-32.0);
        printf("%4.0f %6.1f\n", fahr, celsius);
        fahr = fahr + step;
       runc("break");
       runc("list");
       runc("PROFF");
       runc("SAVE");
    }
}
```

All RUN/C commands are available to **runc()** with the following exceptions:

*	RUN	*	NEW
*	LOAD	*	SYSTEM
*	EDIT or E	*	EDITOR
*	HELP		

Thus you can use the **runc()** function to have your program **LIST** or **LLIST** itself, to do a **SHELL** operation, to load, unload or show a loaded library or to get a directory of available loadable libraries. You can also get a directory listing (**DIR**), list available commands (**CLIST**) or functions (**FLIST**), report available memory (**FRE**), show the **HELP** menu, **KILL** or **UNLINK** a file on disk.

You can also turn the profiler on (**PRON**) or off (**PROFF**), turn on single-step tracing (**STEP**), toggle **TRACE**, activate **TRON** or **TROFF**, set or reset **TABs**, **SAVE** a file and activate other **SET** commands such as the "TRUST-me" mode on or off. The **EXEC** command which allows you to load and execute a program in one operation can also be called from **runc()**.

In addition to multiple breakpointing, most other debugging commands can be turned on or off at a particular point in the program. For example at line 10 the statement:

```
runc("TRACE"); runc("TRON");
```

would activate these two debugging functions. If line 20 contains the statements:

```
runc("TRACE"); runc("TROFF");
```

These facilities will be turned off at that line (recall that **TRACE** is a toggle.) Now every time your program is run the **TRACE** and **TRON** facilities will be activated, but only between the two points of interest.

By the same token, a statement such as:

```
runc("LLIST 10-20");
```

will make a printer listing of your current code for the specified program lines each time the program is run (assuming your printer is on and properly connected).

Don't forget that **runc()** doesn't require a string constant for an argument. Consider this program:

```
main()
{
char ac[80];
cls();
puts("mini-interpreter based on runc() function");
puts("type SYSTEM to return to the REAL interpreter");
do{
 puts("\nOk\n");
 gets(ac);
 runc(ac);
   } while (strcmp(ac, "SYSTEM") != 0) ;
}
```

The **runc()** function also makes it easier to write a self-reproducing program (whatever that's good for) in very short form. Here is such a program:

```
main()
{runc("nlist");
}
```

The program output is above.

Portability issues

Although the **runc()** function follows all normal function rules (for example, it must either be inside **main()** or inside a function called by **main()**) **runc()** is obviously not a standard C library function and cannot be compiled. However you can **#define** the function away with the statement:

```
#define runc(n)
```

in your stdio.h file or the equivalent. Or you can define a dummy function:

```
runc(cp)
char *cp;
{ }
```

BREAK and VAL Commands

Two special interpreter commands can be used only in conjunction with **runc()**, these are: **BREAK** and **VAL.** The **BREAK** command is the equivalent of pressing a <Ctrl-c>. The advantages of **BREAK** are:

1) it can be placed exactly where you want it in your program.
2) it can be used as many times as you wish (multiple break points).
3) it can be activated under program-controlled conditions.
4) It can be used in conjunction with other commands.

For example:

```
if(a < 0 || a > 100)
      runc("break");
```

Will cause a **BREAK** in the program when the statement is executed if the variable 'a' ever equals less than 0 or greater than 100.

Any number of calls to the **runc()** function can be made as in:

```
if(a < 0 || i > 100){
      runc("llist 10-20");
      runc("break");
      }
```

which will list program lines 10-20 at the printer, print the current values of three variables at the printer and execute a **BREAK** command.

When the **BREAK** is executed you will immediately be placed in RUN/C's mini-menu which looks like:

```
Break in line 3
Continue, Dump, Ldump, Interactive, Help or End (C,D,L,I,H,E)?
```

The mini-menu provides an elegantly small but powerful set of debugging commands. The six choices in the mini-menu, accessible by choosing the appropriate letter, are described by the H(Help) option:

```
Continue --        Resume program execution from the point of interruption
Dump --    Display names, types and values of variables currently existent
Ldump --   Same as 'dump,' but to printer; BE SURE PRINTER IS ON LINE
Interactive -- Use subset of interpreter commands with program data available
Help --    Display these messages
End --     Cease program execution and return to interpreter command level
```

These options are described in more detail Section 4 of the manual. Perhaps the most interesting of the group is the Interactive command. If it is chosen a special "double prompt" will appear:

```
(Use HELP or ? for help with interactive mode.)
>>
```

If we select the help command we will see:

```
You can use most, but not all, interpreter commands to examine parts of
your program in its interrupted state. Commands of most likely interest:
        LIST [line range] -- displays your program's source code
        VAL variable_name -- displays type and value of specified variable
        @ function_name(arguments); -- execute one of your program's functions
        @ variable_name = expression; -- change a variable's value
        SET STEP ON/OFF -- turn single-step mode on or off
        TRON/TROFF -- turn source-line trace on or off

You cannot edit or erase your program in this mode. The following commands are NOT
available: EDIT, E, EDITOR, LOAD, NEW, RUN, SYSTEM.

Use SYSTEM or press the <enter> key (alone) to leave this mode.

>>
```

The above help message is quite comprehensive and there is little to add other than to provide a simple example.

If a **BREAK** has been executed because an unexpected condition has occurred -- for example a variable has suddenly and contrary to the laws of physics developed a negative value -- you can determine its exact value with:

```
>>VAL upper
```

which will print out:

```
Automatic variable of current function: upper: int 300
```

At this point you use the interactive command mode to change the value of the variable in question, as in:

```
>>@ upper = 1000;
>>SYSTEM
```

This sets **upper** back to a positive value and returns you to the interrupted-execution mini-menu.

Using **VAL** in conjunction with **BREAK** it is much easier to trace through the possible values of program control variables and understand how your program will execute under most if not all operating conditions. This can be a tremendous time saver in trying to trace through the many paths your program can take depending upon various user-generated or system enforced conditions such as running out of memory or failing to open a particular file.

SECTION 7. IMPORTANT TECHNICAL and OTHER NOTES

Summary

There are a few extra things about RUN/C Professional that we want you to know about ... and this is where we put them.

The first thing might sound silly or stupid, but it's important to us. It's about play.

Most of us who know some BASIC know it because a BASIC interpreter let us play with the language. You remember play -- that's something you do not for some hard-edged purpose, but merely for the fun or the wonder of what you're doing. We sincerely hope that RUN/C will bring the spirit of play to the C programming language. One of the main ideas behind RUN/C is to let you try out a lot of different things in C and make lots of illuminating mistakes, safely and quickly. Soon enough, you'll be setting yourself challenging programming tasks, and completing them, just because they interest you. We think that playing is the ideal way to learn a computer language.

Some people will now be thinking, "Ah ha! they're trying to justify their toy implementation of C." No such thing. RUN/C is not a toy. There are lots of playful things you can do with a computer, a lathe or a piano, but they don't make any of those instruments a toy *per se*. RUN/C can be as serious as you want it to be; what's important is that you can play with it -- something that's not so easy to say of even the best of C compilers.

If everything in life had to make you healthier, wealthier, a better person or a more valued employee, life would be a lot duller. Play is important, and we want to make it possible. Recess!

* * *

If we were interpreting BASIC or compiling C, things would have been a lot simpler for us. We've made compromises, sometimes for our own convenience, but more often for yours; we hope we've made them intelligently. Because RUN/C is one of the first true interpreters for the C language (certainly the first to approach a full implementation of the language), it is guaranteed not to be perfect.

We feel we owe it you -- our customers -- to give you some inside information about the compromises we made (and why we made them) and about RUN/C's known imperfections.

Almost everything below has implications for the compatibility of RUN/C programs with C compilers. Since we expect that most users will want to compile their finished programs eventually, the information should be of interest. Topic-by-topic coverage follows.

Line vs. Stream

A true interpreter is to some degree bound to the program line as a unit of information. (We haven't yet figured out how to get all the way around this; if you have, we'd like to know about it.) The C programming language, by contrast, doesn't know or care a thing about program lines; a C program is a stream of characters.

This fundamental difference has led to these RUN/C characteristics:

Automatic or "virtual" line numbers

If you're coming from BASIC, it may take you a little while to get used to our autoline system of automatic renumbering of program lines; if you're coming from a screen editor and a C compiler, you may find the very idea of line numbers even less appealing. But we needed pegs to hang certain commands and error messages on, and automatic, sequential line numbers were the best pegs available to us. Our line-handling scheme also seemed the way to permit maximum freedom in program formatting -- where it counts most, your formatting style is yours, not ours.

Now that RUN/C includes a full screen editor, line numbers are less important than ever. The **INSERT, DELETE** and **MERGE** commands no longer exists, and while the **EDIT** (a/k/a **E**) command will still accept a line-number argument, you may never need or want to use it that way. Even if your attempt to run a faulty program has resulted in an error message that indicates the problem line, you can command **EDIT ERROR** (or **E ERROR**) and go directly to the trouble spot.

Comment conventions

We used to require that every comment in your program begin and end on the same line -- and if it didn't, we would end it there for you. We provided a loophole by letting you "block comment" by framing an undesired run of lines with **/*** all by itself on a line and ***/** all by itself on another.

But with age we've become more lenient. If you fail to end a comment on the same line you began it on, we will give you a warning, either as you load from disk or in the editor. We will not tinker with your code. RUN/C can now work around comments that begin on one line and end on another. Comments that you forget to complete will lead, on program execution, to errors like "Unmatched braces" or "Unexpected end of program"; in some rare cases they may lead to error-free execution -- but not of the program you had in mind. Let us advise you, in a friendly way, to heed the warnings issued and try to begin and end comments on the same line.

Our block-comment convention remains available, but should be considered tentative and hazardous, as it will easily give bad indigestion to any number of sound C compilers. In a future release, as we strive to follow the emerging ANSI C standard, we will probably stop allowing nested comments and the block convention will go the way of the dodo.

String constants
Most C compilers permit multiple-line string constants. We don't. If you don't end a string constant (anything inside double quotation marks) on the same logical line you started it on, RUN/C will give you a warning that you really should pay attention to. Attempts to execute programs containing unterminated string constants will almost certainly not give the results you wanted. If your program is of the dirty kind that sets a pointer to a string constant and then modifies what that pointer (and/or its value after incrementation or other manipulation) points to, the result is likely to be that mentioned by Shakespeare: "Chaos is come again."

Ditto character constants (framed with apostrophes).

Special characters within strings
or
What you see is (usually) what you get
We started with the goal that what you loaded or typed in as part of your program should always come back the same way when you **LIST** (or **LLIST, NLIST** or **NLLIST**) or **SAVE** your program. We believe we have achieved this goal with our new editing system, which keeps an ASCII copy of your program exactly as you entered or loaded it.

Thus the following discussion now applies only to the way program lines are displayed during execution when the **TRON** command is active:

When you specify a string of characters -- or, more formally, a string constant -- within a string-oriented function such as **printf**

and its family or **puts** and its family, every character within that string is converted to our internal (tokenized) version of your program. Then, when you view your running program with TRON, we first reconvert the internal version back to readable ASCII text. This works exactly as you would expect (well, not quite exactly -- see below) for ASCII text in the range 32-127. For many other characters such as the control codes, you must supply octal values in keeping with C, as it was originally defined by K&R.

The following chart shows character values which require special input (values are decimal except where otherwise specified);

Input	Appears as
\0	\0
\000	\0
\t (tab)	\t
\011	\t
\n (newline)	\n
\012	\n
\f (formfeed)	\f
\014	\f
\r (return)	\r
\015	\r
any other character values less than 32	\ooo*
\177**	\177
128-255	\ooo***

In addition, several ASCII characters must be input using special conventions when they are used within quoted strings:

Character	Input as	Lists out as
\ (92)	\\	\\
" (34)	\042	\042
' (39)	\047	\047

Confused? So were we. Part of the problem must be laid at the (all too) enduring octal tradition that was the low-level *lingua franca* of the minicomputer world C was born into. We do apologize for the rigidity of our own TRON-out conventions and hope that living with them is not excruciatingly painful to you.

In looking over the chart you probably noticed that we don't allow

---

* in octal format.

** 177 octal equals 127 decimal.

*** octal value less 177 (high bit is 'stripped').

the use of characters with decimal values greater than 127 within a quoted string. Our line-input and editing system won't even let you use the DOS input method of holding down the <Alt> key and typing three digits on the numeric keypad to input a character with a decimal value above 127. If you do want to use the PC's (or equivalent's) character graphics in the range 128-255 decimal, you must use **char** or **int** variables or constants given those specific numeric values. For example,

```
int c;

        for (c = 128; c <= 255; c++)
              printf(" %c", c);
```

will display the whole range of high-bit character values.

HERE'S THE GOOD PART -- please note this even if you didn't bother with all those octal thinggummies above: When you use **TRON**, and you have **#defines** in your program, the program lines displayed during execution will show your macros (preprocessor symbols, **#defines**, whatever you want to call them) in their expanded form -- the way they work out after preprocessing. This is all the more important when you use **#defines** that take arguments, and may be one of the most important debugging facilities we've provided. (We can't think of a popular compiler that gives you this kind of window into your code.)

Labels

You may never use a **goto** statement in your C programming -- we didn't when we wrote RUN/C. But when and if you ever do, you should be aware that we imposed restrictions on labels -- 'goto' destinations -- partly because of their rarity. A label must be the first thing on its line that isn't whitespace, or it will probably be misinterpreted. (For other restrictions, see **goto** in Section 4.)

At the same time, we're not anti-**goto** fanatics either, and recognize that **gotos** may conceivably get heavy use in programs that are adapted (especially machine-adapted) from other languages or produced by preprocessors that generate C source code. If your work requires more than 10 (or our current maximum -- use **HELP** to check it) **goto** labels in a program, please contact us -- well try and get a special version that meets your needs ready for you.

A Safe Nest

"We're All Adults Here"

Part of the mystique of C has been that it will let you attempt almost anything, with the result that correct programs can do some pretty amazing things -- while bad programs can cause some pretty spectacular system crashes.

Part of what an interpreter is about is allowing a soft landing from almost any dicey situation -- making a safe environment that lets you walk away from your crashes. We've worked hard to make RUN/C a safe playground for experimentation ... but C is still a grownups' language (the maturity required is not, however, necessarily a function of age) and some warnings are in order:

Array bounds checking

Although RUN/C attempts to do array bounds checking, there are ways around our safeguards. Keep in mind that if you are twiddling array elements and go beyond that top limit, you may just go on twiddling array elements that aren't there -- with potentially disastrous (to your program) consequences.

Pointer safety

As of Version 2.0, we are pleased to provide one more layer of error trapping for pointer use. ALL newly-declared pointers, automatics as well as "globals," are set to NULL if they are not otherwise initialized, as long as you have not turned our SET TRUST flag on (Like most of the other keywords discussed in this chapter, **SET TRUST** is described in Section 4). Additionally, attempts to use built-in input functions, such as **gets()**, with NULL-valued pointers will yield errors. (Previously, you could use **gets()**, e.g., to type into an arbitrary location in memory, most often blowing away system vectors and thus crashing your computer.)

We have found these traps invaluable in finding use of uninitialized pointers. We urge you to remember that a) SET TRUST mode turns the trap off, and that b) no compiler that we know of provides this facility (though a good Lint may).

String constants in your program

If you set a character pointer to a string constant in your program, you are really and truly setting that pointer to point to the string constant where it's stored in RUN/C memory. If you use the pointer merely to print out the constant, or to copy it elsewhere, all's well. But if you change what the pointer points to, you are changing your program itself. If your changes go beyond the end(s) of the original string constant, what they'll end up doing is anyone's guess.

Frankly, we consider this kind of programming dirty code. Yet many a guilty program, when compiled, will run successfully, again and again. Why is this? It's because a compiled program, as an .EXE or .COM file, includes a complete copy of all its constant data, and each time you invoke such a program from the DOS command line, you reload a complete, fresh copy of program and data. Nasty twiddling of the data segment does nothing to the constant data on disk, while such twiddling in the RUN/C environment can change data in the tokenized version of your program (or, worse, make changes beyond the bounds of the original data), and when you attempt to re-execute under RUN/C you'll be starting out with the mutated string.

Unfortunately, when you list your program will now longer show you that a string constant has been modified, because what you see is now the original ASCII version of the program you ran; not a recreation from the tokenized version. Check for this phenomenon with **TRON**, which recreates readable program lines from tokenized code and can therefore show you if string constants in the tokenized version have been affected. (When that's the case, you can make a temporary restoration by using **EDIT** to make some trivial change to the line (not necessarily the particular string constant) involved and thus force retokenization of that line; but we would urge you to repair the deeper flaw in your program.)

Why are we belaboring this issue this way? Because it has proved to be the commonest cause for alarm among RUN/C users and because it is (for now) impossible for us to trap for you.

There is no assured limit to the severity of the problems you may create by these means and Age of Reason Co. can not assume any responsibility for data loss or other resulting problems. We are evaluating methods for eliminating remaining hazards, and may incorporate them in future releases of RUN/C, if they do not prove to be excessively costly in terms of memory use or program execution speed. In the meanwhile, you've been properly warned. We're all adults here, right?

Scope quirks

Implementation difficulties (or the fact that we're not as smart as we'd like to be -- or thought we were) have led to some nonstandard restrictions on the scope of certain program entities, as well as some nonstandard liberties in scoping. These involve preprocessor symbols created with **#defines**, **goto** statements and labels, and **#include**'d files. Please see the corresponding headings in Section 4 for details.

There are quirks, too, in the way RUN/C handles the scope of

aggregate (**struct** and **union**) definitions. Essentially, they work the way variables do. An aggregate definition that appears within a function may, if it contains a tag, be reused anywhere within that function but disappears once the function ends. A collection definition that is external to all functions may, if it has a tag, be used anywhere in your program -- except where it is eclipsed within a function by a local aggregate definition that uses the same tag.

That much is orthodox. The C language, however, permits forward reference among aggregates (as long as the reference is one that can be made without knowing the size of a yet-undefined aggregate). RUN/C does not. If one aggregate is to include another, or a pointer to an example of that other, or an array of those others, the other aggregate has to be defined first. (And it has to be defined with a tag or you can't make any but immediate use of it anyway -- but you knew that.) An example of what we won't buy that your compiler might is:

```
struct s1{ char c;
            int i;
            . . .
           struct s2*s2p;
          };

struct s2{ double d;
           float f;
           . . .
           struct s1*s1p;
          };
```

So don't do that, ok?

Please note the only form of self-reference we permit within aggregates is via pointers. This is in keeping with standard implementions of the language.

About typedef

We're delighted to be providing a reasonably complete implementation of the **typedef** keyword now. We owe you information on that implementation's limits, which are:

(1) **typedef**'s don't nest. If you say:

```
typedef int WHOLENUMBER;
```

and follow that with:

```
typedef *WHOLENUMBER WNP;
```

we just can't cope.

(2) **typedef**'s that are arrays can't be used in forming new objects that are arrays of those typedefs. For example:

```
typedef int[10] SOMEINTS;
```

is fine, and so is:

```
SOMEINTS a10ofint;
```

but

```
SOMEINTS a10by10ofint[10];
```

will yield an (admittedly gratuitous) declaration error.

We will remove these obstacles as soon as possible and hope, in the meantime, that they don't inconvenience you greatly.

Arbitrary limitations
Any release of RUN/C will probably involve some limitations on the number of lines a program may have, the number of functions it may contain, and the like. These are best reviewed via the on-line **HELP** (or **?**) command. Some other limitations merit discussion here:

Declarations within blocks
According to the definition of the C language, automatic variables may be declared within any block (compound statement framed by **{** and **}**) and remain local to that block. This convention is now supported by RUN/C, but you should be aware that **static** variables retain function scope, although a **static** may be (temporarily) eclipsed by an automatic of the same name declared in a deeper block, and statics should still be declared near the beginnings of functions.

About the Editor

The RUN/C built-in screen editor replaces a functional but unexciting line editor. Its virtue -- other than attractiveness and charm -- lies in the fact that changes you make to your program are immediately registered in the interpreter. No recompilation or semi-recompilation of the program or its functions are needed. This is the virtue of a true interpreter.

RUN/C actually maintains two copies of your program -- the editor's copy as you typed it and a separate, tokenized version which is visible only when you use the **TRON** command in

conjunction with **LIST**. This method also speeds up program **SAVE**s considerable; **LOAD**s are somewhat slower than **SAVE**s due to the syntax-checking and tokenization process.

The point of all this is that your version of the source code shares a common data space with your program and its creations. All this is subject to potential damage by stray pointers and arrays without end, etc. Every effort has been made to protect the interpreter/editor combination, but we have observed "dirty" code which bashes the editor's version of the program (i.e., you see garbage when you call the editor) apparently leaving RUN/C intact. If this happens please 1) reboot your system, 2) give us a call if you have a repeatable problem in this regard.

About the Manual

Redundancy Redundancy

Some duplication of information is unavoidable in a software manual unless one is to assume that the user reads the manual thoroughly and completely before attempting to use the program -- unlikely. Consequently, one of the decisions we were faced with was whether to consolidate information in a single place and refer frequently to it or to provide full or near-full documentation each time a topic was discussed.

As you probably have already discovered, we've come down on the side of full documentation with the result being repetition of some key information throughout. We hope the convenience of immediacy outweighs any occasional irritation caused by having to plow though familiar-sounding paragraphs. If not, our suggestion box is always open; our address is listed on the copyright page. (Regarding our decision not to repeat our address ... as usual, the exception proves the rule).

Error messages

Our goal is to provide in-depth information on the cause and cure of error messages in the body of the manual, with only a brief discussion in Appendix A where all the error messages are listed. Unfortunately, our goal remains largely unrealized as of this writing and therefore it is still possible to turn up some pretty baffling and/or generalized error messages. When in doubt, call for help.

How Close to a Full C Are We

RUN/C is presently a "nearly full" implementation of the C programming language. In addition to items already mentioned in this section, the following features are not implemented as of this

writing:

* preprocessor directives other than **#include** and **#define.**
* meaningful storage-class specifiers (see below)

Some of these items will be implemented in the near future; others may wait until the ANSI C standards committee has settled on a definition of C; honest-to-god register variables will probably never exist under RUN/C. Meanwhile, we're getting ready for **D** (or will it be P?).

Running programs from books about C
When we started this project there were perhaps five books on C available through normal bookstore channels. That number has increased about six-fold as of this writing -- we have the complete collection -- with probably another 20-40 books, tutorials and other specialized C publications due within a few months as the industry "ramps up" for the emergence of C as the programmers language through the rest of the decade.

We have made a conscientious effort to make sure that programs listed in everything from K&R's The C Programming Language to Alan Feuer's C Puzzle Book run. But in some cases they don't. Some programs you typed in from a book on C may not run as advertised -- or at all. Here are some possible reasons why:

1. The program you are trying to run uses a feature of C which we have not implemented.

2. The program may have errors in it -- either from the author, the publisher or from you, the person who typed the program in.

3. The program may be based on a non-standard C compiler (such as BDS C), or may be taken from a UNIX operating system using an advanced or experimental version of C (they are around).

4. The program may declare functions. RUN/C has its own set of built-in functions (see **FLIST**), making it unnecessary (and not permissible) to re-declare built-in functions -- even when they return values other than **chars** or **ints.**

5. The program may use preprocessor commands which RUN/C does not support.

6. The program may "hide" variables within blocks; RUN/C now supports this feature -- except for statics.

7. An aspect of the program may exceed the limits placed on RUN/C programs. See **HELP** for an on-line listing of current limits.

8. The program depends on an external library or stdio.h file which RUN/C cannot run (standard information contained in stdio.h is built right into RUN/C).

9. The program attempts to create a single data object (such as an array or structure) larger than 64k bytes.

Usually it is possible to code around such limitations, although it can be a nuisance in some cases. And things can only get better -- we expect to have full C soon, and to go beyond any possible quibbles pretty soon after that.

Compiler issues

We do not expect your program to stay with RUN/C after you have it running right. And frankly, even after your programs runs correctly under RUN/C, compiling them is not guaranteed to be an automatic process. But if you remain aware of the following points, you should find moving your program to any C compiler relatively straightforward.

Choice of C compiler

RUN/C programs are currently maximally compatible with the Lattice C (Microsoft C version 1.05 and 2.0) compiler, although there are many other excellent MS-DOS-based compilers (Aztec C by Manx Software is one that we have used with satisfaction). Microsoft/IBM C (3.0 or greater), Computer Innovation's C86, DeSmet C88 and Wizard C also come highly recommended and will run most RUN/C programs "right out of the box."

Note also that some of RUN/C's built-in functions are modeled after functions in Lattice's C-Food Smorgasbord, which additional library will be required for compilation of programs using those functions.

Header file(s)

Many (but not all) programs that are to be compiled require the inclusion of a standard header file, most often (but not necessarily) named "stdio.h." Because of its predefined preprocessor symbols and predeclared built-in functions, RUN/C currently won't let you include a file named "stdio.h"; you can get away with something like "b:stdio.h" -- but you wouldn't do that, would you?

Predeclared functions

When RUN/C was conceived we decided to predeclare all RUN/C built-in functions returning other than the default data type **int** so that those being introduced to C would be able to, for example, allocate memory without having to master the intricacies of library function declarations such as:

```
char *malloc();
```

Now that RUN/C has grown up a bit, we realize that our plus has a minus side when it comes to compiler compatibility. In our opinion, the best way to handle declarations of library functions returning other than **int** is though a header file as is commonly done in many compilers. This makes for both less cluttered and more portable code.

However, you can reference a built-in function in a RUN/C in a manner which will be satisfactory to most if not all standard C compilers by declaring it as **extern** to your program and outside any function body. (This is actually the case for built-in RUN/C functions, but that's another story.)

For example our friend **malloc** can be declared as:

```
extern char *malloc();
```

Inside RUN/C whether or not **malloc** is declared this way **malloc** will return a character pointer; this is not true in a compiler environment. However in a compiled environment the above declaration will be sufficient to insure that the needed object code for **malloc** will be linked in and that **malloc** will be expected to return a character pointer.

Side effects
All RUN/C built-in functions are implemented as functions, so that side effects from them need be of no concern -- until you port to a compiler that implements some as macros. Study your compiler's manual and "stdio.h" file (or the equivalent).

Pointer size
Under RUN/C (because it is a "large model" system), all pointers are the same size as **longs**, and thus require four bytes -- not two bytes, the size of an **int**. Additionally, the predefined symbol NULL is equivalent to a four-byte, not two-byte, zero. Unless you've played games that depend on pointer size, this should pose no problems in porting; but if you are compiling your program under Lattice's large-memory model, you must redefine NULL as 0L.

Range of character values
RUN/C follows a common convention of taking **char** variables to

be **unsigned** and allowing them to take values in the range 0 to 255. You should be aware that some C compilers take the highest bit of a **char** to be a sign flag, and thus allow a **char** to take only values between -128 and 127. Implications for arithmetic on **chars** follow as a natural consequence.

Closing files
RUN/C closes all buffered files at the end of each program RUN. Compilers do not always provide this safety factor. Your program should end with a call to the **exit** function to ensure proper file closing; be aware that your compiler's **exit** function may or may not close unbuffered as well as buffered files. It is good programming practice to close each file explicitly when you are done using it.

EOF (End-of-file marker)
EOF is pre-defined within RUN/C to equal **-1**. This is in accordance with common practice. Many programs you will encounter call for the generation of an **EOF** from the keyboard. The **<Ctrl-z>** key can be used in this regard, since it generates a -1 (rather than the expected ASCII 26) value. It is not possible to generate a decimal **26** from the keyboard. See **INKEY** in Section 4 for details of key value assignments.

'void'
RUN/C supports the neologistic **void** keyword. Many compilers do not.

storage-class specifiers
In the interest of letting you load and run as many existing C programs as possible, RUN/C will now pass without comment the storage-class specifiers **extern, auto** and **register** within these conventions:

extern — illegal within a function; as a "global" variable a declared **extern** will generate a warning message and the variable declared will be considered a RUN/C global just though the keyword **extern** had not been used. See the discussion regarding predeclared functions, above.

auto — legal and ignored within a function; illegal when used in a declaration outside any function.

register — legal and ignored within a function; illegal when used in a declaration outside any function.

Note that RUN/C, since it largely ignores the three above storage-class specifiers, cannot be concerned with fine points about their use. The result of their addition to RUN/C is that

more of your existing C programs can be loaded and run (can be imported), but that it is now easier to create a program that will run well under RUN/C but will choke a self-respecting C compiler (export not OK). Please be aware of this potential conflict for now. When time, life and breath permit, we will make storage-class specifiers work more strictly as they do under compilation.

As a general caution (and good practice in any programming language), make sure you test your program thoroughly with both good and bad input -- preferably under RUN/C where bugs should be easier to survive and to fix, before you attempt compilation. The opposite also makes sense: if something you are trying to do with RUN/C should work but doesn't seem to, try compiling it with a reputable C compiler.

Additional Notes for RUN/C Professional

There's plenty of good news, but most of it is already collected in Sections 4, 5 and 6 under the headings of the several **LIB** commands and the **runc()** built-in function.

We hope you'll thoroughly explore RUN/C Professional's facilities for:

* Advanced source-code debugging with a flexible window into your running program.

* High-speed execution when needed via Loadable Libraries of your compiled functions (but we trust you'll always test their exception handling under interpretation first).

* Using OPC (Other People's Code) at near-maximum execution speed when it's made available to you either in source form or as an .LIB or .OBJ file created using Lattice's Large Model compiler.

To be continued . . .

APPENDIX A. RUN/C SCREEN MESSAGES

Summary

There are three general groups of messages which RUN/C generates on the screen: 1) numbered error messages; 2) unnumbered error messages; 3) general messages and statements.

An asterisk by the error message indicates either an operating system or internal RUN/C error. If an error of this sort is observed and can be predictably reproduced, the authors of RUN/C would appreciate hearing from you. Please contact us at the address listed on the copyright page.

Whenever a particular function, command or keyword is mentioned, the first place to look for a hopefully illuminating explanation of the error message is the portion of Section 4 devoted to the respective keyword. Failing this, check the index.

Numbered error messages

0* No mistake at all. What am I doing here?
Should not appear under normal conditions.

1* exit code -- this message shouldn't print
Should not appear under normal conditions.

2 Syntax error
A "catch-all" term for several types of illegal construction in a program line. Examine your code carefully for possible errors or, less likely, interpreter bugs.

3 'goto' without destination
No identifier following **goto.**

4* Definite error -- not yet assigned its own message
Should not appear under normal conditions.

---

* Error messages followed by an asterisk(s) indicate internal errors or failure of the interpreter's safety net. Very likely such a message also means problems in your program.

5 Required lvalue not found.
Attempt was made to use an expression (possibly a NULL expression) that could not legally appear on the left side of an assignment expression.

6* Operator not recognized (not yet implemented).
Should not appear under normal conditions. All K&R C operators are implemented.

7 Macros too deeply nested
Too long a chain of interdependent preprocessor symbols was supplied (more than 15).

8 Break or continue not within a loop
break or **continue** statement was found which is not within a **for, do, while** or **switch** body.

11 Wrong number of arguments for built-in function
Attempt was made to pass too many or too few (i.e., the wrong number) of arguments to a function. See Section 4 for the requirements of the function in question.

12 Expected argument type not provided
Argument to a built-in function was not of the type or range of types required. See Section 4 for the requirements of the function in question.

19 Stack underflow in expression evaluation
Confused expression with too few operands for its operators.

20 Stack overflow in expression evaluation
Expression too complicated. *Possible solution:* Simplify, separate expression, or use more parentheses.

21 Variable not found
Attempt to use an undeclared variable or one not available to the current function. May also occur when a call to an undeclared or undefined function is encountered.

22 Declaration problem
Either an attempt has been made to declare an object that is an array of elements each of which is a **typedef** that is itself an array or a syntax error(s) in declaration or use of unsupported construction(s) such as **short int** or **long float** has been detected.

23 Too many automatic variables
More automatic variables active at one time than current implementation of RUN/C permits. See **HELP** or **?** for the current maximum number of automatic variables.

24* Too many arrays
Should not appear under normal conditions.

25 Invalid dimension in array declaration
Either 1) dimension provided did not evaluate to a positive integer, or 2) an array dimension was not a constant.

26 Attempt to declare array with too many dimensions
Attempt was made to declare an array of greater than the seven dimension maximum.

27 Insufficient memory for array or allocation
An attempt was made to declare an array or make a memory allocation larger than the amount of memory available.

28 'break' not within appropriate structure
A **break** statement was found which is not within a **for, do, while** or **switch** body.

29 'continue' not within appropriate structure
A **continue** statement was found which is not within a **for, do, while** or **switch** body.

30 Semicolon expected
Improper end of statement.

31 'do' without 'while'
Improper construction of a **do** statement.

32 Test without parentheses in loop or switch
Improper construction.

33 Malformed 'for' statement
Improper construction of a **for** statement.

34 Constant expression required with 'case'
Attempt to use variable with **case.**

35 Array or allocation release failure
May occur if program has operated legally, but has reassigned a multi-dimensioned array's subarrays. Signals possible memory lost or introduction of

garbage into the user's program. *Possible solution:* Save program (if still intact), execute a **NEW**, reload program.

36 Illegal use of subscript
Subscript applied to identifier that was not declared as an array or a pointer.

37 Illegal scalar use of array
Attempt to use an array name (without subscript) as object other than address.

38* Value out of range for type required
Should not appear under normal conditions.

39 Invalid function invocation
Either 1) **malloc** or **calloc** was passed arguments which evaluated to **0** or, 2) a reference to user-defined function was not followed by a left parenthesis.

40 Bad parameter list or parameter use
Types of arguments supplied to a user-defined function do not match parameter types.

41 Attempt to divide by zero
Divisor was or evaluated to zero.

42 Object is not a pointer
Attempt to de-reference an object that was not or cannot be treated as a pointer.

43 Attempt to free memory not obtained through allocator
Wrong argument supplied to the built-in **free** function.

44 Too many allocations
Attempt to make more memory allocations than the maximum allowable. This is not a count of the total number of memory allocations made, but the number of allocations made less the number of **frees**. Thus the number of allocations at any given time within a running program equals the total number of **mallocs** and **callocs** less the number of **frees** executed. See **?** or **HELP** for the current allocation limit.

45* Too many files
Should not appear under normal conditions. Note:

check for NULL returned by **fopen.**

46 Bad file mode
Mode string argument to **fopen** not recognized.

47 Null-valued file pointer
Attempt to perform a buffered file operation on a file which was not successfully opened with **fopen.**

48 Illegal operand type
Attempt was made to supply an object of a type the operator will not accept.

49 Illegal operands combination
Attempt was made to supply a combination of object types that the operator will not accept.

50 Attempt to add two pointers
Attempt was made to add two pointers together.

51 Negative subscript
Attempt to supply a negative subscript to an array. Note: negative subscripts are legal for pointers and for arrays given as formal parameters.

52 Attempt to relocate array
Attempt to assign an array to a pointer or pointer type object. Note: this is legal for arrays given as formal parameters.

53 Math function argument out of range
Argument given was too large, too small or of the wrong sign. See appropriate function, Section 4.

54 'else' without 'if'
Improper construction.

55 Keyword irrelevant to this environment
Attempt to use a keyword such as **register** or **extern** which is inappropriate and not support in this implementation.

56 '?' without ':'
Improper construction.

57 Label not found
Destination label for **goto** statement could not be found within the current function.

58* Branch (soft error)

Should not appear under normal conditions.

59 Duplicate identifier
Same identifier was used twice or more at the external (global) level, or twice or more within one function.

60 Function not declared or declared incorrectly
Attempt made to call a function returning a value of type other than **char** or **int** without declaration of that function either at the external level or within the calling function.

61 Invalid character constant
Either 1) apostrophes have been used with nothing within them (an illegal construction) or, 2) more than one character or escape sequence was placed within apostrophes.

62 Attempt to de-reference NULL-valued pointer
Attempt to use the object pointed to by a pointer whose current value is **0L**.

63 Subscript out of range
Subscript supplied to an array was greater than the array's dimensioned size. Note: this checking does not apply to arrays given as formal parameters.

64 Out of memory
Attempt made to declare an array or other data object larger than current memory size allows.

65 Pointer value outside user data space
Attempt was made to give a pointer a value outside the permissible range. Note: Subsequent releases of RUN/C may permit override of this safety feature.

99* Failure to release arguments' memory correctly
Should not appear under normal conditions.

191 Illegal termination of expression
Expression or sub-expression ended with character other than a semi-colon, right parenthesis or right bracket.

192* Count on operand stack doesn't equal one
Should not appear under normal conditions.

193 Integer result required for 'switch'

Test expression did not evaluate to an integer type. *Possible solution:* Consider casting expression as an **int**.

194 Argument and parameter counts unequal
A user-defined function was called with more or fewer arguments than the number required by its definition.

195 Missing parameter declaration
A declaration was not found for one of a user-defined function's formal parameters.

254* Unexpected end of program
Should not appear under normal conditions.

266* Aggregate interpreter error
Should not appear under normal conditions.

267 Attempt to redefine aggregate
A **struct** or **union** definition* with this tag name already exists in the program.

268 Aggregate undefined or outside current scope
Attempt to reference a structure or **union** with a tag name when no **struct** or **union** with that tag name has been defined or when the definition is not available to the current portion of the program.

269 Aggregate definition expected
Lack of a subcollection definition when **struct** or **union** occurs within an aggregate without a tag name. If a **struct** or **union** reference is placed within a **struct** or **union**, you must also make reference to a tag name which is defined elsewhere (outside this aggregate) or provide an have an immediate definition for the subcollection. If you have neither, this error will occur.

270 Illegal recursive or forward aggregate reference
Reference is made within an aggregate to an example of the same aggregate in some way other than as a pointer to (or pointer to pointer to ...) of that aggregate. In other words, if you are doing **struct charlie** it is illegal to include within the

---

* The term aggregate refers to concepts which apply to both structures and unions. Aggregates are also known as collections.

definition of **struct charlie** a reference to or a member that is an example of **struct charlie** or an array of **struct charlie** or all other types based on **struct charlie** except for a pointer to **struct charlie** or pointer to pointer to **struct charlie,** etc.

271 Duplicate member name(s) in aggregate

The same name has been used twice within a single aggregate definition. Member names must be distinct only within the same aggregate definition (see "Scope Quirks" in Section 5).

272 Out of aggregate definition space

A limited number of bytes (see **HELP** for current setting) is available for aggregate definitions. There is no limit to the number of aggregates, only to the total number of bytes of the aggregates added together.

273 Illegal or missing type for aggregate member

Any member definition must begin with one of the keywords **char, short, int, unsigned, long, float, double** or the **FILE** macro.

274 Syntax error in aggregate definition

This error message following one of the more specific messages listed below indicates a syntax error related to uses of structures or **unions:**

(274-1) Recursive array

Attempt to include as an aggregate member an array of (or array of array of ...) elements each of which is an example of the aggregate being defined.

(-2) Nonconstant dimension

Any member of an aggregate that is an array must use constant values for all dimensions specified.

(-3) Bad dimension value

Attempt to reference an array member with an unspecified dimension or with a garbage value.

(-4) Noninteger dimension

Dimensions of array members must evaluate to integer types -- or whatever -- no **long, float, double** or other data types.

-5) Lvalue dimension
Any member of an aggregate that is an array must use constant values for all dimensions specified.

-6) Non-unsigned bitfield
All bitfield structure members must be declared **unsigned** (some compilers allow other types).

-7) Bit field not pure unsigned.
Bit field member declaration was for pointer to **unsigned**, array of **unsigned** or some other variant or derivative of **unsigned**; in other words, an object type derived from **unsigned** but not **unsigned** has been detected.

-8) Bit field in union
Bit fields are legal only in structures.

-9) Nonconstant bit field width
The width specified did not evaluate to a constant; an attempt was made to use a variable in a bit field width definition.

-A) Noninteger bit field width
Attempt was made to use a bit field which evaluated to a **long, float, double** or a value other than an integer value.

-B) Bit field width less than 0.
Negative bit fields are of mathematical interest only and are not part of the C language.

-C) Bit width too large
Width greater than 16 bits in current implementation.

-D) Unrecognized member termination
A member specification within an aggregate definition ended with a char actor other than a comma or a semi-colon.

-E) Zero-length aggregate
Attempt apparently made to define an aggregate with no members.

275* Aggregate release failure

Should not appear under normal conditions.

276 Invalid aggregate reference
An attempt has been made to apply the . or -> operators to an object which has not been declared as an aggregate.

277 Invalid aggregate member
The identifier following the . or -> operator did not refer to a valid member of the aggregate in question.

278 Attempt to relocate aggregate
Once declared, a structure or **union** data object has a fixed address in memory; it is illegal to try to change that address.

279 Attempt to take address of bit field
According to formal definitions of the C language, bit field members of structures do not, properly speaking, have accessible addresses.

280 Invalid context for aggregate definition.
Inappropriate environment for a definition. See REP-READ.ME for additional information, if available.

281 'typedef' with initializer
Attempt was made to put a data value into something which is only a data type.

282 'typedef'name in expression
A typedef identifier has been encountered outside the context of a declaration of a data object and not as an operand to **sizeof()** (the only other legal usage).

283 Function call illegal with NULL-valued pointer
Attempt has been made to use a built-in function such as **gets()** that performs input to a user-specified location in memory when that location is zero (NULL). Except when **TRUST** mode is on, RUN/C sets all newly declared pointers equal to NULL. This is done in an effort to trap uninitialized pointer errors in users' programs.

284 Excess parameter declaration
A user-defined function has been given with more parameter declarations than there are formal parameters. For example:

```
func(a,b,c)
int a,b,c,d;
{ ...
```

285 Unusable function pointer value

In RUN/C pointers to functions are not true pointers but follow an internal coding scheme that indicates which of user-defined, built-in or -- in the Professional version -- library function) is desired. When a function pointer value cannot be recognized by the interpreter as indicating a known or plausible function this error is generated. An example of this error would be casting an ordinary character pointer as a function pointer.

380 Initializer type mismatch

An initializer value was found incompatible in type with the data object it was supposed to initialize. For example:

```
int i = "hiya kids";
```

381 Attempt to initialize formal parameter

The declaration of a formal parameter to a user function included an initializer.*

382 Nonconstant initializer for external variable

Variables declared outside function definitions cannot be initialized with expressions which are not constants.

383 Dimension counts unequal in declaration and initializer

The interpreter could not find a working correspondence between the number of dimensions indicated for an array and the "shape" of the initializer provided for that array.

384 Explicit dimension less than initializer dimension

An attempt has been to initialize an array with more elements than have been provided for.

385 Attempt to initialize automatic array or aggregate

Initializers can be provided for arrays, structures and **unions** only when they are declared outside any function or they have been declared to be of

---

* A formal parameter is a variable(s) declared at the beginning of a user function definition to receive values passed by calling functions.

storage class **static.**

386 Attempt to initialize an ineligible object
It is not legal in C to initialize **unions,** a function or a **void** or any variation such as a pointer to a **union.**

901 Unsupported keyword
C keywords currently unsupported include **auto, enum, typedef,** and **entry.** In addition, **register** and **extern** are not appropriate for this implementation of RUN/C.

999* (Got past it all.)
Should not appear under normal conditions.

1000 No more libraries loadable,
The maximum number of libraries which can be loaded at one time is currently 10. This message will appear when an attempt is made to load the eleventh library.

1001 Library is already loaded
attempt has been made to load a library system which is already in memory.

1002 Bad library. Please check library with CHECKRCL.
Several situations may cause this message to appear. 1) The specified library system does not exist; 2) One or more of the files that constitute the library system does not exist; 3) the files which comprise the Loadable Library system exists, but there is another problem which will be described in more detail by the diagnostic program CHECKRCL.EXE.

1003 No functions in library. Please check library with CHECKRCL.
No legitimate functions named in the .RCL file. *Possible remedy:* Check the library with CHECKRCL.EXE.

1004 Not enough memory available to load library
There is no more memory available for the storage of Loadable Libraries. Note: At the end of the information provided by LIBSHOW the quantity of memory available for further library loads is displayed. If at attempt is made to load a library larger than this amount this message will appear.

1005 Bad function declaration. Please check library with CHECKRCL.

One of the functions (which will be pinpointed by CHECKRCL) is incorrectly declared in the .RCL file.

1006 Wrong number of arguments to library function

One of the functions (which will be pinpointed by CHECKRCL) is incorrectly declared in the .RCL file.

1007 Wrong type of argument to library function

One of the functions (which will be pinpointed by CHECKRCL) is incorrectly declared in the .RCL file.

1008 Cannot find COMMAND.COM where specified by COMSPEC

An attempt has been made by the system managing a Loaded Library to access COMMAND.COM from a drive/directory where it was not available. *Possible remedy:* Check to make such the location of COMMAND.COM is correctly specified by the DOS environmental variable COMSPEC. Use the DOS SET command, described in the DOS manual.

Error Messages Without Numbers

(01) **Allmem failure***
Should not appear under normal conditions.

(02) **Allmem failure on return from fork***
Should not appear under normal conditions.

(03) **Allocation problem***
Should not appear under normal conditions.

(04) **Array release failure***
Should not appear under normal conditions.

(05) **Bad file name or no such files**
An illegal character appears in the filename or drive/directory combination. *Suggestion:* See the DOS 2.0 manual for the complete information on legal MS-DOS file names.

(06) **Bad dimension value**
Attempt to reference an array member with an unspecified dimension or with a garbage value. See Error 274 in numbered error messages section for related messages.

(07) **Bad include in line *n***
Either 1) **#include** was called without a file name or 2) **#include** was called without the filename in quotation marks.

(08) **Bad macro name in line *n***
Either 1) no macro name follows **#define**, or 2) the symbol following **#define** is not a valid C identifier, or 3) an attempt was made to redefine a predefined symbols.

(09) **Bit field in union**
Bit fields are legal only in structures. See Error 274 in numbered error messages section for related messages.

(10) **Bit field not pure unsigned**
Bit field member declaration was for pointer to **unsigned**, array of **unsigned** or some other variant or derivative of **unsigned**; in other words, an object type derived from **unsigned** but not **unsigned** has been detected. See Error 274 in numbered error messages section for related messages.

(11) Bit field width less than 0.

Negative bit fields are of mathematical interest only and are not part of the C language. See Error 274 in numbered error messages section for related messages.

(12) Bit width too large

Width greater than 16 bits in current implementation. See Error 274 in numbered error messages section for related messages.

(13) Can't find your 'main'

Interpreter unable to locate the **main** function; program probably does not include one.

(14) Can't find your *name* function

A called function is not located in the program. *Suggestion:* make sure that the name of the called function is exactly the same as that listed in the calling function. Recall that C does not consider **'a'** and **'A'** to have anything in common other than the fact they are both members of the alphabet.

(15) Can't go to EDITOR without a program

An attempt was made to use **EDITOR** without a program in memory. *Recommended action:* Insert a single line of text and try again.

(16) Can't open *filename*

File specified to be **#included** could not be located on the default or specified drive/directory.

(17) Cannot open *filename*

An attempt was made to **SAVE** a file to an illegal filename. See file naming conventions in Section 1 and your DOS 2.0 manual. Another possibility: the disk or directory is full.

(18) Cannot remove filespec

KILL or **UNLINK** failure caused by incomplete specification of the full filename or if the file does not exist on the default or specified drive/directory.

(19) Cannot save entire program

Either 1) insufficient space is available on disk for the program or 2) there is no more space in the directory. *Recommended action:* the **FILES** command will display the number of bytes remaining on the default or specified directory.

(20) Declaration fault in line *n*.
Declaration syntactically wrong or contains syntax errors. It is also possible that the construction is not supported.

(21) DOS error code is *n*
Could occur on return from **SHELL** or **EDITOR**. If this error code appears a reboot of the system is recommended.

(22) Duplicate function *name* in line *n*
A function with the same name as a previously referenced function has been noted during prerun.

(23) Excess macro in line *n*
The limit in the number of macro definitions was exceeded. See **HELP** or **?** for the current maximum number of **#define**'s allowed.

(24) Failure in freeing user-allocated memory.*
Should not appear under normal conditions.

(25) Failure to release #include memory.*
Should not appear under normal conditions.

(26) File *filename* not found
Attempt was made to **#include** a file which does not exist on the default or specified drive/directory.

(27) Function return type mismatch
Attempt was made to pass the wrong type of variable back to the calling function.

(28) goto without target in line *n*
No label is found following the **goto** statement. In RUN/C the identifying label must be on the same logical line as the **goto**.

(29) Ignoring attempt to include "stdio.h"
Attempt was made to #include a file named **stdio.h**. Information usually contained in this "header" file is built into RUN/C; therefore the call is ignored and a warning issued.

(30) Invalid range
Attempt was made to **LIST** (or **LLIST, NLIST** or **NLLIST**) an entirely invalid range of lines. *Recommended action:* Type **EDIT E** to go the built-in editor and end of the program in memory.

(31) Lvalue dimension
Any member of an aggregate that is an array must use constant values for all dimensions specified. See Error #274 in numbered error messages section for related messages.

(32) Memory release failure.*
Should not appear under normal conditions.

(33) No file name given*
Should not appear under normal conditions.

(34) No program in memory
Attempt has been made to use **LIST, LLIST, NLIST, NLLIST** or **SAVE** without a program in memory.

(35) No such files
FILES was called upon to locate a file or group of files which cannot be located in the default or specified drive/directory.

(36) No such insertion point*
Should not appear under normal conditions.

(37) No such line
Attempt was made to **LIST** (or **LLIST, NLIST** or **NLLIST**) a line number which does not exist in the program in memory. *Recommended action:* Use your editor to determine the size and content of program in memory, if any.

(38) No such line to edit
Attempt was made to edit an line which does not exist in the program in memory. *Recommended action:* Use you editor to determine size and content of program in memory, if any.

(39) No valid SET EDITOR argument
SET EDITOR was called, but without an 'alias' name for **EDITOR**. See **EDITOR, SET EDITOR**, Section 4.

(40) No valid SET TRUST argument
SET TRUST mode was called, but without the needed argument (**ON** or **OFF**). See **SET, SET TRUST**, Section 4.

(41) Nonconstant bit field width
The width specified did not evaluate to a constant;

an attempt was made to use a variable in a bit field width definition. See Error 274 in numbered error messages section for related messages.

(42) Nonconstant dimension

Any member of an aggregate that is an array must use constant values for all dimensions specified. See Error 274 in numbered error messages section for related messages.

(43) Noninteger bit field width

Attempt was made to use a bit field which evaluated to a **long, float, double** or a value other than an integer value. See Error 274 in numbered error messages section for related messages.

(44) Noninteger dimension

Dimensions of array members must evaluate to integer types -- or whatever -- no **long, float, double** or other data types. See Error 274 in numbered error messages section for related messages.

(45) Non-unsigned bitfield

All bitfield structure members must be declared **unsigned** (some compilers allow other types). See Error 274 in numbered error messages section for related messages.

(46) Not found

An attempt was made to determine the value of a variable using the **VAL** debugging command when, in fact, the variable was not available to the program. See **VAL** in Section 4 and Section 6.

(47) Please specify line number or range

DEL or **DELETE** was called without a filename.

(48) Program name not found

At attempt was made to access a filename which does not exist on the default or specified drive/directory. Recommended action: Use **FILES** or **SHELL** at the Command Level to check the exact name of the file(s) you are attempting to access.

(49) Recursive array

Attempt to include as an aggregate member an array of (or array of array of ...) elements each of

which is an example of the aggregate being defined. See Error 274 in numbered messages section for related messages.

(50) Right braces exceed left braces in line *n*
The total number of right braces in the user program at the point the message was generated exceeds the number of left braces -- a *de facto* illegal condition in a C program.

(51) SET without argument -- ignored
The only **SET** command currently available is **SET EDITOR**.

(52) Terminating comment*
Should not appear under normal conditions.

(53) Terminating quoted string*
Should not appear under normal conditions.

(54) Too many labels as of line *n*
Number of labels exceeds the maximum combination of **goto/labels**. See **HELP** or **?** for on-line information on the maximum number of **goto/label** constructs.

(55) Unmatched braces
There are an uneven number of curly braces { and } in the program.

(56) Unmatched brackets
There are an uneven number of square brackets [and] in the program.

(57) Unmatched end of comment
An end-of-comment symbol (*/) was found on a line without a beginning comment (/*). See Section 5 for a discussion on RUN/C's comment conventions.

(58) Unmatched parenthesis
There are an uneven number of parenthesis (and) in the program.

(59) Unrecognized command
Should appear only at Command level (after the RUN/C **Ok** prompt). This means you have typed something which is not recognized as a RUN/C Command. (See **CLIST** to review valid RUN/C commands.) This may also be a syntax error such

as:

LOAD"TEST
(a space is needed after **LOAD**)

(60) Unrecognized member termination
A member specification within an aggregate definition ended with a character other than a comma or a semi-colon. See Error 274 in numbered error messages section for related messages.

(61) Unrecognized SET command
The only currently available **SET** command is **SET EDITOR**.

(62) Use AUTO to begin your program*
Should not appear under normal conditions.

(63) Variable release failure.*
Should not appear under normal conditions.

(64) Zero-length aggregate
Apparent attempt made to define an aggregate with no members. See Error 274 in numbered error messages section for related messages.

Other Screen Messages

(65) Appending line*
Should not appear under normal conditions.

(66) Autosave every *n* commands
Reference: **AUTOSAVE**, Section 4.

(67) Autosave OFF
Reference: **AUTOSAVE**, Section 4.

(68) Editor set to *filename*
Reference: **EDITOR**, Section 2; **SET EDITOR**, Section 4.

(69) Line *n* unchanged*
Should not appear under normal conditions.

(70) Loading line *n*
Reference: **LOAD**, Section 4.

(71) MORE (hit any key to continue)
Reference: **FILES**, Section 4.

(72) *nnnnnn* bytes free
Tells the number of bytes currently available for additional lines of user program and for program data requirements.

(73) Printer echo off
Indicates their either the **SET PRINT OFF** command or the <Ctrl-p> or <Ctrl-PtrSc> toggle has been pressed. Reference: **SET PRINT**, Section 4.

(74) Printer echo on
Indicates their either the **SET PRINT ON** command or the <Ctrl-p> or <Ctrl-PtrSc> toggle has been pressed. Reference: **SET PRINT**, Section 4.

(75) Printer not ready
Indicates that a printer is not available to the system after an attempt has been made to toggle on the printer echo or to use **LLIST** or **NLLIST**, described in Section 4.

(76) Press any key to return to RUN/C . . .
Reference: **SHELL**, Section 4; "Editing Programs", Section 2.

(77) Reload PROGRAM.C (Y/N)
Reference: **SHELL**, Section 4.

(78) Save to PROGRAM.C (Y/N)?
Reference: **SHELL**, Section 4.

(79) Saving line *n*
Reference: **SAVE**, Section 4.

(80) Tabs now set every *n* columns.
Reference: **TABS**, Section 4.

(81) Trace OFF
Reference: **TRACE**, Section 4.

(82) Trace ON
Reference: **TRACE**, Section 4.

(83) Type 'exit' when done with external operation(s)
Reference: **SHELL**, Section 4; "Editing Programs", Section 2.

APPENDIX B. SUGGESTED READINGS

1. A Book on C, Kelley, Al and Pohl, Ira, The Benjamin/Cummings Publishing Co., Inc., Menlo Park, CA, 1984.

2. C Primer Plus, Waite, Mitchell and Prata, Stephen and Martin, Donald, Sams, Howard W. & Co., Inc., 1984.

3. The C Programmer's Hardbook, Hogan, Thom, Brady Communications Co., Inc., Bowie, MD, 1984.

4. C Programming Guide, Purdum, Jack, Que Corp., Indianapolis, IN, 1983.

5. The C Programming Language, Kernighan, Brian W. and Ritchie, Dennis M., Prentice-Hall, Englewood Cliffs, NJ, 1978.

6. The C Programming Tutor, Wortman, Leon A. and Sidebottom, Thomas O., Brady, Robert J. Co., Bowie, MD, 1984.

7. The C Puzzle Book, Feuer, Alan R., Prentice-Hall Software Series, Englewood Cliffs, NJ, 1982.

8. A C Reference Manual, Harbison, Samual P. and Steele, Guy L. Jr. Prentice-Hall, Inc., Englowood Cliffs, NJ, 1984.

9. Disk Operating System, Version 2.00, International Business Machines Corp., Boca Raton, FL, 1983.

10. The IBM Personal Computer From the Inside Out, Sargent, Murray III and Shoemaker, Richard L., Addison-Wesley Publishing Co., Reading, MA, 1984.

11. Learning to Program in C, Plum, Thomas, Plum Hall, Cardiff, NJ, 1983.

12. Programming in C, Kochan, Stephen G., Hayden Book Co., Inc., Hasbrouck Heights, NJ, 1984.

13. Programming in C on the IBM PC, Cummings, Bryan J. and Pollack, Lawrence, Prentice-Hall, Englewood Cliffs, NJ, 1984.

14. Understanding C, Hunter, Bruce, Sybex, Inc., Berkeley, CA, 1984.

This page intentionally left blank.

APPENDIX C. SOME COMMON C BUGS

This collection of C bugs was contributed by Randall Swan, President of the C-Team, Inc., a New York-based software development and teaching company. Randy is also president of the C language section of the New York Amateur Computer Club.

Copyright (c) 1985, The C-Team, Inc.*

Uninitialized variables, especially uninitialized pointers.
Uninitialized auto variables will have the junk values of the stack memory in which they're created. This leads to dangling or floating pointers where a value is placed into or retrieved from an arbitrary address. You may not notice the strange results.

Off-by-one errors where a string or any array has advanced one address or location too far.
Since there is absolutely no bounds checking in C, you'll eat into some other area of memory. For instance:

```
char s[10];
s[10] = '\0';
```

Arrays always have their first element in **[0]**, their second element in **[1]**, and their last element in **[n-1]**.

Not closing out comments with an ending ***/**
This causes the next statement or even the rest of your program is eaten up by the comment. The compiler (or interpreter) will often complain about a missing semicolon or missing closing curly brace even though that is not the problem.

The missing semicolon.
The compiler or interpreter usually complains about illegal syntax in the next statement line, misleading you.

---

* Some of the topics below are expansions from Appendix A.9 in Learning to Program in C by Plum-Hall, pg 10-13.)

Missing curly braces.

What you think is proper indentation doesn't make a bit of difference. For instance:

```
while( .. )
            stmt1;
            stmt2;
    stmt3;
```

stmt1 is the only thing tied to the **while** loop. **stmt2** isn't.

A glibly typed semicolon after a **while** or **for** or **if** statement.

This often occurs with no space following the right parenthesis. For instance:

```
while( ... ); {
      stmt;
      }
```

or

```
while( ... );
{
      stmt;
}
```

The indentation doesn't make a bit of difference here either. The compiler won't ever complain to you on this one.

break does not get you out of a block of code controlled by an **if**.

Break can only take you out of a **for, while, do ... while,** or **switch.** For instance:

```
if(...) {
      stmt1;
      if(...) break;
      stmt2;
      } stmt3;
```

break does not get you out to **stmt3,** it throws you out beyond it somewhere.

c = getchar() returning into a **char** instead of an **int.**

The EOF test will always fail and your program will sit there and never end. **c** must be declared as an **int.**

Slash / instead of backslash \, for instance /n for \n.

Using = instead of ==

This causes an unwanted assignment to occur inside an expression instead of evaluating properly. Assignments of non zero values will always evaluate true.

Forgetting about the **\0** null byte in terminating strings, or clobbering it in a gone-one-too-far situation.

Not dimensioning strings large enough to provide for the null byte or to provide for longer user input than expected.

An improper semicolon right after the declared function name in the function body. For instance:

```
foo(a,b);
char a,b;
{
      ..
}
```

Confusing apostrophes (single quotes) with quotation marks (double quotes).

'\0' is one byte and **"\0"** is a two byte string.

Using **getchar** to input a single character.

When the user hits <enter>, another byte immediately follows as a newline **\n** often clobbering the first or, worse, affecting the next input. The best technique is to grab input as a string as in:

```
char stuff[81];
gets(stuff);
if( *stuff == ...)
```

This will always preserve the first byte and turn a newline into a null byte at the end of the string.

A similar situation occurs with **scanf("%d",&n)**, where a number is entered followed by <enter>. The <enter> affects the next input. It's usually better this way :

```
char stuff[81];
int n;
gets(stuff);
n = atoi(stuff);
```

Forgetting that **scanf** must always be passed addresses.

Using elaborate **scanf** input conversion arguments.

Expect anything from interactive input. Avoid **scanf** if possible. Use string input and convert.

Not declaring functions of the proper type that return pointers.

In particular, forgetting declarations like:

```
char *malloc();
```

in the function that places a call to **malloc**. If a function is returning anything other than an **int** or **char**, make sure you declare it for use as the right type in the function that calls it otherwise the returned value will be taken as an **int**.

Forgetting that there is absolutely no bounds checking in C.

Since you can point to anywhere in memory, you can subscript anywhere as well. Remember **num[-500]** is a perfectly reasonable relative address meaning ***(num-500)**. This would be much less obvious perhaps in something like:

```
i = -500;
num[i] = ...
```

There is no precedence to the postfix (after) auto increment; it always happens after the (lvalue) expression is evaluated.

For instance ***p++** and ***(p++)** are identical. The postfix auto increment **p++** will always happen last. This is one circumstance where parenthesis do not force evaluation.

The evaluation of an expression of the form

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

is an unhappy situation and is always ambiguous.

Refer to page 50 K&R. It is not clear whether the auto increment occurs after the right side evaluation, the rvalue, or after the entire statement is complete. Is it **a[1] = 1;** or **a[2] = 1;**? You can bet it will behave differently on various compilers but always consistently with the same one for are an expansion. **a[i++] = i;** can also be a problem as the compiler is free to evaluate the expression in any order it chooses.

ints and pointers are not necessarily the same size objects. In fact it is possible that pointers to different type objects will have different sizes.

It's usually faster and preferable to use casts rather than **unions**.

unions tend to compound the obscurity of an expression. To get an **int** into a pointer, cast it. For instance:

```
char *p;
p = (char *) 0x100;
```

p now points to address 100 hex and the compiler won't complain.

Optimizing for the wrong purpose.

Optimizing will always obscure the clarity of your code. A highly ingenious C expression will not always generate ingenious object code. Only things that occur very frequently should be optimized. Loops and frequently called functions are the best candidates. Consolidate constants wherever possible. Prioritize whether you want to optimize for time or space. Check the assembler output from the compiler to determine if efficient code is happening.

static variables with initializers in functions are not initialized every time the function is called.

They are initialized only the first time encountered and are ignored on subsequent calls. This isn't a problem in **main()** where static is used to allow array initialization, although it's intent and purpose were meant to be quite different.

The macro preprocessor does not know C and will do the substitutions you indicate whether it makes good C or not.

Arguments in a macro are not variables in C, so they have no type. Use the compiler option (or in RUN/C the D(Dump) command) to see the effect of your macro substitutions.

Local variables are automatics, i.e., declared inside of functions, and are safer but slower.

Global variables are externals, i.e., declared outside of functions, and are faster but can have adverse side effects. It is generally poor practice to declare all your indices globally above **main()** for speed considerations unless there is a particular need for speed over safety.

Leaving out a **break** in a switch/case statement and falling through to other case's inadvertently.

It is preferable to place blanks around arithmetic operators.

Here's why:

```
int x,a,*p;
x=a/*p;
  comment accidentally begins; it eats p & the
next line */
x = a / *p;
      /* is no problem nor is  x=a/(*p); */
```

Parameter arguments in function calls are not evaluated in any particular order.

The parameter list is not an expression. It may be looked at right to left or left to right, whatever the compiler writer wants to do. Most compilers will place things on the stack in reverse order, but it can vary how things go on the stack. That's not up to C. For instance:

```
foo(++bar, bar);
```

The second **bar** may happen before or after the auto increment.

An idiom for an auto increment by two might be:

```
i++;
```

but is very dubious to some compilers. This will certainly work:

```
i++ + ++i;
```

but is far worse than simply:

```
i++, i++;
```

Using the logical Boolean OR || instead of the bitwise OR |

.

If you want to OR bits together, the statement:

```
x = foo || bar;
```

doesn't do it. **x** will have the value of 1 if either **foo** or **bar** are nonzero, otherwise **x** is 0.

**** STACK OVERFLOW ****

This is a pernicious and often impossible to diagnose bug. Nonetheless, many of these run time errors occur because there are too many local variables declared inside of a function. Large array declarations, in particular those larger than 1K or 2K will

do it. Stacks do not usually accommodate this quantity of storage. The solution is to reduce the internally declared variables, possibly making them external by lifting them outside of the function as globals or passing them in as parameters.

If it is impossible to reduce the internally declared variables, then there is usually a compiler option and/or a run time option to increase the working stack size. The side effect is often slightly slower execution and additional memory consumption. It may even be handled by tweeking some magic external variable in code to your compiler such as **_STACKSIZE = 4096;** which will override the default. The only way to know is to dig this baby out of your compiler's reference manual somewhere.

Another disadvantage to altering the default stack size is that deep recursion may fail and bottom out.

Stack overflow may also occur when the wrong number or the wrong size arguments are used in function calls. Make sure the correct number of parameters exist and that the data types in the calling function match the data types in the formal parameter list.

APPENDIX D. THE BASIC_C GRAPHICS LIBRARY

RUN/C Professional for the IBM PC includes a Loadable Library system named GRAPHLIB which contains a number of graphic functions which are similar to those built into IBM PC BASIC. These functions are derived from C Source's BASIC_C function library. See the end of this section for details on obtaining the source and object code to the BASIC_C Graphics function library.

The following functions and their requirements are supplied in GRAPHLIB. (Techniques for loading libraries and descriptions of the components of a Loadable Library system can be found in Section 5 of this manual):

int ARC(int,int,int,int,double,double,double)	args:(x,y,rad,color,asp., st.,end)
int CIRCLE(int,int,int,int)	args: (x,y,radius,color)
void COLOR(int,int,int)	args: (foreground,background,border)
int DRAW(*,zero or several arguments)	args: (string, var args)
int ELLIPSE(int,int,int,int,double)	args: (x,y,radius,color,aspect)
int FASTFILL(int,int,int,int)	args: (x1,y1,fill,boundary)
void FREE_PIC(*)	args: pic_pointer
* GET_PIC(int,int,int,int)	args: (x1,y1,x2,y2)
void INITBC()	args: none (inits screen, must execute before other funcs)
int LINE(int,int,int,int,int,int)	args: (x1,y1,x2,y2,color,box)
int LOCATE(int,int)	args: (row,column)
int PAINT(int,int,int,int)	args: (x,y,paint,boundary)
int PATCH(int,int,int,int,int)	args: (x1,y1,x2,y2,color)
int POINT(int,int)	args: (x,y)
int PSET(int,int,int)	args: (x,y,color)
int PUT_PIC(int,int,*,int)	args: (x1,y1,pic_pointer,action)
int SCREEN(int,int,int,int)	args: (mode(0-2),burst(0-1),apage,vpage)

Three of these functions are general to creating a graphic environment:

* INITBC
* SCREEN
* COLOR

The applications of these and other functions are described below in

alphabetical order.

In addition there are some constant values which can be used in conjunction with any graphics function which accepts relative coordinates (**x** and **y** values) such as **LINE**. These constant values are:

Current position	**CURR_PT**	-1347
Step rate	**STEP**	-2415
Fill box	**FILLED_BOX**	1
Leave box empty	**EMPTY_BOX**	2
Draw line	**A_LINE**	0

These values may be supplied as constants or defined either in the program or in a header file. For your convenience the header file graphics.h is included on the distribution disk and contains these definitions.

FUNCTION

```
int ARC(x, y, radius, color, aspect, start, end)
    int x, y, radius, color;
    double start, end, aspect;
```

REMARKS

x	is any integer expression.
y	is any integer expression.
radius	is an integer expression in the range 0-154.
color	is an integer expression in the range 0-15.
aspect	is a **double** with a lower limit of 0.004 and no upper limit.
start	is a **double** expression in the range -6.283 to 6.283 (except for -0.0).
end	is a **double** expression in the range -6.283 to 6.283 (except for -0.0).

ARC draws an arc of an ellipse with the given aspect and center, beginning at the point corresponding to the angle equal to **start**, and ending at the point corresponding to the angle equal to **end**. The radius equals the length of the major axis of the ellipse. A pie slice is drawn if either **start** or **end** or both are negative.

RETURNS

NULL

EXAMPLE

[1]

(BASIC)

```
10 CIRCLE (159, 99) 50, , 0, 3.1415
20 REM  This displays a half-circle of radius 50
30 REM  in the foreground color.
```

(BASIC_C)

```
ARC(159, 99, 50, 1, 1.0, 0.0, 3.1415);
/*  This displays a half-circle of radius 50 */
```

[2]

(BASIC)

```
10 CIRCLE (159, 99), 50, , 0, -1.5708
20 REM  This displays a wedge in quadrant 1
30 REM  in the foreground color with radius 50
```

(BASIC_C)

```
ARC(159, 99, 50, 1, 1.0, 0.0, -PI / 2.0)
/* This displays a wedge in quadrant 1 in the */
/* foreground color with radius 50 */
```

[3]

```
main()      /* arc.c */
{
int i,j;
double start, end, delta_angle;

start = 0.0;
delta_angle = 3.1415 / 36.0;
end = 6.283;
      INITBC();
      SCREEN(1,1,0,0);
      cls();
      ELLIPSE(159,99,50,3,1.2);
      for (i = 0; i < 71; i++) {
 ARC(159,99,50,i&3,1.2,start,end-=delta_angle);
      }

      for (i = 200; i > 0; i -= 5) {
            LINE(60,15,60+i,15,(i+1)&3,0);
      }
      for (i = 170; i > 0; i -= 3) {
            LINE(260,15,260,15+i,(i+1)&3,0);
      }
      for (i = 200; i > 0; i -= 5) {
            LINE(60,185,60+i,185,(i+1)&3,0);
      }
      for (i = 170; i > 0; i -= 3) {
            LINE(60,15,60,15+i,(i+1)&3,0);
      }
      getch();
      SCREEN(2,0,0,0);
}
```

FUNCTION

```
int CIRCLE(x, y, radius, color)
    int x, y, radius, color;
```

REMARKS

x	is any integer expression.
y	is any integer expression.
radius	is any integer expression in the range 0-154.
color	is any integer expression in the range 0-15.

CIRCLE draws an circle with the given center.

RETURNS

NULL on error

EXAMPLES

[1]

(BASIC)

```
10 CIRCLE (159, 99), 50
20 REM  This displays a circle of radius
30 REM  50 in the foreground color
```

(BASIC_C)

```
CIRCLE(159, 99, 50, 1)
/*  This displays a circle of radius 50 */
```

(Continued next page)

[2]

```
main()      /* circle.c */
{
INITBC();
cls();
SCREEN(1,0,0,0);
CIRCLE(130,95,15,1);
CIRCLE(130,95,45,2);
CIRCLE(130,95,60,3);
CIRCLE(130,95,75,4);
CIRCLE(130,95,90,5);
CIRCLE(130,95,30,6);
CIRCLE(130,95,15,7);
LOCATE(1,1);
puts("PRESS ANY KEY");
getchar();
}
```

SEE ALSO

ARC, ELLIPSE

FUNCTION

```
COLOR(foreground, background, border)
    int foreground, background, border;
```

foreground	is any integer expression between 1 and 16.
background	is any integer expression between 1 and 16.
border	is any integer expression between 1 and 16.

REMARKS

COLOR sets the colors for foreground, background and border (also called palette). **COLOR** has the same format for text and graphics modes; i.e., the foreground color can be specified in graphics mode.

In both modes all three arguments must be specified. The values set for the arguments produce the same values as in BASIC. The following colors are allowed:

0	Black	**8**	Gray
1	Blue	**9**	Light Blue
2	Green	**10**	Light Green
3	Cyan	**11**	Light Cyan
4	Red	**12**	Light Red
5	Magenta	**13**	Light magenta
6	Brown	**14**	Yellow
7	While	**15**	High intensity White

The integers 16-31 cause the characters to blink on and off.

You may find it useful to use the preprocessor to **#define** names for all the color numbers and keep them handy in a header file such as graphics.h.

The foreground argument must be specified in graphics mode (set with **SCREEN**). It should be a number between 1 and 3 specifying a color from the current palette.

The palette has the same range of values as the border. If the palette number is even, it corresponds to palette 0. An odd palette number corresponds to palette 1. The colors are as follows:

COLOR	PALETTE 0	PALETTE 1
1	Green	Cyan
2	Red	Magenta
3	Brown	White

Note: In text mode, **COLOR** makes it possible to change the color of characters generated through **printf**-type statements if ANSI.SYS is loaded.

RETURNS

NULL, if error

EXAMPLE

(BASIC)

```
10 COLOR 1,4,7
20 REM this sets the foreground to blue,
30 REM the background to red
40 REM and the border to white
```

(BASIC_C)

```
#define BLUE     1
#define RED      4
#define WHITE    7

COLOR(BLUE,RED,WHITE);
```

SEE ALSO

PAINT

FUNCTION

```
int DRAW(fmt, args)
    str fmt;
    int args;
```

REMARKS

fmt is any string expression
args is any integer expression

DRAW is similar to the DRAW command of BASIC. The first argument is a format string, like that found in BASIC's PRINT and PRINT_USING functions. The string is followed by any arguments that are needed.

The format string may contain the following special characters:

Movement commands:

Un	up
Dn	down
Ln	left
Rn	right
En	diagonally up and to the right
Fn	diagonally down and to the right
Gn	diagonally down and to the left
Hn	diagonally up and to the left

where n is an integer specifying the distance to move.

Mx,y moves to an absolute or relative location. If **x** or **y** has a plus (+) or minus (-) sign in front of it the move is relative, otherwise it is absolute.

Commands that may prefix a move command are:

B	move, but don't plot any points
N	move, but return to the original position when finished.

The following commands are also available:

Cn	set the color to n.
Sn	set the scale factor to n, where 4 is the defaults size.
X	the next argument is a pointer to a format string to be executed. The difference between BASIC's implementation and BASIC_C's for this command is

that **X** can only point to another format string; thus the macro can only have constants.

Any argument can have an n which replaces the number in the format string with the equals sign (=). This tells **DRAW** to get the actual value as from the next argument in the list of arguments passed to the function. See an illustration of this technique below.

RETURNS

NULL on error.

EXAMPLES

```
main()     /* draw.c */
{
char *string;
int i, color, x, y, bcolor, fcolor;

     SCREEN(1,0,0,0);
     DRAW("BM160,70");
     for (i = 0; i++ < 2; DRAW("S4 BM20,60")) {
          DRAW("C2 U= R= D= L=",60,60,60,60);
          getchar();
          DRAW("BU30 C1 E= F= G= H=",30,30,30,30);
          getchar();
          DRAW("BR30 C3 NU10 NL10 ND10 NR10 NH10 NG10
          NE10 NF10");
          getchar();
          }
}
```

FUNCTION

```
int ELLIPSE(x, y, radius, color, aspect);
    int x, y, radius, color;
    double aspect;
```

REMARKS

x	is any integer expression.
y	is any integer expression.
radius	is an integer expression in the range 0-154.
color	is an integer expression in the range 0-15.
aspect	is a **double** with a lower limit of 0.004 and no upper limit.

ELLIPSE draws an ellipse with the given aspect and center. The radius is the length of the major axis. If the aspect is less than 1, then the **x**-axis is the major axis. If the aspect is greater than 1, then the **y**-axis is the major axis.

RETURNS

NULL on error

EXAMPLES

(BASIC)

```
10 CIRCLE (159, 99), 50, , , , 0.5
20 REM  This displays an ellipse with a 25 x-radius
30 REM  and a y-radius of 50 in the foreground color
```

(BASIC_C)

```
ELLIPSE(159, 99, 50, 1, 0.5);
/*  This displays an ellipse with a 25 x_radius */
/*  and a y-radius of 50 */
```

SEE ALSO

ARC, CIRCLE; example under **ARC**

FUNCTION

```
int FASTFILL(x, y, fill, boundary)
    int x, y, fill, boundary;
```

REMARKS

This is a faster version of **PAINT**. Its primary limitation is that it cannot handle complex figures (such as those with islands or peninsulas).

The **x** and **y** coordinates specify the beginning points for the **FASTFILL** to begin.

The **fill** parameter is a number in the range of 0 to 3. In medium resolution, this color is the color from the current palette as defined by the **COLOR** function. Zero (0) is the background color. The default is color 3, the foreground color. In high resolution, **fill** 0 is black and 1 is white.

boundary is the color of the edges of the figure to be filled in in the range 0 to 3. It is the **boundary** value which helps the function identify the area you want "filled". In high resolution mode **fill** and **boundary** should be the same.

The starting point of **FASTFILL** must be inside the figure to be painted.

EXAMPLE

```
main()     /* fastfill.c */
{
int i,j;
     INITBC();
     SCREEN(1,1,0,0);
     for(i=0;i<10;++i){
          COLOR(1,2,0);
          CIRCLE(100, 80, 80, 3);
          PAINT(90,2,3,4);
          cls();
          COLOR(1,2,0);
          CIRCLE(100, 80, 80, 3);
          FASTFILL(90,2,3,4);
          }
}
```

SEE ALSO

PAINT

FUNCTION

```
void FREE_PIC(pic_ptr)
     char *pic_ptr;
```

REMARKS

FREE_PIC frees memory allocated by a GET_PIC. This function is used upon completion of any use of GET_PIC. A FREE_PIC with the name of the address pointer must be used for each instance of GET_PIC. This is illustrated in the example shown in GET_PIC, the mv_block.c sample program.

SEE ALSO

GET_PIC, PUT_PIC

FUNCTION

```
pic *GET_PIC(x1, y1, x2, y2)
     int x1, y1, x2, y2;
     char *pic;
```

REMARKS

x1,x2	is an integer expression in the range 0-320 or 640.
y1,y2	is an integer expression in the range 0-200.
GET_PIC	is similar to the GET statement in BASIC in its graphics mode. GET_PIC also works in text mode.

After specifying a rectangle on the screen GET_PIC will allocate memory and then read in the data from the designated screen rectangle. The two points can be any two opposite corners of a rectangle.

General discussion
Together GET_PIC and PUT_PIC can read and write blocks of data to and from the screen. A block of screen data is termed a "pic" (short for picture).

The two coordinates must be on the screen, no error checking is currently performed for illegal points, and the result of using such points will be undefined. These points do not currently modify the last referenced point.

The functions GET_PIC and PUT_PIC are very similar to BASIC's GET and PUT graphics statements. The major difference is that GET_PIC automatically allocates the block of memory to hold the screen information, so you don't have to calculate anything.

A block of screen data is organized in the following manner: At the start is a structure called a PIC which contains information needed to handle the screen data. A PIC has two logical parts -- at the start is a header of information (also called a PIC) describing the 2-dimensional array which holds the actual screen data. The header contains the array sizes for the two-dimensional array, and it also contains the number of "slop bits" on each side of the array.

Example (assuming that the screen is in medium resolution):

```
pic = GET_PIC(10, 50, 15, 53);
```

pic would be a pointer to a (in this case) 16-byte chunk of memory where the first four bytes would be the PIC header,

and the remaining 12 bytes would be a 3(col) x 4(rows) two dimensional array.

The actual memory would look as shown below where every box represents a byte. In the two dimensional array containing the screen data, each bit is represented by a 1, 0 or -, where a '-' implies a don't care condition (this is a "slop bit"). A period is also included to show each nibble in the byte.

```
-------------------------------------------------------------
| _xbytes = 3 | _yrows = 4 | _lopen_bits = 4 | _ropen_bits = 6 |
-------------------------------------------------------------
|----.0101|0101.0000|01--.----|   Note that the # of '-'s on the left
----------------------------      corresponds to the # of _lopen_bits,
|----.0101|0000.0101|01--.----|   and the # of '-'s on the right
----------------------------      corresponds to the # of _ropen_bits.
|----.1101|0101.0101|00--.----|
----------------------------
|----.1101|0101.0101|00--.----|
----------------------------
```

Since we are in medium resolution (2 bits/pixel), it can take up to three bytes to hold the required 6 pixels. **GET_PIC** reads the screen a byte at a time, and therefore may have 'slop bits' on either side of the array. We read and write data to the screen a byte at a time. Now suppose we wish to write the pic to location (40, 23) on the screen (**PUT_PIC(40, 23, 'P')**):

```
-------------------------------------------------------------
| _xbytes = 3 | _yrows = 4 | _lopen_bits = 0 | _ropen_bits = 10|
-------------------------------------------------------------
|0101.0101|0000.01--|----.----|
----------------------------
|0101.0000|0101.01--|----.----|
----------------------------
|1101.0101|0101.00--|----.----|
----------------------------
|1101.0101|0101.00--|----.----|
----------------------------
```

EXAMPLE

```
/*
        The FREE_PIC function
        releases memory allocated by GET_PIC.
        It should eventually be used to every time GET_PIC is used.
*/

/*
        When using GET_PIC the TRUST-ME flag must be set to ON
        This is done below
*/

main() /* mv_block.c */
{
char *pic1, *pic2, *pic3, *pic4;
int p1x, p1y, p2x, p2y, p3x, p3y, p4x, p4y;

INITBC();
SCREEN(1,1,0,0);
runc("SET TRUST ON");
cls();
COLOR(2,0,3);
PATCH(100, 60, 158, 98, 2);
PATCH(100, 102, 158, 140, 3);
PATCH(162, 60, 220, 98, 3);
PATCH(162, 102, 220, 140, 2);
pic1 = (char *)GET_PIC(99, 59, 159, 99);     p1x = 99;  p1y = 59;
pic2 = (char *)GET_PIC(161, 59, 221, 99);    p2x = 161; p2y = 59;
pic3 = (char *)GET_PIC(99, 101, 159, 141);   p3x = 99;  p3y = 101;
pic4 = (char *)GET_PIC(161, 101, 221, 141); p4x = 161; p4y = 101;
```

(Continued next page)

```
        while(1) {
                if (p1x != 161) {
                        if (p1x-- < -60) p1x = 320;
                        PUT_PIC(p1x, p1y, pic1, 'P');
                        }
                if (p4x != 99) {
                        if (p4x++ > 320) p4x = -60;
                        PUT_PIC(p4x, p4y, pic1, 'P');
                        }
                if (p2y != 101) {
                        if (p2y-- < -40) p2y = 200;
                        PUT_PIC(p2x, p2y, pic2, 'P');
                        }
                if (p3y != 59) {
                        if (p3y++ > 200) p3y = -40;
                        PUT_PIC(p3x, p3y, pic3, 'P');
                        }
                if ((p1x == 161) && (p4x == 99) && (p2y == 101) && (p3y == 59)) break;
                }
        LOCATE(2,2);
        SCREEN(2,0,0,0);
        cls();
        FREE_PIC(pic1);
        FREE_PIC(pic2);
        FREE_PIC(pic3);
        FREE_PIC(pic4);
        runc("SET TRUST OFF");
        }
```

SEE ALSO

FREE_PIC, PUT_PIC

FUNCTION

void INITBC( )

REMARKS

INITBC performs initialization of parameters used by other BASIC_C functions. Specifically it determines the current data segment, determines whether a color card or monochrome card is being used, forces the CRT mode and clears the screen.

INITBC should be executed one time only prior to the use of any BASIC_C graphics function in your program.

EXAMPLE

See other example programs.

Function

```
int LINE(x1, y1, x2, y2, color, box)
    int x1, y1, x2, y2, color, box;
```

REMARKS

x1,x2,y1,y2	is any integer expression.
color	is an integer expression in the range 0-3.
box	is any integer expression.

LINE draws lines and boxes between the two points specified by **x1,y1** and **x2,y2**. If **box** equals 0 a line is drawn. If **box** equals 1 a box is drawn and filled in. Any other non-zero value causes the outline of a box to be drawn.

For example, assume you are in medium resolution w/palette set at 1:

```
LINE(23, 23, 10, 10, 1, 0);
```

draws a green line from (23,23) to (10,10).

```
LINE(CURR_PT, 46, 46, 2, 0);
```

draws a red line from (10,10) to (46,46) assuming that **CURR_PT** has been properly defined.

```
LINE(CURR_PT, STEP, 10, 10, 3, 1);
```

draws a brown box at (46,46) to (56,56), assuming that **CURR_PT** and **STEP** have been properly defined.

```
LINE(CURR_PT, 30, 30, 1, FILL_BOX);
```

draws and fills a green box at (56,56) to (86,86) assuming that **CURR_PT** and **FILL_BOX** have been properly defined.

Although you can draw a box and fill it in with **LINE**, you will find the process relatively slow. If you need better speed, use the **PATCH** function.

EXAMPLES

```
#include "graphics.h"

main()     /* line.c */
{
     INITBC();
     SCREEN(1,0,0,0);
     LINE(73,73,10,10,1,A_LINE);
     getkey();
     LINE(CURR_PT,96,96,2,A_LINE);
     getkey();
     LINE(CURR_PT,STEP,10,10,3,EMPTY_BOX);
     getkey();
     LINE(CURR_PT,30,30,1,FILLED_BOX);
     getkey();
}

getkey()
{
     LOCATE(1,1);
     puts("Press any key");
     getchar();
}
```

FUNCTION

```
LOCATE(row,column)
    int row, column;
```

REMARKS

row	is an integer expression in the range 1-25.
column	is an integer expression in the range 1-80.
LOCATE	moves the cursor to the screen position indicated by row and column.

LOCATE is the graphics equivalent to the **locate** function built into RUN/C Professional.

RETURNS

NULL, on error.

EXAMPLES

```
main()     /* locate.c */

{
INITBC();
cls();
SCREEN(1,0,0,0);
CIRCLE(130,95,15,1);
CIRCLE(130,95,45,2);
CIRCLE(130,95,60,3);
CIRCLE(130,95,75,4);
CIRCLE(130,95,90,5);
CIRCLE(130,95,30,6);
CIRCLE(130,95,15,7);
LOCATE(1,1);
puts("PRESS ANY KEY");
getchar();
}
```

FUNCTION

```
int PAINT(x, y, paint, boundary)
    int x, y, paint, boundary;
```

REMARKS

PAINT fills in an area delimited by boundary color, with the paint color. The **x** and **y** coordinates specify the beginning points for the **PAINT**'ing to begin.

The **paint** parameter is a number in the range of 0 to 3. In medium resolution, this color is the color from the current palette as defined by the **COLOR** statement. Zero is the background color. In high resolution, **paint** 0 is black and 1 is white.

boundary is the color of the edges of the figure to be filled in in the range 0 to 3. It is the **boundary** value which help the function identify the area you want "painted".

In high resolution mode **paint** and **boundary** should be the same.

The starting point of **PAINT** must be inside the figure to be painted.

EXAMPLE

```
main()     /* paint.c */
{
INITBC();
SCREEN(1,1,0,0);

COLOR(1,2,0);
CIRCLE(120, 100, 100, 3);
PAINT(100,100,3,4);
}
```

SEE ALSO

FASTFILL, COLOR

FUNCTION

```
int PATCH(x1, y1, x2, y2, color)
    int x1, y1, x2, y2, color;
```

REMARKS

PATCH writes a rectangular patch of color on the screen. Any or all of the patch may be off the screen without adverse effects.

EXAMPLES

```
main()     /* patch.c */
{
INITBC();
SCREEN(1,0,0,0);
LINE(123,123,30,30,1,0);
printf("press key once");
getchar();
LINE(123,123,30,30,3,2);
printf("Here comes PATCH");
getchar();
PATCH(123,123,30,30,3);
printf("press--LINE fills block with new color");
getchar();
LINE(123,123,30,30,2,1);
printf("press key for last time");
getchar();
}
```

FUNCTION

```
int POINT(x,y)
    int x,y;
```

REMARKS

x is any integer expression in the range of 0-320 for medium resolution or 0-640 for high resolution.
y is any integer expression in the range of 0-200.

POINT returns the value of the pixel's color at the specified absolute coordinates.

RETURNS

-1 on error

EXAMPLE

[1]

(BASIC)

```
10 IF POINT(12,40) = 0 THEN CLS
20 REM This clears the screen if color is equal
to 0
```

(BASIC_C)

```
if(POINT(12,40) == 0)
      cls();
```

[2]

```
main()
{

int ret;
INITBC();
SCREEN(1,0,0,0);

LINE(173,173,30,30,3,1);
getchar();
ret = POINT(120,40);
printf("%d", ret);
getchar();
}
```

FUNCTION

```
int PSET(x,y,color)
    int x,y, color;
```

REMARKS

x	is an integer expression in the range of 0-320 for medium resolution and 0-640 for high resolution.
y	is an integer expression in the range of 0-200.
color	is an integer expression in the range 0-3.

PSET colors a pixel at the position specified by the absolute coordinates.

There is no default value for the color argument. A value of 3 is the equivalent of BASIC's default value for this function.

If the coordinates specify a position off the screen, PSET returns NULL (0). Otherwise it colors the pixel and returns success (1).

RETURNS

NULL, on error.
1, for success.

EXAMPLE

[1]

(BASIC)

```
10 SCREEN 1,1
20 COLOR 1,1
30 FOR VAL1 = 1 TO 10
40 PSET(18-VAL1,VAL1)
50 PSET(VAL1,VAL1
60 NEXT VAL1
50 REM This draws a white X on a blue background
```

(BASIC_C)

```
COLOR(3,1,1);
for(val1 = 1; val1 <= 10; val1++){
      PSET(18-val1,val1,3);
      PSET(val1,val1,3);
      }
/* This draws a white X on a blue background. */
```

[2]

```
main()     /* pset.c */
{

int ret;
INITBC();
SCREEN(1,0,0,0);
LINE(173,173,30,30,3,1);
getchar();
ret = POINT(120,40);
printf("%d", ret);
getchar();
}
```

FUNCTION

```
int PUT_PIC(x1, y1, pic, action)
    int x1, y1, action; PIC *pic;
```

REMARKS

PUT_PIC is similar to BASIC's PUT graphics command. It will write a picture to the screen that has been "grabbed" by the GET_PIC function. You may put a "pic" anywhere you like. A pic can be drawn globally. This is, it is possible that part -- or even all -- of the figure will be off the screen.

If you try to use a graphics pic in text mode, nothing will happen, except a null will be returned.

RETURNS

NULL, on error.

EXAMPLE

See GET_PIC

SEE ALSO

FREE_PIC, GET_PIC

FUNCTION

```
int SCREEN(mode, burst, apage, vpage)
    int mode,burst,apage,vpage;
```

REMARKS

mode	is an integer expression in the range 0-2.
burst	is an integer expression in the range 0-32,767.
apage	is an integer expression in the range 0-3.
vpage	is an integer expression in the range 0-3.

SCREEN provides the facility of BASIC's SCREEN statement in setting the parameters for the screen. The first two arguments are of primary interest, since they select the resolution and whether the screen is to be displayed in color or monochrome. In graphics mode the last two arguments are ignored.

The mode can be set as described above, for a specific resolution:

0	text mode
1	medium resolution
2	high resolution (monochrome only)

The second argument in the function is either true (color) or false (monochrome). Any even number (such as 2) will yield monochrome; any odd integer in this position will yield color.

Once medium resolution (40 columns) graphics has been selected it is necessary to go to high resolution (80 columns) before returning to 80 column text mode.

RETURNS

NULL

EXAMPLES

See other examples.

A Special Offer for RUN/C Users

We at Age of Reason Co. are pleased to be able to include the quality graphics functions from C Source, Inc. within RUN/C and RUN/C Professional. In addition to providing some nice color and enhancing our artistic abilities, graphics functions in a C interpreter are ideal for mastering the subtle features of the C programming language. This is because of the immediate feedback provided by these visual routines.

Please keep in mind, however, that you cannot compile programs using these functions without having either the source code or object code for these functions.

Fortunately, C Source, Inc. of Kansas City, Mo. makes these functions available for several MS-DOS C compilers at a reasonable price in both source and object code from. Best of all, **there are no royalties on the use of any BASIC_C or BASIC_C graphics functions in executable code form.**

There are two packages available:

* The BASIC_C function library contains all the BASIC_C Graphics functions (available in RUN/C Professional's GRAPHLIB) plus approximately 80 functions which are similar to many Microsoft(R) and PC BASIC 1.0 statements and functions. It includes routines which provide BASIC-like random and sequential file I/O, string-handling, graphics, screen and keyboard control. This package is available for $125 to RUN/C users, a $50 discount off the retail price.

  The BASIC_C routines makes it convenient for C programmers to access BASIC files with floating point data and move them over to C. In addition, the BASIC_C documentation contains a C tutorial designed for the C novice who is coming from a BASIC background.

* For those who are only interested in giving their compiled code access to the functions contained in GRAPHLIB, the BASIC_C Graphics library in both source and object form is available at a reduced price ($60).

C compilers supported by the BASIC_C and BASIC_C graphics function libraries include Lattice (all models), Microsoft 3.0 and IBM C, DeSmet, Computer Innovations and Mark Williams C.

To order either library at the special discount rate provided to RUN/C users, just fill out and send the coupon on the opposite side of this page to the people at C Source. Or, if you are in a hurry, just call 816-353-8808 (and have your RUN/C serial number ready).

BASIC_C Function Library

Order Form

The **BASIC_C**(tm) full library with approximately 100 functions including approximately 15 graphics functions with source costs $125 (a $50 savings). The **BASIC_C Graphics** library (approximately 15 graphics functions with source) costs $60.

Please fill out the information below and and return it to:

C Source, Inc.
12801 Frost Rd.
Kansas City, Mo.
64138

Name: _____

Address: _____

Address: _____

City: _____ State: _____ Zip: _____

Country: _____

RUN/C Serial number (must be supplied): _____

Yes, I want source and object code from BASIC_C. Please send:

__ The full **BASIC_C** library including graphics ($125)
__ The **BASIC_C Graphics** library only ($60)

The C compiler which I will be using **BASIC_C** with is:

__ DeSmet C __ Mark Williams C
__ Lattice C __ Microsoft C 3.0 / IBM C

Preferred method of payment:

__ Check (enclosed) __ MasterCard __ Visa __ COD

Credit Card Information

Account number: _____

Expiration date: _____

Signature (required for credit cards): _____

Daytime telephone number: _____

There is no charge for UPS ground within North America. For COD add $3, for UPS Blue add $3, for UPS Red add $10.

Shipping instructions: __ UPS __ UPS BLUE __UPS RED

APPENDIX E. DIFFERENCES BETWEEN RUN/C 1.32 and RUN/C PROFESSIONAL 1.0

For those interested, this documents some of the changes made between RUN/C -- The C Interpreter version 1.32 and RUN/C Professional version 1.0.

Added features

* A full screen editor with search-and-replace and block manipulation facilities has been integrated into the product; the extended line editor has been removed. This is fully described in Section 2 of this document; individual commands are described in Section 4.

* In RUN/C Professional a Loadable Library system has been added. This is fully described in Section 5 of this document.

* Multiple breakpoints, single stepping and other debugging commands have been added to RUN/C Professional. These are fully described in Section 6 of this document; individual commands are described in Section 4.

* The **runc()** function has been added to facilitate debugging and the allow execution of some interpreter commands by the program being interpreted.

* The editing commands **INS, INSERT, DEL, DELETE, MERGE** and **AUTO** have been removed.

* The editing commands **E, EDIT ERROR** and **EDIT END** have been added.

* A **PREC** command which displays the precedence table has been added.

* Help is now available after a breakpoint or <Ctrl-c> and for the Interactive mode in RUN/C Professional.

* On the IBM PC version, the function key assignments have been changed to correspond to changes in the editor and other commands.

* A problem with **else if** has been resolved.

* Uninitialized pointers are now trapped unless the **TRUST** flag has been set on. All newly declared pointers are initialized to NULL and RUN/C will reject any attempt to use an input function such as **gets** with a NULL-valued pointer argument. (Any value other than NULL that is not in the user data space will also be flagged as being unavailable to the user program.)

* An **EXEC** command has been added to allow **LOAD** and **RUN** of a program through a single command.

* The **SET TRUST** toggle is now available under program control using the **runc()** function.

* **typedefs** are now available to the program being interpreted.

* A **WAIT** message appears momentarily at the top of the screen whenever a **NEW** is executed.

* <Ctrl-PtScr> and <Ctrl-p> are now available within RUN/C and work essentially as they do in DOS -- toggling the printer echo on or off.

* Attempts to use <Ctrl-p>, <Ctrl-PtScr>, **LLIST** or **NLLIST** when a printer is not available will be trapped as interpreter errors.

* Function pointers can be now be declared and functions can be executed indirectly via function pointers. This applies to built-in functions, user-defined functions and RUN/C Professional library functions. Note: Function pointer values are not actual memory addresses of object code. In fact, they are not actual memory addresses at all (but that's another story). Consequently casting a function pointer value to some other pointer value and attempting to use it (as with **gets**) will probably result in Error 65 (pointer outside program data space), but may produce unintended results without being trapped. To summarize: function pointer values may be assigned to other function pointers or dereferenced for execution. No other use is recommended.

* RUN/C now supports legal C constructions such as:

```
*(puts)("Hi there");
```

which are not supported by all MS-DOS C compilers.

* Trapping for conditionals without parentheses (like **if, while**) has been added.

* The transcendental math functions (such as **tan**) have been removed from the RUN/C Professional version of the interpreter and instead have been made a Loadable Library named MATHLIB. These functions are documented individually in

Section 4.

* A number of graphics functions from C Source's BASIC_C library have been collected and placed in a Loadable Library named GRAPHLIB. These are documented in Appendix D.

* The distribution disk system has been redesigned. This is described in RC-READ.ME on disk 1 of the distribution disk. This "read me" file will also contain any late changes and should be read by all users.

* An immediate mode has been added to allow execution of built-in functions from the command line.

* An interactive debugging mode has been added to the mini-menu (accessible after a break or <Ctrl-c>) to allow for the polling and changing of available variables before continuing with execution of the program.

* User-defined functions can be executed by name from the interactive mode.

* Library functions loaded into RUN/C Professional can be executed by name both from the interactive mode and using the immediate mode.

* Automatic variables are now local to blocks rather than functions. This conforms to K&R C.

* The keywords **auto, extern** and **register** are now allowed in programs, although they are not meaningful in most situations. See details in Section 4.

* All references to collections (**structs** and **unions**) have been changed to aggregates to reflect the terminology being incorporated in the emerging ANSI C standard.

* The documentation has been edited and reprinted. Several new chapters have been added where appropriate.

Repairs and bug fixes

* In a **struct** definition when the opening left curlie brace was on a new line the interpreter failed to understand the definition. This has been corrected.

* A problem with **else if** has been resolved.

* Trapping keywords other than **sizeof** in expressions has been added.

* Profiler performance and accuracy has been improved.

* Documentation for **movmem** has been added to Section 4.

* Angle brackets in **#include** statements (i.e., **#include <header.h>**) are now permitted.

* **sizeof()** now returns correct results for abstract data types such as **int**.

* Casts to aggregates and casts to pointers to aggregates are now supported.

Known Bugs and Deficiencies

* Preprocessor commands other than **#include** and **#define** are not currently supported.

* Built-in functions cannot currently be declared or redefined.

* It is not yet possible for user-defined interpreted functions (as opposed to loadable library functions) to take a variable number of arguments.

* **typedefs** cannot be nested and **typedefs** that are arrays cannot be used in forming new objects that are arrays of those **typedefs**.

* **statics** are local to functions rather than to blocks.

(See also Section 7 for additional discussion on RUN/C conventions and limitations.)

ADDENDA-A FOR VERSION 1.10 of RUN/C Professional

Highlights of the Release

Although this is considered an interim release, the features contained in RUN/C Professional 1.10 have been considerably enhanced over previous versions.

—> Loadable Library support has been added for the popular Microsoft C compiler, version 3 and 4, as well as version 3 of Lattice C.

—> A new program, MAKELIB.C greatly aids in the creation of Loadable Library .EXE and .MAP files.

—> The amount of memory left after a library has been loaded has been made user-definable via a command line argument.

—> Many of the program's maximum (i.e., maximum number of **goto's**, etc.) have been made user-adjustible via command line arguments.

—> Program and library loading times have been improved by 300-400 percent.

—> Several new functions and keyboard commands have been added, see next page for contents and page numbers.

—> Many reported bugs have been repaired and RUN/C's ability to handle various types of **typedef** and arrays as well as objects returning **void** have been improved. See the end of Addenda A for a list of new messages, changes from previous versions and our revised bug and deficiency list.

—> The large RCP-READ.ME file has been consolidated and incorporated in this addenda.

New Addresses:

Lifeboat Associates
55 S. Broadway
Tarrytown, NY 10591

1-800-847-7078

Age of Reason Co.
151 First Ave. #123
New York, NY 10003

1-212-334-1234

Thank you for your continued interest, suggestions and support.

Contents

Using Loadable Libraries 3
Expanded Compiler Support for Loadable Libraries 5
Command Line Options 9
New Keyboard Commands 12
New or Renamed Functions 13
- **bas_peek** 13
- **bas_poke** 15
- **peek** 19
- **poke** 22
- **SET BREAK** 25
- **SET CONTINUE** 27
- **SET END** 28
- **SET EXCHAR** 29
- **SET QUIET** 30

Files on the Distribution Disk 32
- For IBM-compatible version 32
- For "generic" MS-DOS version 38

New and Newly Documented Messages 39
Catalog of Update Changes 41
- Version 1.03 to 1.10 41
- Version 1.00 to 1.03 44

Known Bugs and Other Anomalies 47

Copyright (c) 1986, Age of Reason Co. All rights reserved.

Using Loadable Libraries

Occasionally we get comments that remind us that we still have a lot to learn about the art of documentation.

'No-problem' #1: Some have incorrectly concluded from our manual that the source code of a function library is needed in order to create a Loadable Library. Not so. All you need is the .OBJ or .LIB file of that library which must have been compiled under the large model of a supported C compiler.

Look over Section 5 of the RCP manual for examples of extracting the functions you want from .LIB files and creating the .RCL, .EXE and .MAP files which are required for a Loadable Library.

'No-problem' #2: If, after you load a library there is not enough memory left for your program to load, what do your do? The problem here is that your program should have been loaded first, before the library or libraries. The reason for this is that once a library is loaded something less than 40,000 bytes are left for your program and data or -- if you loaded your program first -- for program data.

In this release we have made a change to make this even more of a non-problem. Among the command line options which have been added with this release is the -i option which stands, roughly, for interpreter program/data space. This may now be adjusted between 10k bytes and 640k bytes, the default being 40k bytes, as was the case with the previous release. In all cases an integer (representing k bytes) must be specified. Thus if you wanted to amount of memory reserved for your program and data space to be 256k rather than the default 40k the command-line invocation would be:

```
rcp -i256
```

Important: This option should be used only if you plan to load a function library.

It is also worth noting that there is in the second and all subsequent loaded libraries an overhead of about 80 bytes per function take from the interpreter program/data space. Thus a second loaded library with 100 functions would reduce the

working space for the interpreted program by (100 x 80) about 8,000 bytes.

Another explanation: If your Loadable Library is too large to load, and your interpreter program (and its data-space requirement) is relatively small, try setting -i with a value less than the default 40k. If your Loadable Library loads with room to spare, but you run out of room for your interpreted program or its data, try setting -i with a value above the default. (Note also that some duplicated overhead memory may be saved by combining two or more Loadable Libraries, if possible, into one.

Expanded Compiler Support for Loadable Libraries

The RUN/C Professional Loadable Library system can now be used with .OBJ and .LIB files created by the following compilers:

* Lattice C/Microsoft C, version 2.nn
* Lattice C, version 3.nn
* Microsoft C, version 3.nn
* Microsoft C, version 4.nn

In each case the large model option must be selected.

Now that we are supporting several different versions of several different compilers, the method of creating a loadable library has changed slightly from that described in Section 5 of the RUN/C Professional manual. These changes are described below. However, all of the RUN/C library commands such as **LIBLOAD** and **LIBSHOW** remain as described.

Best of all, previously created Loadable Libraries should still work!

New compiler-specific linkable files

Loadable Libraries are created by linking .OBJ or .LIB files created with a supported compiler and model with two compiler-specific .OBJ files.

In previous releases of RUN/C Professional, libraries were created and linked with two files that we provided: RCLMAIN.OBJ and RCL2.OBJ (see the RCP manual Section 5, "Creating your own loadable libraries"). Those two object files have now been superseded by a pair of object files for each compiler that we support:

Lattice 2.0+	RCA-LAT2.OBJ	(formerly RCLMAIN.OBJ)
	RCB-LAT2.OBJ	(formerly RCL2.OBJ)
Lattice 3.0+	RCA-LAT3.OBJ	
	RCB-LAT3.OBJ	
MS 3.0	RCA-MS3.OBJ	
	RCB-MS3.OBJ	
MS 4.0	RCA-MS4.OBJ	
	RCB-MS4.OBJ	

Note: These .OBJ files along with a sample .BATch file for creating a loadable library under the specified compiler are contained in the appropriate subdirectory on RCP distribution disk 2. See pp. 32-38 for a description of the contents of distribution disks.

The new system

Assume that you have a file called TESTLIB.C which you want to transform into a loadable library. Listed below are the new procedures for compiling and linking with each of the supported compilers. You must still create the file TESTLIB.RCL as explained in the manual -- that part hasn't changed.

Remember, after you complete the process below check every Loadable Library that you create or change with the CHECKRCL.EXE program before loading it into RUN/C -- that part hasn't changed either.

Compiling and linking with Lattice 2.0+

The following routine assumes that the Lattice 2 .LIB files LCL.LIB and LCML.LIB are located in the directory: **\lc2.**

```
lcl testlib -ml
lc2 testlib
link \lc2\c1+rca-lat2+rcb-lat2+testlib, testlib/m, testlib, \lc2\lcml+\lc2\lcl
```

Compiling and linking with Lattice 3.0+

The following routine assumes that the Lattice 3 .LIB files LCL.LIB and LCML.LIB are located in the directory: **\lc3.**

```
lcl -ml testlib [note that "-ml" now comes before the file]
lc2 testlib
link \lc3\c1+rca-lat3+rcb-lat3+testlib, testlib/m, testlib, \lc3\lcml+\lc3\lcl
```

Compiling and linking with Microsoft 3.0

The following routine assumes that the Microsoft 3 .LIB and **#include** files are in the directory **\ms3.**

```
msc testlib.c /AL /X /I \ms3;
link rca-ms3+rcb-ms3+testlib, testlib/m, testlib, \ms3\
```

Notes:

1. The switch **/AL** chooses the large model of the compiler (far data and code pointers) -- this switch must be used -- Loadable Libraries support only large model functions.

2. The switches /X and /I \ms3 indicate that any #include files needed by TESTLIB.C will all be found in the directory \ms3. If this is not true, modify these commands accordingly.

3. Use the linker program LINK.EXE provided on the Microsoft distribution disks (version 3.01 or later is required).

Compiling and linking with Microsoft 4.0
The following assumes that Microsoft's 4 compiler .LIB files and #include files are in the directory \ms4.

```
msc testlib.c /AL /X /I\ms4;
link rca-ms4+rcb-ms4+testlib, testlib/m, testlib, \ms4\
```

Notes:

1. The switch /AL chooses the large model of the compiler (far data and code pointers) -- this is mandatory for use with RUN/C.

2. The switches /X and /I \ms4 indicate that any #include files needed by TESTLIB.C will all be found in the directory \ms4. If this is not true, adjust accordingly.

3. Use the linker program LINK.EXE provided on the Microsoft distribution disks (version 3.51 or later).

To make life a little complicated we are providing on distribution disk No. 2 a program called MAKELIB.C which can be run under RUN/C to create a .BAT file (called MAKELIB.BAT) which will take into account the compiler you are using, any additional command line arguments you want passed to the compiler, and the like. After you have run this program you can adapt the generated .BATch file to your needs -- we don't guarantee the results, but it should be easier than starting from scratch. Some general purpose instructions for using MAKELIB.C follows:

Using MAKELIB.C
The MAKELIB system is used to create a DOS compatible batch file which in turn can be used with your compiler and your source code and/or object code and/or .LIB files to create a Loadable Library .EXE and .MAP files. MAKELIB.C is to be run under RUN/C (what else?), but also could be compiled.

MAKELIB.BAT -- the file created by MAKELIB.C -- makes the following assumptions. If any of these are not valid, modify the

resultant BATch file accordingly.

1. The compiler and linker are in your path.
   If they are not, and you are using DOS 3.0+, then explicitly name the *drive:dir\subdir* path where the compiler and/or linker may be found. If you are using DOS 2.0+ then you must put both the compiler and linker in the path.

2. The compiler's pre-processor will find all **#include** files that you are using, without having to name the directories where these files reside.

   If not, then use the **-i\dirname** option for Lattice C, or the **/I \dirname** option for Microsoft C on each compile line explicitly.

3. The linker will find the libraries in the default directory, or the libraries' location is built into the object files (Microsoft only).

   If the linker can't seem to find the named libraries on the linker command line, then their locations must be explicitly stated.

4. If you are using Lattice 3.nn, be sure that the start-up object code (normally named "C.OBJ") is so named, and corresponds to the large-model start-up code (formerly CL.OBJ).

In general, the MAKELIB.BAT file should be seen as only a starting point for modifications. Due to the wide variety of compiler locations, options, etc. all possibilities cannot be accounted for, so we are providing this only as a convenience, i.e, without support.

Once you have your MAKELIB.BAT file and your have used it to create the Loadable Library .EXE file, use CHECKRCL.EXE to verify that your library system (i.e., the "RCL", "MAP" and "EXE" files) is truly "loadable" into RUN/C Pro. See the manual for detail on using CHECKRCL and creating Loadable Libraries.

Another thought: The nice thing about MAKELIB.C and its child, MAKELIB.BAT, is that once you get it working right, creating a loadable library will become almost an automated process except for creating the .RCL file. We're think about that problem, too.

Command Line Options

(The following discussion replaces the same topic in the RUN/C Professional manual, pp. 1-18 through 1-20.)

When RUN/C is invoked there are several *command line options* which can be specified. Certainly the most familiar command line option is one which specifies the name of a program file to be immediately loaded and run. For example, if RUN/C is invoked with the command:

```
A>RCP HELLO
```

the current default drive/directory will be searched for a file named HELLO.C. If it is found, the program will be loaded and run. If the program executes without error or <Ctrl-c> interruption (and does not chain to another program, as with **runc("exec otherpgm");**) RUN/C will execute to DOS after the run. If there is no such file, RUN/C will display the message:

```
Program hello.c not found
Ok
```

Other command line options include changing the size of the program stack (described below), the maximum number of lines that RUN/C can interpret (-l option) and the maximum number of **#defines** that can occur in an interpreted program (-d option). In addition, it is possible for the user to adjust maximum number of:

* Program blocks (-b, portions of the program contained within curly braces)
* Functions (-f)
* Memory allocations (-m, the number of memory allocations the interpreted program can have active at one time
* Labels (-g, for use with **goto** statements)
* Bytes available for aggregate (-a, structure and union) definitions.

In addition, the amount of memory reserved for use by interpreted programs when an object library has been loaded can be adjusted with a command-line option (-i). See discussion on pp. 3-4.

These memory allocation options can be used in any combination.

The chart on the next page shows the default, minimum and maximum settings for the various run-time switches.

Command Line Argument Default, Minima and Maxima

Flag	Meaning	Default	Minimum	Maximum
-a	Aggregate definition space*	3000	1500	29970
-b	Number of blocks	500	100	999
-d	Number of #defines	25	25	999
-f	Number of functions	100	10	999
-g	Number of labels	10	10	999
-i	Interpreted program/data space after a library is loaded**	40	10	640
-l	Number of program lines	2000	100	10000
-m	Memory allocations	50	10	999

* Aggregate definition space is allocated in "slot" units of 30 bytes each. Every structure or union definition requires one slot for overhead and one additional slot for each member, a total of 60 bytes. The maximum number of aggregate definitions depends on the way the user divides this space -- i.e., it could be many small structures and/or unions or one large on or anything in between.

** In K bytes, thus **-i256** installs any loaded libraries above the first 256k bytes of available RUN/C program and data space.

The following "maximums" are not adjustable:

```
Automatic variables          200
(active at one time)

Global & static variables    200
(in entire program)
```

Changing default command-line argument settings

If the amount of memory available to your system is insufficient for RUN/C and your program to comfortably reside together (for example, if you are using DOS 3.2 on a system with only 320k of memory), you can gain additional space for your programs by reducing the amount of memory allocated for various command line options as well as reducing -- with care -- the stack size (discussed in Section 1 of the RUN/C manual) to, say, 5,000-10,000 bytes.

Example: modifying the maximum number of program lines

The number of program lines can be changed by invoking RUN/C Professional in this way:

```
RCP -Lnnnn
```

This changes the maximum number of lines to *nnnn*. As of this writing

the default is 2,000, and the specified number must be in the 100 to 9,999 range.

Another example: modifying the maximum number of #defines
You can modify the maximum number of **#define**'d terms with:

```
RCP -Dnnn
```

This sets the maximum number of **#defines** in a program to *nnn*. The default is 25, but the maximum number of **#define**'s can be between 10 and 999.

Note: The reason for all these maximums is that a small amount of memory is preallocated when RUN/C is invoked. The amount of memory varies considerably -- nearly 200 bytes are allocated for each **#define** while only 8 bytes are preallocated for each program line.)

Another note: The above chart and other discussions regarding default "maximums" for various parameters and minimums and maximums of same are as of this writing. Please see the READ.ME file which comes with your distribution disk any possible changes to this information.

Changing the defaults "permanently"
If you wish to change these maxima on a continuing basis, consider creating a batch file containing something like:

```
RCP =10000 -1250 -d10
```

If you invoke RUN/C with these command line arguments the maximum number of lines in the program will be set at 250, the maximum number of **#define**'s at 10 and the stack at 10,000 bytes -- smaller, but still sufficient for many programs. On our system with these command line flags set we gain approximately 25,000 additional bytes for our programs.

The command line changes will remain in effect as long as RUN/C remains loaded into memory. For your convenience this batch program is on the distribution disk under the name RUN.BAT.

New Keyboard Commands

Several new keyboard commands have been added for the convenience of the user. These are:

! The Exclamation Point Command

When in the RUN/C command mode (i.e., the Ok prompt or the Interactive mode (the >> prompt)) whenever you press the exclamation mark your most recent command will appear on the prompt line. This make it possible to re-execute the command or -- in a simple sense -- edit the command by destructive backspacing through the command until you reach the portion which you want to save.

This is particularly helpful when using RUN/C's immediate mode for executing and testing functions.

<Ctrl-p>
<Ctrl-PrtSc>

This DOS keyboard toggle has been activated within RUN/C. The user can now echo screen I/O at the printer. This toggle can be activated or deactivated at the command and interactive mode levels.

When activated the message **Printer echo on** will appear. When deactivated the message will be: **Printer echo off.**

PURPOSE Reads value of a location in memory.

FORM

```
long address;
char data;

data = bas_peek(addr);
```

USAGE **bas_peek** looks at an absolute address in memory and returns its contents. The values are read in 8-bit bytes.

In the above sample:

`long addr;`
declares a variable to hold the target memory address.

`char data;`
declares a variable to hold the value currently held in that memory location.

`data = bas_peek(addr);`
reads the contents of memory at addr, assigning the returned value to data.

NOTES

* In previous versions of RUN/C this function was named **peek**. The current name acknowledges the fact that this routine is similar to BASIC's PEEK statement.
* The data returned value will be in the range 0 to 255.
* Be sure to use an **L** (denoting **long**) after the value assigned to the memory address element if it is a constant.
* This function can be used to examine any location in memory, regardless of whether the "SET Trust me" flag is on or off.
* Programs that used the **peek** and/or **poke** functions under previous versions of RUN/C should be modified to use **bas_peek** and/or **bas_poke**. No other changes should be required.

RETURNS Contents of memory location, if successful.

(Continued next page)

bas_peek

EXAMPLE

```
main()
{
/* bas_peek.c -- illustrates use of bas_peek
(a la a memory dump) */
int i, j, v;
long a, a2;

cls();
puts("Hex/ASCII dump of 256 bytes of memory");
printf("Absolute decimal address? ");
scanf("%ld",&a);
for (i = 0; i < 256; i += 16, a += 16) {
     printf("%05lx: ",a);
     a2 = a;
     for (j = 0; j < 16; j++, a2++)
          printf("%02x ",bas_peek(a2));
     a2 = a;
     for (j = 0; j < 16; j++, a2++) {
          v = bas_peek(a2);
          if ( (v < 32) || (v > 127) )
               printf(".");
          else
               printf("%c",v);
          }
     printf("\n");
     }
}
```

SEE ALSO bas_poke, peek, poke

PURPOSE Write a byte into a location in memory.

FORM

```
long address;
char data;

bas_poke(address, data);
```

USAGE **bas_poke** puts a numeric value in the range of 0-255 into an absolute memory address in the system.

In the above sample:

`long address;`
declares a variable to hold the target memory address.

`char data;`
declares a variable to hold the character value being sent to the memory address.

`bas_poke(address,data);`
puts the value assigned to **data** into the specified address in memory.

NOTES

* In previous versions of RUN/C this function was named **poke**. The current name acknowledges the fact that this routine is similar to BASIC's POKE. The example given below will only work on the IBM PC and close compatibles. The significance of particular addresses may vary from computer to computer, even in situations which claim complete PC compatibility. See the technical documentation for your computer for information on key memory addresses and the values which they are designed to hold.

* When you put a value into a memory location using **bas_poke**, you are changing the contents of memory. With **bas_poke** you can change any memory location in your system regardless of whether the "SET TRUST mode" is on or off -- and quite possibly crash the system in the process. Critical elements of the PC -- the screen, program memory, system memory, etc. -- can be accessed via **bas_poke**. RUN/C does not check the address you are using for safety or correctness. We quote from IBM's BASIC manual: "Don't go POKEing around" where you don't belong.

* Be sure to use an L (denoting long) after the value given for the memory address if it is a constant.

* Programs that used the peek and/or poke functions under previous versions of RUN/C should be modified to use bas_peek and/or bas_poke. No other changes should be required.

RETURNS Nothing.

EXAMPLE
```
main() /* bas_poke.c -- demos use of bas_poke */
     /* monochrome screen attributes*/
{
     int i;
/* color use 753665L; monochrome use 720897L */
     int attribute_byte;
     int c;
     long a=720897L;
     cls();
     puts("\n\nSelect your mode:");
     puts("\t1 Blinking");
     puts("\t2 Blinking and highlighted");
     puts("\t3 Blinking and underlined");
     puts("\t4 Blinking and reversed");
     puts("\t5 Reversed video");
     puts("\t6 Underlined");
     puts("\t7 Highlighted");
     puts("\t8 Highlighted and underlined");
     puts("\tEsc to exit gracefully");
```

(program listing continues next page)

```
        if(( c = getchar()) != EOF)
              switch(c)
              {
                    case '1': attribute_byte = 212;
                          break;
                    case '2': attribute_byte = 222;
                          break;
                    case '3': attribute_byte = 129;
                          break;
                    case '4': attribute_byte = 240;
                          break;
                    case '5': attribute_byte = 112;
                          break;
                    case '6': attribute_byte = 1;
                          break;
                    case '7': attribute_byte = 95;
                          break;
                    case '8': attribute_byte = 121;
                          break;
                    case 27: exit();
                    default:
                          puts("\nNot an acceptable
                          answer. Run me again.");
                                exit();
              }

        cls();
        puts("Why don't word processing programs
        use these features?");
        for (i = 0; i < 53; i++, a += 2)
              bas_poke(a, attribute_byte);
}
```

SEE ALSO bas_peek, peek, poke

PURPOSE Reads a series of bytes from anywhere in memory.

FORM
```
unsigned segment;
unsigned offset;
char local_array[integer constant];
unsigned nbytes;

peek(segment, offset, local_array, nbytes);
```

USAGE peek reads an area of memory into a local array set up by the programmer.

In the above sample:

`unsigned segment;`
Sets up the "segment" in segment:offset form of the area of memory to be read.

`unsigned offset;`
this is the "offset" in segment:offset form of the area of memory to be read.

`char local_array[nbytes];`
Creates a character array to contain the values in memory which are to be read.

`unsigned nbytes;`
Creates an **unsigned** variable to contain the number of bytes to be read.

`peek(segment, offset, local_array, nbytes);`
A portion of memory is read, the amount determined by the value of **nbytes**, into an array named **local_array** beginning with the specified segment:offset address.

NOTES * **peek** and **poke**, below, address memory in *segment:offset* form in a way which circumvents memory addressing limitations inherent in 8086-based *small model* C compilers, whose pointers could only reach within a designated area of memory 64k bytes in size. With these functions, one can reach anywhere in memory to look at data and/or to change it (using **poke**).

The advent of *large model* C compilers which by definition allow pointers to point anywhere in memory makes the value of these functions

somewhat academic, particularly with RUN/C which is itself a large-model program that supports only four-byte pointers.

* Programs that used the **peek** and/or **poke** functions under previous versions of RUN/C should be modified to use **bas_peek** and/or **bas_poke**. No other changes should be required.

RETURNS Nothing.

(Continued next page)

EXAMPLE

```
main(){
     /* peek.c -- demonstrates use of the new
     peek() and poke() functions, this example
     will work only on IBM PC's and close
     compatibles! */

char scr_buf[4096]; /* holds the screen data */

int i;
     /* start of video memory is B000:0 for
     monochrome systems and B800:0 for color
     systems (these addresses are in
      segment:offset format in hexadecimal) */

unsigned video_seg = 0xB000;
     /* use 0xB800 for color */

unsigned video_off = 0;
     /* now copy the video memory into a local
     array where it can be manipulated */

peek(video_seg, video_off, scr_buf, 4096);

     /* now change a column of the screen image
     copy to the letter: 'A'. Remember that each
     line of the screen is 160 bytes long where
     every other byte is a character
     "attribute", so 80 bytes in from the start
     of the array is a character on the
     40th column of the first line */

for (i = 0; i < 25; ++i)
     scr_buf[i*160 + 80] = 'A';

     /* now write it back to video memory so
     that it will appear on the video screen */

poke( video_seg, video_off, scr_buf, 4096);
}
```

SEE ALSO poke, bas_peek, bas_poke

poke
Function

PURPOSE Writes to a specified memory address.

FORM

```
unsigned segment;
unsigned offset;
char local_array[nbytes];
unsigned nbytes;

poke(segment, offset, local_array, nbytes);
```

USAGE **poke** writes a specified number of bytes from a local array set up by the program into memory.

In the above sample:

`unsigned segment;`
: This is the "segment" in segment:offset form of the area of memory to be read.

`unsigned offset;`
: This is the "offset" in segment:offset form of the area of memory to be read.

`char local_array[nbytes];`
: Sets up a character array to hold **nbytes** being written to memory.

`unsigned nbytes;`
: Creates an unsigned variable to contain the number of bytes to be written to memory.

`poke(segment, offset, local_array, nbytes);`
: Beginning with the segment:offset address, **nbytes** of memory will be replaced with the contents of **local_array**.

NOTES

* **poke** and **peek**, above, addresses memory in *segment:offset* form in a way which circumvents memory addressing limitations inherent in 8086-based *small model* C compilers, whose pointers could only reach within a designated area of memory 64K bytes in size. With these functions, one could reach anywhere, to look at data and/or to change it.

  The advent of *large model* C compilers which by definition allow pointers to point anywhere in memory makes the value of these functions somewhat academic, particularly since RUN/C is a large model system with four-byte pointers.

* In the example above memory will be filled beginning with the address and extending to the top of memory for **nbytes**. If **local_array** has no contents or only garbage values, that is what memory will be filled with. Thus practically no effort is required to write over important areas of memory. User beware!

* Programs that used the **peek** and/or **poke** functions under previous versions of RUN/C should be modified to use **bas_peek** and/or **bas_poke**. No other changes should be required.

RETURNS Nothing.

EXAMPLE See example under **peek**.

SEE ALSO poke, bas_peek, bas_poke

PURPOSE Allows the setting of a *macro* string which will be executed immediately after the program received a <Ctrl c> or <Ctrl Break> from the keyboard.

FORM [1] SET BREAK @ SCREEN(0,0,0,0);

[2] SET BREAK

USAGE **SET BREAK**, in conjunction with **SET CONTINUE** and **SET END**, allows the user to more fully control the interpreter after a break has been detected by allowing an arbitrary macro string to be associated with a **BREAK**. The **SET BREAK** macro string will be executed each time you generate a *break* from the keyboard, from the program using the **runc()** command or when an error is detected by the interpreter. Once set, the **SET BREAK** macro will remain in effect until replaced by another string or a null string as shown in Form [2], above.

SET BREAK and the other environmental commands were added primarily to aid in the debugging of graphics applications. To give a simple example, returning from the GRAPHLIB IBM-compatible graphics mode to 80 column text mode requires the following two commands either from the program or the immediate or the interactive mode (the immediate mode command is shown here):

```
@ SCRN(2,0,0,0); SCRN(0,0,0,0);
```

And you can automatically return to text mode upon any break in the execution of the program if the following line is placed in your program or a header file:

```
runc("SET BREAK @ SCRN(2,0,0,0);SCRN(0,0,0,0);");
```

Once a macro string is associated with **SET BREAK** this setting will continue in effect until changed or the session is ended. If there is any problem with the string you assign to **SET BREAK**, you will get an immediate mode (line 0) error message as soon as the program break occurs. For example, if you try to use a graphics reset before loading the **GRAPHLIB** graphics library, you will get the error indicating that the variable **SCREEN** is not found.

NOTES * **SCREEN**, the function shown in format above, is drawn from the graphics library which is supplied

with IBM-compatible versions of RUN/C. However, it should be viewed in a generic sense in that any user could create a screen function and that function could be invoked anytime a break were encountered.

* You can use any RUN/C command as an argument to **SET BREAK, SET CONTINUE** or **SET END.** For example, you could conceivably have **SET BREAK** list a particular line in a program each time a break or error occurs or even **printf** a message when a break is encountered.

* After your special application has run you may wish to disable the macro strings associates with **SET BREAK.** This can be done by executing the command are allowed as a string argument. Thus:

```
SET BREAK
```

will clear any strings assigned to these mini-menu commands.

* Escape codes such as those which control the ANSI.SYS device driver can also be generated upon **SET BREAK** since hexadecimal or octal characters are allowed as part of the environmental macro string.

SEE ALSO SET, SET BREAK, SET CONTINUE, SET END, BREAK (Section 6)

PURPOSE Allows the setting of a *macro* string which will be executed immediately after the user selects the Continue option from the RUN/C mini-menu.

FORM

```
[1] SET CONTINUE @ SCREEN(1,0,0,0);

[2] SET CONTINUE
```

USAGE **SET CONTINUE**, in conjunction with **SET BREAK** and **SET END**, allows the user to more fully control the interpreter after a break has been detected by allowing an arbitrary macro string to be associated with a Continue selection from the mini-menu. The **SET CONTINUE** macro string will be executed each time you select **Continue** from the mini-menu. Once set, the **SET CONTINUE** macro will remain in effect until replaced by another string, a null string as shown in Form [2], above or when the session is terminated.

See the discussion above on **SET BREAK** for further information and examples on these environmental commands.

If there is any problem with the string you assign to **SET CONTINUE**, you will get an immediate mode (line 0) error message as soon as you attempt to continue an interrupted program. For example, if you try to use a graphics reset before loading the **GRAPHLIB** graphics library, you will get the error indicating that the variable **SCREEN** is not found.

NOTES

* After your special application has run you may wish to disable the macro string associated with **SET CONTINUE**. This can be done by executing the command without a string argument as in:

```
SET CONTINUE
```

* Escape codes such as those which control the ANSI.SYS device driver can also be generated upon **SET BREAK** since hexadecimal or octal characters are allowed within the environmental macro string.

SEE ALSO SET, SET BREAK, SET END, BREAK (Section 6)

SET END <string>
Environmental Command

PURPOSE Allows the setting of a *macro* string which will be executed immediately after a program ends normally or after the user selects the End option from the RUN/C mini-menu.

FORM

```
[1] SET END @ SCREEN(0,0,0,0);

[2] SET END
```

USAGE SET END, in conjunction with SET BREAK and SET CONTINUE, allows the user to more fully control the interpreter after a break has been detected by allowing a previously assigned macro string to be executed when program execution ends or the End option is selected from the mini-menu. This macro string will also be executed whenever the program terminates normally. Once set, the macro will remain in effect until replaced by another string, a null string as shown in Form [2], above, or when the session is terminated.

See the discussion above on SET BREAK for further information and examples on these environmental commands.

If there is any problem with the string you assign to SET END, you will get an immediate mode (line 0) error message as soon as it is used. For example, if you try to use a graphics reset before loading the GRAPHLIB graphics library (supplied with IBM-compatible versions of the interpreter), you will get the error indicating that the variable SCREEN is not found.

NOTES

* After your special application has run you may wish to disable the macro string associated with SET END. This can be done by executing the command without a string argument as in:

```
SET END
```

* Escape codes such as those which control the ANSI.SYS device driver can also be generated upon SET END since hexadecimal or octal characters can be part of the environmental macro string.

SEE ALSO SET, SET BREAK, SET CONTINUE, BREAK (Section 6)

PURPOSE Turns on or off the ability to read or write program files containing high bit characters.

FORM
```
[1] SET EXCHAR ON
[2] SET EXCHAR OFF
```

USAGE **SET EXCHAR ON** turns on RUN/C's ability to handle the full character set contained in most MS-DOS systems including characters with decimal ASCII numbers above 127. On IBM PC compatible systems both graphics and foreign characters are available in the ASCII range of 128-255.

With **EXCHAR** set to on, program files containing *high* bit characters can be loaded into the interpreter. High bit characters will be displayed in the editor mode. In addition, when **EXCHAR** is set on high bit characters can be entered within RUN/C's editor by pressing the <Alt> key followed by the appropriate ASCII number. For example, <Alt>130 will insert an accented é at the cursor point.

If **EXCHAR** is set to OFF, all high bit characters will be "stripped" and no high bit set character can be inserted into your text. This allows RUN/C to load programs created with WordStar and other text editors which make use of high bit codes as though the program files were straight ASCII text.

Calling **SET EXCHAR** without an argument will result the message:

```
No valid SET EXCHAR argument
```

EXAMPLE See above.

SEE ALSO SET, runc(), EXEC

SET QUIET
Environmental Command

PURPOSE Turns off some of the screen messages which RUN/C routinely generates in the course of program management.

FORM

```
[1] SET QUIET ON

[2] SET QUIET OFF
```

USAGE **SET QUIET ON** silences a subset of screen messages such as reporting the number of program lines which have been loaded by the program. This command was installed primarily to facilitate creation of various types of demonstration programs and program chaining via the **EXEC** command.

Once Quiet Mode is active it will remain active throughout the session unless **SET QUIET OFF** is invoked either from the command line or, in the case of RUN/C Professional, from the **runc()** command. Calling **SET QUIET** without an argument will yield the message:

```
No valid SET QUIET argument
```

EXAMPLE

```
LOAD FTOC
Loading ftoc.c
Loading line nn

SAVE FTOC
Saving ftoc.c
Save ftoc.c (21 lines)
Ok

SET QUIET ON

LOAD FTOC
Ok

SAVE FTOC
Ok
```

SEE ALSO SET, runc(), EXEC

Files on the Distribution Disks

This section contains the names of the files distributed with the various English-language versions of RUN/C Professional.

Currently all versions of RUN/C are distributed on two diskettes. The contents of each diskette are listed below and comments about special files are made where appropriate.

Disk contents will be listed in the following order:

A. RUN/C Professional for the IBM PC and compatibles
B. RUN/C Professional for "generic" MS-DOS systems

Note: Please see the appropriate "READ.ME" file on your distribution disk for any changes which may have occurred from these listings.

A. RUN/C Professional for the IBM PC/XT/AT and Compatibles

Distribution Disk No. 1

Because of the large number of example programs supplied with RUN/C a subdirectory named EXAMPLES is included on distribution disk 1. Thus when you take the initial directory from the RUN/C distribution disk you will see something similar to:

```
EXAMPLES   RCP-READ ME  RCP      EXE
INTRPT   H NULLRUNC C   RCANSLIB C   RCIBMLIB C   RUN      BAT
```

Important: The only file you need to run RUN/C Professional is RCP.EXE.

To see the files on the EXAMPLES subdirectory type:

```
chdir examples<enter>
```

This uses DOS's CHange DIRectory command. (For more information on DOS's path and subdirectory system see the DOS manual.)

Here are the names of the files on the EXAMPLES subdirectory:

```
APUT      C   ASCII     C   ATOF      C
ATOI      C   ATOL      C   BAS_PEEK  C   BAS_POKE  C   CALLOC    C
CLOSE     C   CONTINUE  C   CREAT     C   DEFINE    C   DOSDATE   C
DOSTIME   C   EXIT      C   FCLOSE    C   FEOF      C   FERROR    C
FFLUSH    C   FGETC     C   FGETS     C   FLOAT     C   FOPEN     C
FOR       C   FPRINTF   C   FPUTC     C   FPUTS     C   FREAD     C
FREOPEN   C   FSCANF    C   FSCANF    TST FSEEK     C   FTELL     C
FTOC      C   FWRITE    C   GETC      C   GETCH     C   GETCHAR   C
GETS      C   GOTO      C   HELLO     C   HMRBI     C   INKEY     C
INT86     C   INTDOS    C   INTDOSX   C   INTRPT    H   IS_TESTS  C
LOCATE    C   LPRINTF   C   LPUTC     C   LPUTS     C   LSEEK     C
MAIN      C   MALLOC    C   MONTHS    C   MOVMEM    C   OPEN      C
OUTP      C   PEEK      C   PRINTF    C   PRINTF2   C   PUTC      C
PUTCHAR   C   PUTS      C   RAND      C   READ      C   RENAME    C
RETURN    C   REWIND    C   RUNC      C   SCANF     C   SETCOM    C
SETTRUST  C   SIZEOF    C   SPRINTF   C   SRAND     C   SSCANF    C
STATIC    C   STDERR    C   STDIN     C   STDOUT    C   STRCAT    C
STRCMP    C   STRCPY    C   STRIP     C   STRLEN    C   STRUCT    C
SWITCH    C   TEST      FIL TOLOWER   C   TOUPPER   C   TYPEDEF   C
UNGETC    C   UNGETCH   C   UNION     C   UNLINK    C   VIDEO     C
VOID      C   WINDOWS   C   WRITE     C
```

In most cases the programs with ".C" extensions such as GETC.C are the same as those given as examples in Section 4 of the RUN/C Professional manual.

In addition, the following files are included on distribution disk 1:

ASCII.C -- Helps determine what your favorite printer prints

when you ask for the ASCII character set. Turn printer on before using. Last surviving example of RUN/C manual writer Katz's legendary C coding skills.

FTOC.C -- K&R's favorite fahrenheit to centigrade program.

HELLO.C -- K&R's first and most famous program.

HMRBI.C -- C version of ancient computer game (circa 1976 A.D.).

INTRPT.H -- A header file which may be used with RUN/C's **intdos, int86, intdosx** and **int86x** functions. INTRPT.H is contained on both the \EXAMPLES subdirectory and the root directory of distribution disk No. 1.

NULLRUNC.C A function which can be compiled under a standard C compiler and then linked with other object modules to cause any calls to RUN/C's **runc()** debugging function to be negated. By linking in an object module derived from this program you will be able to move code back and forth between your compiler and RUN/C without having to remove calls to the debugging function **runc()**.

MONTHS.C -- Full (and corrected) program from the program fragment on page 3-10 of the RUN/C Pro. manual.

RCIBMLIB.C -- Source code for **cls** and **locate** functions for the IBM PC and PC-compatible MS-DOS systems. The source code for RCIBMLIB.C can be **#include**'d when compiling programs which use the RUN/C functions **locate** or **cls**. Alternatively, they can be compiled into object modules.

RUN.BAT -- Invokes RUN/C with parameter-setting command line flags. We almost named this TINY-RC.BAT, since it shrinks the run-time interpreter by about 25k bytes by lowering the maximums on program **#defines**, lines and by reducing the size of the interpreter's stack.

STRIP.C -- Author Steve Walton has contributed a version of his High-Bit Nosher which pulverizes all the high bits in WordStar(tm) files -- making the files much more readable at the DOS level and with other word processors.

VIDEO.C -- Illustrates a specialized use for **movmem** -- sunglasses are recommended. Note: This program will only work properly on IBM compatible systems.

WINDOWS.C -- Illustrates how **int86** can help you build a windowing product (well, one has to start somewhere). Note: This program will only work properly on IBM compatible systems.

Distribution disk No. 2

This disk contains several sample loadable libraries, files necessary to create Loadable Libraries for each supported compiler, and several practical libraries such as MATHLIB and GRAPHLIB. MATHLIB contains a number of transcendental math functions, listed alphabetically in Section 4 of the manual. GRAPHLIB contains functions selected from the BASIC_C Graphics library (described in detail in Appendix D of the RUN/C Professional manual).

The files contained on this root directory of disk 2 are:

Subdirectories on the disk

MATHLIB	Loadable math functions subdirectory
GRAPHLIB	Loadable graphics functions subdirectory
LAT-2	RUN/C Loadable Library system for Lattice 2
LAT-3	RUN/C Loadable Library system for Lattice 3
MSC-3	RUN/C Loadable Library system for Microsoft C version 3
MSC-4	RUN/C Loadable Library system for Microsoft C version 4

Files on the disk

READ.ME	Identifying the disk
CHECKRCL.EXE	A program to check the validity of a Loadable Library System
MAKELIB.C	A program to create MAKELIB.BAT which can be used to aid the creating of Loadable Library .MAP and .EXE files (see discussion, pp. 7-8).
DISKSTAT.C	Source code to the DISKSTAT Loadable Library
DISKSTAT.EXE	Loadable Library executable file
DISKSTAT.MAP	Loadable Library extended .MAP file
DISKSTAT.RCL	Loadable Library ASCII function description

	file
STATS.C	RUN/C program to exercise DISKSTAT functions
GLOBALS.C	Source code to the GLOBALS Loadable Library
GLOBALS.EXE	Loadable Library executable file
GLOBALS.MAP	Loadable Library extended .MAP file
GLOBALS.RCL	Loadable Library ASCII function description file
GLOBTEST.C	RUN/C program to exercise GLOBALS functions
ASSEMLIB.C	Source code to the ASSEMLIB Loadable Library
ASSEMLIB.EXE	Loadable Library executable file
ASSEMLIB.MAP	Loadable Library extended .MAP file
ASSEMLIB.RCL	Loadable Library ASCII function description file
ASMTEST.C	RUN/C program to exercise ASSEMLIB functions

The three demonstration libraries are DISKSTAT, ASSEMLIB and GLOBALS. DISKSTAT, as the name implies, provides information about the contents of a disk drive using loaded library functions which in turn make use of system interrupts.

The ASSEMLIB illustrates use of assembly language routines in loaded libraries. The assembly language program is based on a similar program described in Section 5 of the manual.

The GLOBALS loadable library illustrates the technique of modifying variables in loaded libraries.

Source code for all three libraries is included.

MATHLIB subdirectory

The files contained in the MATHLIB subdirectory are:

MATHLIB EXE	MATHLIB Loadable Library system
MATHLIB MAP	" " " "
MATHLIB RCL	" " " "

The following example programs drawn from Section 4 of the manual are contained in this subdirectory:

MATHLIB EXE	MATHLIB MAP	MATHLIB RCL					
8087TEST C	ACOS C	ASIN C	ATAN C	ATAN2 C			
CEIL C	COS C	COSH C	EXP C	FABS C			
FLOOR C	LOG C	LOG10 C	POW C	SIN C			
SINH C	SQRT C	TAN C	TANH C				

GRAPHLIB subdirectory
The files contained in the GRAPHLIB subdirectory are:

GRAPHLIB EXE	GRAPHLIB Loadable Library system
GRAPHLIB MAP	" " " "
GRAPHLIB RCL	" " " "
GRAPHICS H	Useful #defined values but not required for operation of the system.
G_BREAK C	Program to assign special strings to the BREAK (<Ctrl-c>), C(Continue) and E(End) commands.

The following example programs drawn from Appendix D of the manual are also contained on the GRAPHICS subdirectory:

GRAPHLIB EXE	GRAPHLIB MAP	GRAPHLIB RCL					
GRAPHICS H	ARC C	CIRCLE C	DRAW C	FASTFILL C			
GET_PIC C	LINE C	LOCATE C	PAINT C	PATCH C			
POINT C	PSET C						

LAT-2 subdirectory
The files contained in the LAT-2 subdirectory are:

RCA-LAT2 OBJ	One of two files required to create a Loadable Library .EXE and .MAP files using the Lattice version 2 C compiler.
RCB-LAT2 OBJ	Second of two files required to create a Loadable Library with the Lattice 2 compiler.

LAT-3 Subdirectory
The files contained in the LAT-3 subdirectory are:

RCA-LAT3 OBJ	One of two files required to create a Loadable Library .EXE and .MAP files using the Lattice version 3 C compiler.
RCB-LAT3 OBJ	Second of two files required to create a Loadable Library with the Lattice 3 compiler.

MSC-3 subdirectory
The files contained in the MSC-3 subdirectory are:

RCA-MSC3 OBJ	One of two files required to create a Loadable Library .EXE and .MAP files using the Microsoft version 3 C compiler.
RCB-MSC3 OBJ	Second of two files required to create a Loadable Library with the Microsoft 3 compiler.

MSC-4 subdirectory

The files contained in the MSC-4 subdirectory are:

RCA-MSC4 OBJ	One of two files required to create a Loadable Library .EXE and .MAP files using the Microsoft version 4 C compiler.
RCB-MSC4 OBJ	Second of two files required to create a Loadable Library with the Microsoft 4 compiler.

B. RUN/C Professional for the "Generic" MS-DOS Systems

Distribution disk #1

The contents of disk No. 1 are the same as those for IBM PC compatible systems, listed above. The RUN/C Professional .EXE file is different, of course.

Distribution disk #2

The GRAPHLIB loadable library is not provided with Generic MS-DOS versions of RUN/C Professional, since IBM-compatible graphics capabilities are not available on these systems. Also please note that example programs which take advantage of the video memory-mapping PC-compatible systems -- such as VIDEO.C -- will probably not work on generic MS-DOS systems without modification by the user.

New and Newly Documented Screen Messages

(Source code excavation has uncovered the following previously undocumented numbered error messages.)

9 Too few arguments to built-in function.

A built-in function has been called with an insufficient number of arguments.

10 Two many arguments to built-in function.

A built-in function has been called with a surplus of arguments.

11 Wrong number of arguments for a built-in function.

More than *n* arguments were given to a built-in function that does not allow a variable number of arguments; *n* equals seven in versions with built-in BASIC-type functions, otherwise the maximum number is four.

13 Subtraction of incompatible pointers.

You can subtract two pointers only if they are both pointers to exactly the same kind of object. Type must match and they must match in all details. For example, if you are subtracting pointers to arrays they must be arrays of the same things.

14 Expression syntax -- operator or semicolon missing.

This error is generated when a semi-colon or, less likely, an operator is expected. Please note that the absent semi-colon may be missing from the previous line.

14 Illegal use of 'void' in declaration

An attempt was made to declare an object using the type **void** improperly, such as pointer to **void**, array of **void**, or just plain **void**. The only proper use for **void** in declarations is: function returning **void**, pointer to function returning **void**, and the like.

286 auto or register in global or static declaration.

An attempt has been made to apply **auto** or **register** to contexts where they cannot be used; **auto** and **register** cannot be used in global situations or for **static** variables.

287 Built-in function declared with wrong type.

An inappropriate return type has been declared for a built-in function. See the RUN/C manual, Section 4, for the return types expected of built-in functions.

391 Argument and parameter counts unequal in macro call.

A macro that was defined with parameters was called without visible arguments (no immediately following left parenthesis was found on the same line) or the number of arguments given was not equal to the number of parameters in the definition.

393 Unmatched parentheses in macro call.

No closing right parenthesis for macro arguments was found (the closing right parenthesis must be on the same line with the rest of the call). RUN/C is looking for a right parenthesis that balances the left parenthesis that began the arguments list, since material enclosed in parentheses may be included among arguments. *Possible remedy:* make the left and right parenthesis count balance in the argument call.

394 Macro-expansion cycle limit exceeded -- suspect mutual definitions.

Macros have been found and expanded 100 times on the same line without resolving the logic in the expansion, i.e. macros are either too complex or circularly defined. *Remedy:* Examine carefully the logic of the macro.

Unnumbered messages

The following unnumbered messages have been added beginning with version 1.02:

Last left in line n.

In a situation where there is an unbalanced number of braces, brackets or parentheses, the last opening brace, bracket or parenthesis mark was found on the named line.

Last right in line n.

In a situation where there is an unbalanced number of braces, brackets or parentheses, the last closing brace, bracket or parenthesis mark was found on the named line.

No left found.

In a situation where there is an unbalanced number of braces, brackets or parentheses, no opening brace, bracket or parenthesis mark was found.

No right found.

In a situation where there is an unbalanced number of braces, brackets or parentheses, no closing brace, bracket or parenthesis mark was found.

Changes Between Version 1.03 and Version 1.10

1. Program loading has now been sped up by a factor of three.

2. Spurious **duplicate function** messages occurred occasionally. For example, a global declaration of two or more functions in the same statement (as in **char *f1(), *f2();**) formerly caused this message to appear. Another circumstance was in the presence of an initialized static array. We believe this problem has been corrected.

3. Performance of the RUN/C Profiler (PRON) has been improved. Formerly counts were off by 1 in some situations, null statements were sometimes not counted and a line that called another interpreted function was sometimes counted an extra time.

4. Declarations such as **char *cp={"hello"};** did not work and now do.

5. Similarly, declarations such as **char ac[]={"hello"};** did not work; now do.

6. Initialized static arrays within functions sometimes caused a variety of problems with seemingly unrelated symptoms including false **duplicate function** messages, seeming failure of character casts, an occasional Error 22 and others. This has been fixed.

7. Formerly it was impossible to increment an **unsigned** parameter within a function; this can now be done.

8. A spurious **label not found** error was produced when a **goto** followed a **switch** statement; this has been corrected. (However, please note that **gotos** must still follow the documented discipline.)

9. A problem which occurred when an attempt was made to allow 200 or more **#defines** (command line option **-d**) has been corrected; the number of **#defines** can now be set as high as 999, memory permitting.

10. **getch()** modified. The **getch()** function no longer echoes to the screen the characters it reads. We've made this change to be compatible with the Lattice and Microsoft C compilers' **getch** functions.

    Note: Both Lattice's and Microsoft's documentation state that

their **getch** functions read characters directly from the keyboard. In reality, their **getch** and ours read from the **stdin** (standard input device). If this is redirected using the '<' symbol at execution time (or in RUN/C, by the **RUN** command) then characters will be read from a file by **getch** instead of from the keyboard (see also the RUN/C manual, Section 4, on the **RUN** command).

11. Formerly, it was possible to crash the editor (and occasionally the entire interpreter) by using a **typedef** to specify a function's returned data type. This is no longer true.

12. Formerly, it was possible for the user to inadvertently close any of the standard input/output channels (**stdin, stdout, stderr**) or redirect it to other devices or files during a run. RUN/C will now reopen them at the end of a run. Note: An output channel like **stdout** or **stderr** can be closed by sending the character <Ctrl-z> (ASCII 26) to it. Subsequent attempts to send anything out to these channels after a <Ctrl-z> has been sent will be painfully fruitless.

13. Math library errors are now trapped. For instance attempting to take the square root of the a negative number will no longer hang up the interpreter; instead a zero value will be returned with an appropriate error message.

14. A number of command line arguments (described on pp. 9-11) have been added to make it possible to tailor the interpreter to specific program requirements and memory limitations.

15. When a structure was defined containing consecutive bitfield members totaling more than 16 bits the interpreter would hang; this has been corrected.

16. Loadable Library .RCL files now permit blank lines except in the first three lines of the file. Formerly, this was not allowed and, in fact, could cause serious problems if the blanks were present at the end of the .RCL file.

17. **strcat** and **strcpy** -- which formerly returned nothing -- now return a pointer to the first argument of the function; this conforms with standard C library syntax.

18. Problems related to an array **typedef** and an array of array **typedefs** have been fixed to the extent that such objects can be declared.

19. The function **poke** has been renamed **bas_poke**; the function **peek** has been renamed **bas_peek**. See new descriptions of these functions, pp. 13-23.

20. New Lattice-compatible functions for **peek** and **poke** have been added. See descriptions of these functions, pp. 13-23.

21. A number of **SET** options have been added and their documentation consolidated above, see pp. 25-30.

22. The example program PEEK.C has been renamed BAS_PEEK.C to correspond with the new name for the function. Same with POKE.C to BAS_POKE.C. The current PEEK.C reflects the new **peek** function.

    The example programs CREAT.C, FGETS.C, FTOC.C, GETC.C, WRITE.C, INTDOSX.C and G_BREAK.C have all undergone minor renovation. Please follow the programs on disk rather than their respective listings in the manual.

    The sample program FUNCTPTR.C has been removed since it illustrated a use of function pointers which was not standard C (SW apologies for getting carried away).

23. In IBM compatible versions, the \GRAPHICS subdirectory now contains some simple example graphics programs for use on IBM compatible systems.

24. Handling of the ANSI **void** data type has been improved. Improper **void** declarations such as **void v;**, **void *vp;**, and **void va[10];** were not trapped, they now are.

    Also, proper **void** declarations such as **void fv();** and **void(*pfv)();** were incorrectly rejected; they are now processed correctly.

25. The user has been given the ability to tailor the amount of memory reserved for interpreted programs and data when a function library has been loaded. This option (-i) makes it possible to vary the amount of memory between 10 and 640k bytes, although a more realistic upper limit would be in the neighborhood of 256k. See discussion on -i option, pp. 3-4.

26. The maximum number of functions which may be listed from a single loadable library (**LIBSHOW**) has been raised from 128 to 255.

27. The file MAKELIB.C has been added to Distribution Disk No. 2. This program can be used to create MAKELIB.BAT, a batch routine for creating a Loadable Library .MAP and .EXE files. MAKELIB is described on pp. 7-8.

Changes between versions 1.00 and 1.03

28. A bug (in version 1.02 only) which prevented inputting a vertical bar (|) (ASCII 124) in the editor has been fixed.

29. Handling of comments has been improved. Previously there were circumstances in which a comment could not be freely placed within the character stream of the interpreted program without problems affecting execution. (A bug related to this problem still in around, however, and is described in the next section.)

30. A bug which limited the number of global variables to less than 50 (resulting in an **Out of memory** error) has been repaired.

31. A new version of CHECKRCL.EXE (for use with Loadable Libraries) is now supplied. The previous version provided false error messages in many situations.

32. Library loading speeds were increased by a factor of four.

33. The manner of evaluation of numeric constants has been improved.

34. Handling of **unsigned** numbers improved. Formerly when an **int** was assigned to an unsigned or vice-versa we were not copying the exact bit pattern which resulted in data types being evaluated in a non-standard manner. In addition, overflow of an assignment into an **unsigned** now works as it should in that the result is the original value modulo 2 to the 16th as per Harbison & Steele (C -- A Reference Manual, Prentice-Hall, 1984).

35. RUN/C now retains the name of the file which was LOADed into memory through the LOAD command. This way you can save your program to simply by executing the SAVE command either from the RUN/C Command line or by using the **runc()** function. To determine the name of the currently logged-in file enter the screen editor -- the name of the currently file will be placed at the top of the screen. If no name appears and a SAVE is executed without a name, the file will be saved to PROGRAM.C.

36. Data type declarations of RUN/C built-in functions are now permitted. They are still not required for interpreted execution, but using such declarations appropriately can increase the likelihood of your programs compiling and linking successful on the first attempt with a standard C compiler.

    In using data declarations there are two things to keep in mind:

    1) Make sure that the declaration yields the correct type for the function (e.g., **char *malloc();**, since it returns a

character pointer) or you'll get Error No. 287: built-in function declared with wrong type;

2) Do not attempt to declare a built-in function which returns nothing (a **void**) such as **printf** and **cls**.

37. If a library function returns a pointer which is put into a local variable, the SET TRUST flag must be ON (e.g., **SET TRUST ON**). This is necessary to override RUN/C's normal pointer checking. TRUST mode should be returned to the default off condition (**SET TRUST OFF**) as soon as the library pointer operation is completed.

38. RUN/C Professional does not make direct use of teh ANSI.SYS driver even if it is installed in the system's CONFIG.SYS file. However, the user can from within RUN/C take advantage of ANSI escape sequences facilities if the device driver is properly installed. For instance if the user executes:

```
printf("\033[2J");
```

the screen will clear if ANSI.SYS is installed since <Esc>[2J is the ANSI control code for clear screen. Similarly, the user can adjust screen color and strings assigned to function keys as well as do direct cursor movement using ANSI codes. See your DOS Technical Reference Manual for more information on use of ANSI control sequences.

Note: If you have a "generic" MS-DOS system which does not support the PC's video-mapped display (IBM ROM BIOS or emulation) you will need to get the version of RUN/C Professional which uses the ANSI device driver. Contact your dealer for more information regarding whether your system can use the ANSI version of RUN/C Professional. Also see a related discussion in Section 1 of the manual.

39. As noted in Section 5 of the manual, it is necessary to eliminate or otherwise mask the **runc()** function when compiling programs which have been developed under RUN/C and which use the **runc()** function. One way to do this is to **#include** the following statement in any program module:

```
runc(s)
char *s;
{}
```

This effectively "nulls out" any call to the **runc()** function. Alternatively you can compile our RUNCNULL.C file and link it in when you compile your program using the Microsoft or compatible linker. This file is included on distribution disk 1.

40. A number of minor improvements have been made in the way **gotos** and their labels are handled to conform to standard C. If you are within some nested combination of loops and/or switch blocks, the effect of a **goto** will be that of sufficient breaks to get you out of all the loops and back to the level at which the target label exists.

    Although RUN/C may permits it, labels should not be placed in the *then* or *else* arm of an *if* or *if ... else* statement; inside the body of an *if*, *for* or *while* loop; or outside the current function. If a label is to be found, it must be in the same block as the corresponding **goto** or within an enclosing block and certainly within the same function.

41. It has been called to our attention that confusion can arise regarding the use of uninitialized variables in RUN/C. Unlike the BASIC interpreter, RUN/C conforms to standard C by not initializing local automatic variables to zero (or any other value) when the program is run. In exception to the above rule, RUN/C initializes local variables which are of type pointer to zero as a preventative measure in catching mistaken use of uninitialized pointers. To reiterate: since all types of variables (other than pointer) which are declared local to a function are NOT initialized a declaration such as:

```
int a;
printf("%d", a);
```

    may print out a 0 or any other decimal value which happens to exist at the point in memory assigned to hold the value of a. Therefore it is important that variables never be used until a value is assigned to them.

Known Bugs and Other Anomalies

1. Preprocessor commands other than **#include** and **#define** are not currently supported.

2. Aggregates (structures and unions) that are defined within **#include**'d files cannot be used for global or static declarations, even though the scope of these definitions should make them available everywhere past the **#include** line. Automatic declarations are not affected. To work around this bug, please put aggregated definitions that will be needed for global and/or static declarations within your primary source file.

3. Initialized **typedef** arrays such as:

```
typedef char fname[20];
fname fileset[3]={"pgm1.c","pgm2.c",pgm3.c"};
```

   don't work; the interpreter cannot yet grasp the correlation between overall array dimensions -- [3][20] here -- and the "shape" of the initializer.

4. A **typedef** cannot be used to define a function pointer type, such as **typedef int (*fred)()**.

5. An intermittent problem related to memory being corrupted when **floats** are passed as arguments to a function whose parameters are declared **float** and/or which returns a **float** has been noted. However, symptoms of this problem temporarily disappeared; we would appreciate detailed reports of any reproducible problem(s) of this sort.

6. An **#include** directive remains active even when "removed" by means of a block comment, e.g.:

```
/*
#include "myhdr.h"
*/
```

   however the statement:

```
/* #include "myhdr.h" */
```

   will be properly treated as a comment.

7. Some programs cause a small, permanent (for the duration of the RUN/C session) memory loss each time they are executed. Amounts range from 16 to 56 bytes at a time; a global use of

typedefs has been implicated.

8. Built-in functions cannot currently be redefined.

9. It is not yet possible for user-defined interpreted functions (as opposed to loadable library functions) to take a variable number of arguments.

10. RUN/C cannot at present deal with the declaration of an array of function pointers as a single statement, for example:

```
int (*afp[10])();
```

for example. Such an object can, however, be created by means of a typedef like this:

```
typedef int (*IFP)();
  /* IFP: type: pointer to function returning int */
IFP aifp[10];
  /* aifp: array of pointers to functions returning int
*/
```

Similarly, pointers to arrays for example,

```
int (*pai)[10];
```

do not work, but if you use:

```
typedef int AI[10];
AI *pai;
```

you'll get the object of your desires.

11. It is sometimes not possible to assign a decimal number without a leading digit. Thus:

```
float a = .9;
double b = .8;
```

will not be accepted while:

```
float a = 0.9;
```

would be.

12. The equality testing on doubles will sometimes fail. For example in:

```
double A = 4.9;
double B = 5.0;
```

if you add 0.1 to A the result will print out as five but a test to see if they are equal:

```
if(a == b)
```

will indicate that they are not equal under some circumstances.

13. Nested **?** **:** operations are not currently working properly. However, appropriate use of parentheses (which actually make these expressions much easier to read) will permit nested **?** **:** to work properly.

14. Buffered files (those opened with **fopen**), are by default read or written using "newline" translation. This means that when the file is read, Carriage return/linefeed character pairs are collapsed to linefeed characters (the '\n' char). Also, if a <Ctrl-z> (Decimal 26) is encountered this will signal the end-of-file. Such translation will occur whenever a file is opened using mode: **"r"**, **"w"** or **"a"** (e.g. **fopen(filename,"r")**). Translation is turned off (binary mode) whenever a file is opened in mode: **"rb"**, **"wb"**, or **"ab"**.

Currently **ftell** returns an incorrect file position when the file has been opened in "translation" mode. **ftell** will return correct values for a file opened using "binary" mode.

INDEX

#define 4-22, 4-202, 4-367, 5-24, 6-5, 7-5, 7-7, 7-11
#include 4-30, 4-185, 5-24, 7-7, 7-11
%% 4-261, 4-296
%c 4-261, 4-296
%d 4-261, 4-296
%e 4-261, 4-296
%f 4-261, 4-296
%g 4-261, 4-296
%o 4-261, 4-296
%s 4-261, 4-296
%u 4-261, 4-296
%x 4-261, 4-296
& operator 4-184
.C extension 4-214
/m flag 5-18
80287 1-26
8086/8088 machine registers 4-183, 5-30
8087 1-26
<Backspace> 2-11
<Ctrl-a> 2-9
<Ctrl-Break> 1-7
<Ctrl-c> 1-7, 2-10, 4-89, 4-173, 4-210, 4-238, 4-244, 4-245, 4-306, 4-387, 5-2, 6-6
<Ctrl-d> 2-9
<Ctrl-f> 2-9
<Ctrl-g> 2-11
<Ctrl-h> 2-11
<Ctrl-i> 2-16
<Ctrl-l> 2-15
<Ctrl-m> 2-16
<Ctrl-n> 2-15
<Ctrl-q><Del> 2-15
<Ctrl-qA> 2-14
<Ctrl-qC> 2-10
<Ctrl-qD> 2-8, 2-9
<Ctrl-qF> 2-12
<Ctrl-qR> 2-10
<Ctrl-qS> 2-8, 2-9
<Ctrl-qY> 2-15
<Ctrl-r> 2-10
<Ctrl-s> 2-9, 4-75, 4-210, 4-244

<Ctrl-t> 2-11
<Ctrl-u> 2-15
<Ctrl-v> 1-17, 2-12
<Ctrl-y> 2-12
<Ctrl-z> 4-130, 4-133, 4-164, 4-178, 7-14
<Del> 2-11
<down arrow> 2-8
<End> 2-9
<enter> 2-16, 4-167
<Esc> 2-15
<F10> 2-15
<Home> 2-9
<Ins> 1-17, 2-12
<left arrow> 2-9
<PgDn> 2-10
<right arrow> 2-9
<tab> 2-16
<up arrow> 2-8
Abort 1-26
Absolute memory addressing 4-191
Access mode 4-128
acos 4-34
Address-of operator (&) 3-7
aget 4-35
Aggregates 4-172, A-8, 7-7, 7-8
allmem 5-20
ANSI.SYS 1-8, 1-17, 2-8, 4-68
APL 3-9
Arc cosine 4-34
Arc sine 4-38
Arc tangent 4-39, 4-40
Arguments to **main()** 4-234
Arithmetic operators 3-1
Array 5-14, 7-8
ASCII text 2-24, 4-41, 4-43, 4-45, 4-341, 7-4, 7-7, 7-14
Assembly language 5-29
Assignment operators 3-2
Associativity 3-10
Asynchronous communications 4-35, 4-36, 4-313
atan 4-39
atan2 4-40
atof 4-41
atoi 1-22
auto 4-47, 4-329, 7-14
Autolines 4-92, 4-209, 7-2
Automatic variables 4-47, 4-75, 4-172
AUTOSAVE 2-15, 2-24, 4-48, 4-95, 4-353

AUX: 1-21, 4-337
Auxiliary Port 1-21
Aztec C 1-9
Back-up 1-13
Backspace 4-267
Base 10 logarithm 4-221
BASIC 1-1, 1-9, 2-3, 2-8, 4-169, 7-1, 7-2
BDOS 4-186
Binary bit pattern 4-310
Binary data 3-1, 4-130, 4-311
Binary mode 4-130, 4-150
Binary operators 3-1
BIOS 4-186, 4-195
Bitwise operators 3-5, 4-184
Bitfield structure A-8
Block 4-172, 7-9
Boot disk 1-14
Breakpoint 4-50, 4-81, 4-134, 4-169, 4-173, 4-350, 6-1, 6-6
Buffered I/O 1-23, 5-21
Built-in functions 4-124
C compiler 1-9
C-Food Smorgasbord 4-35, 4-83, 4-225, 4-227, 4-229, 4-313
C identifier 4-202
C preprocessor 4-26
C Puzzle Book 4-28
C--A Reference Manual 4-28
C86 7-12
calloc 4-55, 4-147
Capital letters for commands 2-2
Carriage return 4-267
Carry flag 4-183
Case 4-350
Cast 4-56
Cast operator 4-240
ceil 4-60
char 4-61, 4-164, 7-5, 7-11
Character position pointer 4-281
Character stream (primitive) I/O 1-23
Character- and string-related functions 4-2, 4-11
CHECKRCL 5-8, 5-16, 5-18, 5-33
CHKDSK.COM 1-8
CLIST 4-63, 4-172, 6-3
Clones 1-8
close 4-247
clrerr 4-66, 4-113, 4-141
cls 1-10, 4-68
Collection 4-347

Collections ..... See Aggregates
COM 1-21, 4-93, 4-337
Comma Operator (,) 3-7
Command level 2-1
Command line 6-2
Command line arguments 4-233
Command line options 1-18
Command mode 5-1
COMMAND.COM 2-21, 2-21, 2-22
Communications port 4-36, 4-310
Compiler compatibility 1-9
Computer Innovation's C86 1-9
CON: 1-21, 4-337
Conditional expression operators 3-8
CONFIG.SYS 1-16, 4-334
Console 4-333
continue 4-51, 4-53, 4-69, 4-81, 4-169
Control codes 1-10
Control keys ..... 2-8, See also <Ctrl-
Control-c ..... 1-7, See also <Ctrl-c above>
Conversion character format 4-296
Conversion characters 4-260
cos 4-70
cosh 4-71
Cosine 4-70
CPU flags 4-183
CPU returned value 4-196
CPU value 4-183
creat 4-72, 4-156, 4-231, 4-281, 4-388
Ctrl-<left arrow> 2-9
Ctrl-<PgDn> 2-10
Ctrl-<PgUp> 2-10
Ctrl-<right arrow> 2-9
ctype.h 4-30, 4-202
Cursor control keys 2-8
Data matching 4-153, 4-300
Data structure 4-345
Data types 4-2, 4-8, 4-10
Decimal floating point 4-296
Decimal 4-261
Default drive 1-23
Default extension 2-2
Default 4-59, 4-78, 4-350
DEL 4-205
DELETE 7-2
Deleting characters 2-7
DeSmet C88 1-9

Device driver 1-17
Digital Research C 1-9
DIR 4-79, 6-3
Disk error 4-109
Disk full conditions 4-66
Disk primitive functions 4-156
DISKCOPY.COM 1-13
DISKSTAT 5-9
Distribution disk 1-9, 1-12
Divide-equals (/=) 3-3
do ... while 4-81, 4-170
do 4-50, 4-69
Document mode 2-24
Documentation conventions 1-10
DOS 4-186, 4-233, 7-5
DOS 3.1 1-8
DOS command line 4-103
DOS device names 4-336
DOS disk 1-16
DOS interrupts 4-183
DOS path 4-96, 5-10
dosdate 4-82
dos.h 4-188
dostime 4-85
Double precision 4-34, 4-41
double 4-88, 4-125, 4-266
Drive name 1-23
Drive/directory 1-2, 1-16, 1-18, 1-25, 4-213
Dummy program 5-18
Dummy values 4-218
Dump 4-75
E 4-41, 4-41
EC.EXE 2-23
EDIT ERROR 4-91
Edit level 2-1
EDIT 4-91, 4-95, 4-172, 7-2, 7-7
EDITOR 2-1, 2-21, 4-93, 4-214
else 4-98
else if 4-58, 4-100
End-of-file 4-117, 4-72
EOF 4-28, 4-31, 4-164, 4-172, 7-14
Equals sign (=) 3-2
Equals 3-4
Escape sequences 1-10, 4-266
EXAMPLE 4-21
EXE file 4-93, 5-18
EXEC 4-102, 6-4

exit 4-104, 4-110, 4-134, 4-169, 4-184, 4-314, 7-14
exp 4-106
Exponent field 4-263
Exponentation 4-41
Exponential floating point 4-261
Exponential value 4-106
Extension 1-23
extern 4-107, 7-13, 7-14
Externals 5-2, 5-25
fabs 4-108
fclose 4-109
feof 4-111
ferror 4-113
Feuer, Alan 4-28
fflush 4-115
fgetc 4-117, 4-162, 4-165, 4-288
fgets 4-119, 4-167
Field width 4-265
File Allocation Table 4-307
File errors 4-66
File functions 4-2
File I/O 5-20
File names 1-23
File number 4-64, 4-247
File pointer 1-23, 4-231
File position pointer 4-72, 4-231, 4-372
FILE 1-23, 4-30, 4-121, 4-172
FILES 4-79, 4-122
Filespec 1-23
Fixed decimal floating point 4-261
FLIST 1-22, 4-124, 4-172, 6-3, 7-11
float 4-125, 4-348, 4-375
floor 4-127
Flushing buffer 4-109
Flushing 4-104
fopen 4-104, 4-109, 4-115, 4-128, 4-137,
..... 4-139, 4-150, 4-162, 4-247, 5-3, 5-21
for 3-8, 4-50, 4-69, 4-132, 4-170
Formal parameter A-11
Formfeed 4-267
FORM 4-21
fp 4-109, 4-115
fprintf 4-136, 4-259
fputc 4-139, 4-274, 4-288
fputs 4-141
FRE 2-22, 4-143, 6-3
fread 4-281

free 4-147
freopen 4-149
fscanf 4-152, 4-295
fseek 4-155, 4-231
ftell 4-157
FTOC.C 1-1, 4-290
Function keys 2-8
getc 4-117, 4-162, 4-165, 4-335
getch 4-164
getchar 4-117, 4-133, 4-162, 4-162, 4-164, 4-165, 4-334
getmem 5-20
getml 5-20
gets 4-167, 7-6
Global and static variables 4-75
Global variables 4-107, 5-2, 5-19, 5-25, 7-6
goto 4-81, 4-134, 4-169, 4-172, 7-5, 7-7
Greater than or equal to 3-4
Greater than 3-4
Harbison & Steele 4-28, 4-47
Hard disk 1-14, 1-14
Header file 4-202
Heap space 5-20, 5-21
HELP 4-172, 6-3
Hexadecimal 4-41, 4-200, 4-261, 4-296
High bit 2-24, 4-95
Home 2-8
Horizontal tab 4-267
Hyperbolic cosine 4-71
Hyperbolic sine 4-319
Hyperbolic tangent 4-356
I/O 1-23
I/O addresses 4-250
I/O functions 4-2, 4-13
I/O library 5-3
I/O port 4-180
IBM C 1-9
IBM 1-8, 1-17, 7-12
Identical strings 4-341
Identifier 4-22, 4-26
if-else 3-8, 4-98
if 4-69, 4-98, 4-100, 4-176
Ignore 1-26
Illegal operands combination 4-57
Illegal termination of expression 1-12
Immediate mode 1-21, 5-1, 5-13, 5-17, 6-1
Increment and decrement operators 3-4
Indirection operator 3-6

inkey 4-177, 7-14
inp 4-180
Input stream 4-372
Input/output 1-23
Insert mode 1-17, 2-16, 7-2
Installing RUN/C 1-12
int 4-181, 4-222, 4-317, 4-348, 4-375, 7-5, 7-11, 7-13
int86 4-195
int86x 4-198
intdos 4-83, 4-182
intdosx 4-190
Integer conversion 4-43
Integer number 4-43
Intel 4-191
Interactive mode 4-51, 4-173, 4-306, 4-353, 6-1
Interrupts 4-184, 4-187
intrpt.h 4-184
Invalid stack size 1-20
Inverse cosine 4-34
Inverse sine 4-38
Inverse tangent 4-40
isalnum(c) 4-201
isalpha(c) 4-200
isascii(c) 4-202
iscntrl(c) 4-201
iscsym(c) 4-202
iscsymf(c) 4-202
isdigit(c) 4-200
isgraph(c) 4-201
isodigit(c) 4-200
isprint(c) 4-201
ispunct(c) 4-201
isspace(c) 4-201
isupper(c) 4-200
isxdigit(c) 4-200
K&R 4-371
Keywords 1-22, 4-2
KILL 4-205, 4-378, 6-3
Label 4-169, 7-5
Lattice C 1-9, 4-188, 5-2, 5-16, 5-18, 5-29
Ldump 4-206
Left justified 4-265
Less than or equal to 3-4
Less than 3-4
Level 1 (unbuffered) I/O 1-23, 4-156, 4-247,
Level 2 buffered I/O . 1-23, 4-109, 4-121, 4-155, 4-162, 4-247, 4-335
LIB 4-208, 5-4

LIBDIR 5-5
LIBLOAD 5-7
LIBSHOW 5-9
LIBUNLOAD 5-12
LIMITS 4-21
Line numbers 2-8
LINK.EXE 5-1, 5-16, 5-24
Linker 5-34
Linker map 5-18
List device 4-224, 4-227, 4-229, 4-245
LIST 2-5, 4-209, 4-271, 6-3, 7-3, 7-9
LLIST 1-25, 4-211, 4-271, 6-3
LOAD 1-1, 4-102, 4-205, 4-213, 7-9
Loadable Library 5-1, 6-3
locate 1-10, 4-216
log 4-220
Logged-on directory 1-25
Logical AND (&&) 3-4
Logical OR (||) 3-4
long float 4-126, 4-223, A-2
long int 4-45, 4-223, 4-266
long 4-155, 4-222, 4-348, 4-375, 7-13
lprintf 1-25, 1-26, 4-224, 4-259
LPT1: 1-21, 1-25, 4-337
LPT2: 4-337
LPT3: 4-337
lputc 4-227
lputs 4-229
lseek 4-231
Lshift-equals 3-3
LST: 1-21
Lvalue 3-2
Macro substitution 4-22
Macros 4-28, 4-202, 4-363, 7-13
main() 4-233, 5-16, 5-18, 6-5
malloc() 4-31, 4-147, 4-238, 5-3, 5-20, 7-13
Manifest constants 4-28
Manx Software's Aztec C86 1-9
MAP file 5-1
Mark Williams C (Let's C) 1-9
Math chip support 1-26
Math functions 4-2, 4-18
math.h 4-30
Members 4-347
Memory addresses 4-254
Memory mapped video 4-240
Memory requirements 2-21

Memory segment registers 4-185, 4-190
Memory space requirements 5-21
Memory-related functions 4-2, 4-15
MERGE 2-19, 7-2
Microsoft C 1-9
Microsoft 1-17, 7-12
Mini-menu command 4-51, 4-75, 4-89, 4-206, 4-353
Minus-equals 3-3
Miscellaneous 4-2, 4-20
MODE 4-312
Modulus operator 3-1
Modulus-equals 3-3
movmem 4-240
MS-DOS 1-8, 1-8, 1-12, 1-16, 4-186, 4-312, 4-378
Multiple break points 6-6
Native mode 1-17
NEW 4-243
Newline 4-41, 4-119, 4-165, 4-167, 4-267, 4-278
NLIST 4-244, 4-271
NLLIST 4-245, 4-271
Non-ASCII key 4-178
Not equal 3-4
NOTES 4-21
NUL: 1-21, 4-337
NULL 4-28, 4-31, 4-167, 4-172, 7-6, 7-13
Numbered or unnumbered error message 1-11
Object library file 4-35
Octal bit pattern 4-267
Octal 4-41, 4-200, 4-261, 4-296, 7-4
Offset address 4-155, 4-191, 4-231
Ok prompt 1-3, 2-1
open 1-23, 4-64, 4-72, 4-156, 4-231, 4-247, 4-281, 4-388
Opening curly brace 1-11
Operand 3-5
Operating system activities 4-314
Operator precedence 3-10
Operators 3-1
outp 4-250
Output device 4-277
Output file buffers 4-104
Overlay files 2-23, 4-96
Overwrite mode 4-354
Parallel printer port 4-336
Parameter list 4-2
Parentheses 3-2, 3-10
Parity 4-310
Pascal 4-26

Pathname 1-24, 4-79, 4-122
PC DOS 1-8, 1-17, 4-186
PC Jr 1-8
pcasm 4-313
peek 4-252
PgUp 2-10
PMATE 2-3
Pointer a value A-6
Pointers 4-57, 4-154, 4-167, 4-307, 4-328, 5-2, 5-30, 7-3, 7-6, 7-8, 7-13
poke 4-254
Port 4-313
Port numbers 4-250
Portability issues 6-5
pow 4-257
Pre-run checking 1-11
PREC 4-258
Precedence 3-2, 3-11
Precision field 4-263, 4-264, 4-299
Predefined symbols 4-172
Preprocessor 4-202
Preprocessor commands 5-24
Preprocessor directives 4-22
Primary name 1-23
Primitive I/O 4-247
Printer support 1-21, 1-26, 4-245, 6-4
printf 1-11, 4-222, 4-259, 4-295, 4-295, 5-3, 5-22, 7-4
PRN: 1-21, 4-337
Processor status flag 4-183
PROFF 4-270, 6-4
Profiler 4-270
Program lines 4-172
Program size 5-35
PROGRAM.C 2-24, 2-26, 4-48, 4-93, 4-214, 4-293, 4-353
PRON 4-271, 6-1, 6-4
Pseudo-random number generator 4-325
PUN: 1-21
Punctuation character 4-201
PURPOSE 4-21
putc 4-139, 4-274, 4-337
putch 4-276
putchar 4-165, 4-276, 4-277, 4-337, 4-388
puts 4-267, 4-278, 7-4
Quotation marks 1-24, 4-79, 4-122
r and r+ 4-128
Radian value 4-38
RAM disk 1-23

RAM 2-21
rand 4-280
Random garbage 4-56
RCANSLIB.C 1-10, 4-68, 4-218
RCIBMLIB.C 1-10, 4-68, 4-218
RCL file 5-1, 5-19
RCLMAIN.OBJ 5-18
RCP-READ.ME 1-9
RDR: 1-21
Read only 4-73, 4-73
Read 4-281, 4-288
Read/Write 4-73
Real number 4-257
Real time 5-3
Reboot 4-387, 5-2, 5-21, 7-10
Recursion 1-20
Redirected I/O 1-21, 4-103
Register 4-283, 7-14
Registers 4-186
Relational and Logical Operators 3-3
REN 1-11, 2-23, 4-93
Rename 4-284
Reserved words 1-22
Retry 1-26
Return type 5-22
return 4-134, 4-169, 4-286
RETURNS 4-21
rewind 4-288
Right justified 4-265
ROM BIOS 4-195
Rshift-equals 3-3
Rules of precedence 4-24
RUN 1-7, 1-11, 4-102, 4-238, 4-290, 4-292
RUN/C commands 4-3
runc() 4-51, 4-102, 4-173, 4-306, 5-1, 5-5, 5-8, 6-1
Rvalue 3-2
SAVE 4-95, 4-205, 4-213, 4-293, 6-4, 7-3, 7-9
Scalars 5-25
scanf 4-152, 4-222, 4-295
Scientific notation 4-41, 4-263
Screen 4-216
Screen cursor 2-5
Screen editor 2-1, 4-95
Secondary name 1-23
SEE ALSO 4-21
Seed value 4-325
Segment address 4-191

SET 4-303, 6-4
SET EDITOR 2-22, 2-23, 4-93, 4-304
SET STEP 4-306, 6-1, 6-4
SET TRUST 4-168, 4-307
setcom 1-25, 4-310
SHELL 1-25, 2-1, 2-21, 4-86, 4-95, 4-214, 4-314, 6-3
short int A-2
short 4-317
Sieve of Eratosthenes 5-3
Signed decimal 4-296
sin 4-318
Sine 4-318
Single Disk System 1-13
sinh 4-319
sizeof 4-55, 4-320, 4-374
Software interrupt 4-196
sprintf 4-259, 4-321, 5-23
sqrt 4-324
Square root 4-324
srand 4-325
sscanf 4-152, 4-295, 4-326
STACK OVERFLOW ERROR 1-20
Stack size 1-19
static 4-47, 4-329, 5-19, 7-9
stderr 1-25, 4-172, 4-331
stdin 1-25, 4-165, 4-172, 4-333
stdio.h 4-30, 4-337, 6-5
stdout 1-25, 4-277, 4-172, 4-278, 4-336
Stop bits 4-310
strcat 4-339
strcmp 4-341
strcpy 4-343, 5-17
String 4-261
strlen 4-344
struct 4-345, 5-29, 7-7
Structure member operator (.) 3-8
Structure pointer operator (->) 3-9
Subdirectory 1-25
switch 4-50, 4-58, 4-78, 4-170, 4-350
Syntax error 1-11
SYSTEM 4-353
TAB 4-354, 6-4
Tag name 4-347, 7-7, A-7
tan 4-355
Tangent 4-355
tanh 4-356
Template 4-347

Text file translation 4-73, 4-247, 4-282, 4-388
Times-equals (*=) 3-3
tolower 4-357
toupper 4-358
TRACE 4-306, 4-359, 6-1, 6-4
Translated mode 4-129, 4-130
TROFF 4-362, 6-4
TRON 4-306, 4-363, 6-1, 6-4, 7-3, 7-7, 7-9
TRUST flag 4-168, 4-307, 6-4
TYPE 4-365
typedef 4-366, 7-8
Unary Minus 3-2
Unary negative 3-4, 3-4
ungetc 4-371
ungetch 4-373
union 3-8, 4-374, 5-29, 7-7
UNIX 7-11
UNLINK 1-22, 4-205, 4-378, 4-379, 6-3
Unsigned decimal 4-261, 4-296
unsigned int 4-381
unsigned 4-381, 7-13
Untranslated mode 4-130
USAGE 4-21
VAL 4-382, 6-6
VAR 5-22, 5-23
VER 1-9, 4-385
Video RAM 4-240
void 5-14, 5-22, 7-14
w and **w+** 4-128
while 4-50, 4-69, 4-81, 4-134, 4-387
whitespace 4-41, 4-234, 4-264, 4-300, 7-5
Wildcards 4-79, 4-122
WINDOWS.C 4-197
Wizard C 1-9
WordStar 2-3, 2-6, 2-8, 2-23, 2-26, 4-94
write 4-73, 4-288, 4-388, 4-388
Write fault error writing device PRN 1-26
Write mode 4-109
Write-protect tab 1-12
Zero character 4-267
Zeros out 4-56
\0 4-267
\b 4-267
\f 4-267
\n 4-267
\r 4-267
\t 4-267