Q/C User's Manual

Jim Colvin

Version 3.2

THE CODE WORKS

Developed by:
Jim Colvin
Quality Computer Systems
3394 E. Stiles Avenue
Camarillo, CA 93010
U.S.A.

Published by:
The Code Works
Box 6905
Santa Barbara, CA 93160
U.S.A.
805/683-1585

Q/C User's Guide

Disclaimer

Quality Computer Systems and The Code Works make no representations or warranties, either expressed or implied, with respect to the adequacy of this documentation or the programs which it describes in regard to merchantibility or fitness for any particular purpose or with respect to its adequacy to produce any particular result. The computer programs and documentation are sold "as is," and the entire risk as to quality and performance is with the buyer. Should the computer programs or documentation prove defective, the buyer (and not Quality Computer Systems, The Code Works, its distributors or retailers) assumes the entire cost of all repair or correction and any incidental or consequential damages. In no event shall Quality Computer Systems or The Code Works be liable for special, direct, indirect or consequential damages resulting from any defect in the programs, documentation or software. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitations and exclusions may not apply to you.

9 8 7 6 5 4 3 2 1

Table of Contents

Introduction

Q/C is a compiler for the C programming language operating under the CP/M-80 operating system. The current version supports all standard C features with the exception of **float** and **long** data types, parameterized **#define**, declarations in compound statements, and bit fields.

Since portability is one of the most important reasons for writing programs in C, great care has been taken to make Q/C and its function library compatible with the UNIX Version 7 C compiler from Bell Laboratories. The Q/C function library contains over 80 I/O and utility functions. Both the compiler and the library are written in C. The source for a few functions in the compiler and the library are provided in C and in hand-coded assembly language for speed. In both cases, you can compile the C version by simply defining the symbolic constant PORTABLE as shown in the comments at the beginning of the files.

The output from Q/C is 8080 or Z80 assembler code. The 8080 code can be assembled with either the Digital Research RMAC assembler or the Microsoft MACRO-80 (M80) assembler. The Z80 code uses Zilog mnemonics and must be assembled with the Microsoft M80 assembler or with The Code Works CWA assembler. A good deal of optimizing is being done within the limitations of a one-pass compiler.

Q/C is also a compiler learning tool. If you want to learn more about compilers, Q/C provides you with the source code to a working compiler and an explanation of the major parts of the compiler. You can study the source code, experiment with it to improve efficiency, and expand it to make the compiler more powerful.

This manual does not attempt to teach you C. There are several good books available for learning C. All C programmers should have a copy of The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie (Prentice-Hall, 1978). It is difficult, but it is the authoritative reference for the language. It teaches good programming style, and contains a wealth of well-written, usable C programs. Throughout the manual I will refer to this book as Kernighan & Ritchie.

A book which is easier to read is C Programming Guide by Jack Purdum (QUE Corporation, 1983). Unless you are an experienced programmer, you will probably want to start with a book like this.

Another fascinating book is The C Puzzle Book by Alan R. Feuer (Prentice-Hall, 1982). When I got my copy, I kept reading it for hours. It explores more ways than you can imagine of writing C and figuring out what the results will be. Every puzzle has a solution and an explanation of why the results were produced. All of these books are available from The Code Works.

I also recommend that you get a copy of Software Tools by Brian W. Kernighan and P. J. Plauger (Addison-Wesley, 1976). This book provides so many well-designed and useful programs you will get new ideas and inspiration each time you open it. The programs are written in RATFOR (RATional FORTRAN), but RATFOR is patterned so closely after C that the conversion effort is minimal. The text formatter used to print this manual started out as a translation of the format program presented in Software Tools.

Finally there is the problem of bugs. Although the Q/C compiler has been tested extensively, a program this size is bound to have errors lurking somewhere. Unlike most software you buy, you have the source code so you may be able to correct the problem by recompiling the compiler. Whether you correct it or not, please report any bugs to me. I can usually be reached evenings (between 7 and 11 P.M. Pacific time) and weekends at (805) 482-3935. If you want to write, my address is:

Quality Computer Systems
3394 E. Stiles Ave.
Camarillo, CA 93010

Please include a sample of the C code that produces the error and any information about your system which you feel might be pertinent.

If you have any comments or suggestions for improving Q/C or the Q/C User's Manual, please write to me at the address above. In any case, I hope you enjoy working with Q/C as much as I have.

Jim Colvin

Acknowledgements

I would like to thank the following people for their contributions to Q/C. Glen Fisher of The Code Works developed the type handling routines used in Q/C Version 3. Also, the assembly language version of the library function makfcb is modeled on one he wrote. Randy Gilleland helped with the first Z80 version of Q/C. Kirk Bailey wrote the port I/O functions and developed improved run-time library routines for the Z80 version which do relation tests, multiplication and division significantly faster. Lyle Bickley contributed the full printf for this release of Q/C.

Special thanks go to Ron Jeffries of The Code Works who has contributed to Q/C in so many ways. His ideas and encouragement really keep me going. In addition, he does much of the work that makes Q/C a polished, professional product. Without his support you would not be reading these words.

Style Conventions Used in This Manual

When I show the general form of a command (or some part of a command such as a file name) I use the convention

KEYWORD required [optional] ...

where	means
KEYWORD	a keyword which must be typed as shown (like CC to run the compiler). These can be typed in upper or lower case.
required	a required input specified by you (like a file name)
[optional]	an optional input which you can specify (such as options for the compiler)
...	the preceding item can be repeated as many times as necessary

When I show an example, the part that you type will be underlined like this

A>CC HELLO

meaning that you type **CC HELLO** after the CP/M prompt **A>**.

Finally, in the body of the text I use the following conventions:

convention	use
boldface	used for C keywords (e.g. **char**). Also used to refer to parts of an example which is being explained (just as I used it above in referring to the parts of the **A>CC HELLO** example.
^C	used to indicate a control character. For example, ^C means the character generated by holding down the CONTROL key (may be called CNTL or CTRL on various terminals) while pressing the C key. These are the ASCII control characters having values in the range '\0' through '\37'.

1

Getting Started ... FAST!

If you're like me, when you get a new piece of software you like to see it do something immediately. I know you don't feel like wading through a lot of reading. But, if you stick with me through the first few sections of this chapter, you'll know you've got a working C compiler.

1.1 Q/C System Requirements

Q/C Version 3 is designed to run on a 56K CP/M 2.2 system. Your CP/M Transient Program Area (TPA) should be 50K or larger (see Section 1.5 for a way to measure this). You need one disk with approximately 200K capacity, but two are recommended. If you want to compile Q/C itself, you must have two disk drives. Q/C comes in two versions -- one compiles your C program into 8080 assembler code, the other generates Z80 assembler code using Zilog mnemonics. If you use the 8080 version you will need either the Digital Research RMAC assembler or the Microsoft MACRO-80 (M80) assembler. For the Z80 version you must have the Microsoft M80 assembler or The Code Works CWA Z80 assembler.

Q/C should run on CP/M-compatible systems, but this is not guaranteed. If you have problems, see Appendix F "Q/C on CP/M-Compatible Systems."

1.2 Backup the Q/C Disk

I strongly recommend that you make at least one backup copy of the Q/C distribution disk before you do anything else! File the original as your "archive", and always use a copy. (Please remember to put the copyright notice "Copyright (c) 1984 Quality Computer Systems" on the label of all your copies of my compiler.)

You can use your normal procedure for copying an entire disk. If your version of CP/M does not have a special utility to copy disks, you can use the standard CP/M utility PIP. Put a CP/M disk with PIP in drive A and a blank formatted disk in drive B. Then type (underlined part only):

```
A>PIP
```

PIP will prompt you with an asterisk (*) for the files to be copied. Now remove the CP/M disk from drive A and replace it with your Q/C disk. Type

```
*B:=*.*[OV]
```

to copy all the files from the Q/C disk to the blank disk. The O option in [OV] is necessary to copy CRUNLIB.REL. This is the relocatable version of the Q/C standard library. Since it is an object file, it may contain the CP/M end-of-file character ^Z anywhere in the file.

From now on, whenever I refer to the Q/C disk, use your personal copy.

1.3 Set Up Your Working Q/C Disk

The next thing you need to do is copy just the files you'll normally use onto a "working" disk. It is most convenient to have the compiler, the files it always needs, and your assembly tools on one disk. I suggest that you set up your working disk as follows:

1. Start with a blank, formatted disk. Put a copy of the CP/M system on it so it is "bootable." This is normally done with the utility program SYSGEN supplied with your CP/M system. See your CP/M documentation for the correct program and instructions for using it.

2. Copy the Q/C compiler (CC.COM) and the standard header file (QSTDIO.H) to your working disk from your personal copy.

3. Copy the source file HELLO.C for the test session in the next section.

4. Copy the relocatable version of the library (CRUNLIB.REL). Don't forget to use the "O" (object) flag when you PIP this file.

5a. If you use RMAC, copy RMAC.COM and LINK.COM from your copy of those tools. Note: these programs are <u>not</u> part of the Q/C distribution disk.

5b. If you use M80, copy M80.COM and L80.COM to your working disk. Note: these programs are <u>not</u> part of the Q/C distribution disk.

5c. If you use The Code Works CWA assembler, copy CWA.COM and CWLNK.COM to your working disk. Note: these programs are <u>not</u> part of the Q/C distribution disk.

You will probably want to have several other tools on your working disk, such as STAT, PIP and your favorite text editor. But for now, you are ready to try a quick sample session and see if everything works.

1.4 Compile and Run a Program

This section shows three sample sessions, one for the RMAC assembler, one for M80, and the other for CWA. Your working disk must be in drive A. In each session, you type the underlined parts. The messages which are printed on the screen will look similar to the sample sessions.

If you can't get Q/C to run, it may mean that you don't have enough memory. Run QRESET described in Section 1.5 "Using QRESET to Customize Q/C" to see how much memory is actually available in your system for running programs. As I note there, 50K is about the minimum space necessary to run Q/C.

Sample RMAC Session

If you set up your working disk as suggested in the previous section, you should have plenty of space left. The sample session needs about 5K of disk space to hold the files which will be created.

Step 1: Compile the program (reads HELLO.C, writes HELLO.ASM)

```
A>CC HELLO -A
*** read    HELLO.C
*** include qstdio.h
*** resume  HELLO.C
=== main()
Q/C Compiler V3.x (8080) Copyright (c) 1984 Quality Computer Systems
Symbol table entries left: 136  Memory unused: 1905 bytes
Literal space left: 986 bytes   Macro space left: 916 bytes
No errors found
```

Step 2: Assemble the output (reads HELLO.ASM, writes HELLO.REL)

```
A>RMAC HELLO $PZ-S
CP/M RMAC ASSEM 1.1
000E
003H USE FACTOR
END OF ASSEMBLY
```

Step 3: Link the program with the Q/C library (reads HELLO.REL and CRUNLIB.REL, writes HELLO.COM)

```
A>LINK HELLO,CRUNLIB[S,$SZ]
LINK 1.3
(link messages)
```

Step 4: Run the program (HELLO.COM)

```
A>HELLO
Hello, world
```

4 Getting Started ... FAST !

Sample M80 Session

If you set up your working disk as suggested in the previous section, you should have plenty of space left. The sample session needs about 5K of disk space to hold the files which will be created.

Step 1: Compile the program (reads HELLO.C, writes HELLO.MAC)

```
A>CC HELLO
Q/C Compiler V3.x (Z80) Copyright (c) 1984 Quality Computer Systems
*** read    HELLO.C
*** include qstdio.h
*** resume  HELLO.C
=== main()
Symbol table entries left: 136  Memory unused: 1905 bytes
Literal space left: 986 bytes   Macro space left: 916 bytes
No errors found
```

Step 2: Assemble the output (reads HELLO.MAC, writes HELLO.REL)

```
A>M80 =HELLO

No Fatal error(s)
```

Step 3: Link the program with the Q/C library (reads HELLO.REL and CRUNLIB.REL, writes HELLO.COM)

```
A>L80 HELLO,CRUNLIB/S,HELLO/N/E

Link-80  3.42  19-Feb-81  Copyright (c) 1981 Microsoft

Data    0103    0C17    < 2949>

37114 Bytes Free
[0111   0C17       12]
```

Step 4: Run the program (HELLO.COM)

```
A>HELLO
Hello, world
```

Sample CWA Session

If you set up your working disk as suggested in the previous section, you should have plenty of space left. The sample session needs about 5K of disk space to hold the files which will be created.

Step 1: Compile the program (reads HELLO.C, writes HELLO.MAC)

```
A>CC HELLO
Q/C Compiler V3.x (Z80) Copyright (c) 1984 Quality Computer Systems
*** read    HELLO.C
*** include qstdio.h
*** resume  HELLO.C
=== main()
Symbol table entries left: 136  Memory unused: 1905 bytes
Literal space left: 986 bytes   Macro space left: 916 bytes
No errors found
```

Step 2: Assemble the output (reads HELLO.MAC, writes HELLO.REL)

```
A>CWA HELLO
CWA Z80 Assembler V x.x Copyright -(c)- ZEE MicroWare 1983

Code Length in Bytes = 0015 (21)
Data Length in Bytes = 000E (14)
No Errors
```

Step 3: Link the program with the Q/C library (reads HELLO.REL and CRUNLIB.REL, writes HELLO.COM)
NOTE: The comma before the slash is required.

```
A>CWLNK HELLO,/X

CWLNK V x.x Copyright -(c)- ZEE MicroWare 1983
Processing File = HELLO   .REL
Processing File = CRUNLIB .REL
Library Scan

  Entry =0111  End =0C20  File Size in K = 4
  Kilobytes of Free Memory = 46
```

Step 4: Run the program (HELLO.COM)

```
A>HELLO
Hello, world
```

Now that you have compiled and run a program you can relax. Sit back and read some more of the manual; then you'll be ready to use Q/C with some real C programs.

1.5 Using QRESET to Customize Q/C

Before you read this section, please realize that unless you are using RMAC, you don't need to bother with any customization right now. Even if you do use RMAC, all you'll save by customizing right now is that you won't have to use the -A option each time you compile.

The QRESET program lets you change two major characteristics of the Q/C compiler. First, you can change some default compiler options, such as whether it initializes large arrays. Second, you can change the size of important tables used by the compiler. (It is unlikely that you'll ever need to change the table sizes since they are set to large values. If a table size is exceeded, Q/C gives you an error message saying which table must be enlarged.)

You run QRESET by typing the command

```
A>QRESET
```

This will change the file CC.COM on the currently logged drive. If Q/C is on another drive or you have changed its name, QRESET will also accept a drive specifier or a different file name. For example, to change CC.COM on drive B say

```
A>QRESET B:
```

As another example, if you change the compiler name to QC.COM, say

```
A>QRESET QC
```

and you will change QC.COM on the drive currently logged. Finally, if QC.COM is on drive B, the command

```
A>QRESET B:QC
```

will change it.

QRESET announces itself like this:

```
QRESET customization program for Q/C Compiler V3.x
```

The version number is given because QRESET checks to see that the file it is changing has the same version number. If there is a difference you will get the following messages:

```
This version of QRESET only works with Q/C V3.x

No changes made
```

This prevents you from destroying your executable compiler.

Another check is made to insure that the file being changed actually appears to be a Q/C compiler. If it does not, you will get the message:

```
CC.COM does not look like a Q/C Compiler -
 either the file is damaged, or you have changed
 the order of global variables in CGLBDEF.C
```

If everything looks all right, QRESET first reports the size of your CP/M Transient Program Area (TPA) like this

```
Your CP/M TPA size is: 50K
```

This is calculated from the jump address at location 6H. If this is less than 50K, Q/C will not run without decreasing the size of some of the compiler tables. If your TPA size is less than 49K, Q/C will probably not run at all.

Next QRESET reports each of the current settings and asks you for a new value. To leave a setting unchanged, type a carriage return. If you just want to see what the current settings are, answer all the questions with a carriage return and no changes will be made. If you change your mind at any point during a QRESET run, you can quit by answering ^C to any question. None of your previously specified changes will take effect.

The following two sections show the rest of a sample dialogue. Each exchange is followed by a short explanation. Notice that when you specify a change, QRESET confirms the change that it will make.

Changing Compiler Default Settings

```
Currently compiler will generate code for M80          [8080 version only]
    Enter A (RMAC), M (M80) or <CR> for no change: a
    Compiler changed to generate code for RMAC
```

The underlined response a tells QRESET to change the default to RMAC code. Notice that your answer can be upper or lower case.

```
Currently compiler is in verbose mode
    Enter T (terse), V (verbose) or <CR> for no change:
```

Q/C is sent to you in the verbose mode. As it runs it will notify you when it starts reading an input or include file and when it starts compiling a new C function in your program. See the section "Compiler Output" for an illustration of verbose mode. If you change to terse mode, the only thing you see on your screen during a compilation is a summary at the end.

```
Currently compiler pauses after 6 errors
    Enter new size or <CR> for no change:
```

No change is made to this setting. When verbose mode is on, Q/C automatically pauses after each error message to prevent the message from being forced off the screen by the verbose mode output. When terse mode is on, the error pause count is the number of errors that Q/C will write on the console before it pauses. This prevents error messages from scrolling off the screen if you walk away during a compile. Each error message takes three lines. The pause count is initially set to 6 which works nicely on a 24 by 80 screen. This still allows enough lines for the summary messages at the end of the compile if the final screen has six errors.

```
Currently compiler does not initialize large arrays
    Enter I (initialize), N (do not initialize) or <CR> for no change:
```

The default is to skip initialization of arrays which are longer than 128 bytes. The C language definition says that all global and static variables initially contain zero if no explicit initialization is given. However, if your C program has large arrays, the .COM file can be quite large. Because of this, arrays larger than 128 bytes are not initialized automatically. You can still initialize selected arrays by initializing at least one element. Q/C will then set the remaining elements to zero.

```
Currently compiler excludes redirection
    Enter R (redirect), N (do not redirect) or <CR> for no change:
```

The default of excluding redirection is left unchanged. Since the linkers only load the library functions which are needed, a program which does not need redirection will be smaller. Depending on which functions are actually required, you can reduce the size of the .COM file by 1 to 5K. You can still include redirection in individual programs by using the -R compiler switch described in Section 2.3 "Compiling a Program."

Changing Compiler Table Sizes

```
SYMBOL TABLE size is: 150 entries
    Enter new size or <CR> for no change: 200
   New SYMBOL TABLE size is: 200 entries
```

Here the compiler symbol table is being changed from 150 to 200 entries. This is ordinarily done in response to the compiler error message "Symbol table overflow." If this happens with any table, the best thing to do is to increase the size by a fair amount (say 20%) and try again.

```
MEMBER TABLE size is: 50 entries
    Enter new size or <CR> for no change:

TYPE TABLE size is: 50 types
    Enter new size or <CR> for no change:

LITERAL (string) POOL size is: 1000 characters
    Enter new size or <CR> for no change:

MACRO (#define) POOL size is: 1000 characters
    Enter new size or <CR> for no change:

SWITCH/LOOP nesting depth is: 10 levels
    Enter new size or <CR> for no change:

SWITCH/CASE TABLE size is: 50 cases
    Enter new size or <CR> for no change:
```

The last six compiler table sizes are left unchanged. Normally, the only time you change any of the table sizes is in response to a compiler error message saying that a particular table has overflowed. When you change table sizes, you should be sure that you have enough memory available. The sizes you specify for the literal pool and macro pool are already in bytes. Each symbol table entry takes 15 bytes and each type table entry takes 9 bytes. The member table, the switch/loop queue and the switch/case table each require 4 bytes per entry. The summary messages at the end of each compilation tell you how much memory was unused by the compiler. Since this will vary from program to program, be conservative.

```
B:CC.COM has been changed
```

QRESET informs you that the changes have been made and shows you exactly which file was changed. If you do not request any changes this message will be "No changes made."

2

Using the Q/C Compiler

2.1 A Quick Overview

Q/C reads one or more files containing the source code for your C program and compiles (translates) it into 8080 or Z80 assembler language. This version of your program is then assembled and linked with a group of subroutines called the function library to form a complete CP/M .COM file which can be run simply by typing its name.

The function library contains three types of subroutines. First are the compiler support routines. Several common operations such as 16-bit arithmetic and logical tests are done by calling these subroutines rather than inserting the assembler code each time the operation is performed. This makes the program shorter but slightly slower. Calls to the compiler support routines are generated automatically by Q/C, so you will normally not be aware of them.

The second group includes the input/output (I/O) functions. These are the functions you use in your program to do terminal and disk I/O. As an example, the C statement

```
putchar('a');
```

calls the library function **putchar** which will place the character 'a' on the terminal screen at the current cursor position.

The final group contains the string and character handling routines. These are the functions you use to work with strings and characters. For example, to copy string b to string a you would say

```
strcpy(a, b);
```

The .COM file generated by Q/C is a true "native code" compiled version of your program. Since no interpretation is being done, programs run fast. On a 4 Mhz Z80, the "do-nothing" loop

```
for (i = 0; i < 32767; ++i);
```

runs in less than 5 seconds. Try that in BASIC and see how long it takes!

2.2 A Closer Look at a Q/C Program

Let's look at the program HELLO.C which is supplied on your disk:

```
#include "qstdio.h"
main()
        {
        printf("Hello, world\n");
        }
```

Although it's not long, it illustrates several important features of a Q/C program. It starts with the **#include** directive which tells Q/C to include the file **QSTDIO.H** as part of the input to the compiler. This file contains the definition of some frequently used constants like TRUE (1) and EOF (-1). It also includes the definition of the FILE data type and several external variables needed by I/O library functions. Any program which uses the I/O library must include **QSTDIO.H**.

HELLO.C consists of a single function named **main**. A program may consist of any number of functions. If you want it to be a stand-alone program (meaning one which will become a .COM file which you can run), it must contain a function named **main**.

Since your external variable names and function names are labels in the assembler program, they must meet certain requirements that the assemblers put on label names. The two things you must remember are:

1. Label names are translated to upper case, so names like **holdx**, **Holdx**, and **HOLDX** which are unique in C are all equivalent in assembler.

2. The assemblers retain only the first 6 characters of an external label. This means that the unique C names **ungetch** and **ungetc** are considered the same by the assemblers.

This version of Q/C insures that none of your global names will conflict with any assembler reserved words. This means that it is perfectly legal, for example, to define a global variable **hl** even though **HL** is the name of an 8080/Z80 register.

So, a quick review of the requirements for a Q/C program:

1. If you call any library functions, the program must start with the preprocessor statement **#include "qstdio.h"**.

2. If the program will be run as a stand-alone program (a .COM file), it must contain a function named **main**.

3. All external variables and function names must be unique within the first six characters regardless of case. (The distinction between upper and lower case which makes names unique in C is lost in the assembler and will cause duplicate label errors.)

2.3 Compiling a Program

Some Examples

Now that you have seen what a Q/C program should look like, let's see how you compile it. The simplest possible way to compile (when everything fits on the same disk) is:

```
A>CC HELLO
```

CC is the name of the compiler, and the file containing the program is **HELLO.C**. If you don't specify a file extension, Q/C will use .C as the default. If no output file name is given, the input file name is used with the file extension **.MAC** since the default is to produce code for M80. Thus the output file is called **HELLO.MAC**.

Although this simple command will handle many situations, there are a several options and additional ways of naming files that can be used. Let's look at some common situations first. Suppose you want to rename the output file. Type

```
A>CC HELLO -O GOODBYE
```

The input file is still **HELLO.C**. The dash tells the compiler you are giving it an option, and the O says you are specifying the output file. The space between O and the output file name **GOODBYE** is required. Since no file extension is given, Q/C will add the extension **.MAC** as before, making the output file **GOODBYE.MAC**.

The file names can also have a drive specifier, and the automatic file extension can be stopped. If you type

```
A>CC HELLO. -O B:
```

the input file will just be **HELLO** with no extension. When Q/C finds a period in the file name it doesn't change the name. The drive specifier B: on the output file name will make it **B:HELLO.MAC**. In this case, Q/C uses the input file name plus the default extension .MAC to form the output file name. As you can see, you can use any file names and any drives you want, or you can let Q/C do most of the work for you.

Sometimes your program will be so large you want to keep it in smaller pieces to make it easier to edit. If your program is in two parts it could be compiled like this

```
A>CC PART1 PART2 -O BIGPROG
```

This would compile **PART1.C** and **PART2.C** and call the output **BIGPROG.MAC**.

Summary: Running the Compiler

The preceding examples show the pattern for running the compiler. You type the compiler name **CC** followed by the input file(s). Then you type the options, if any, and finally the new file name if desired. The general form is

```
CC infile ... [-options] [outfile]
```

where the fields are separated by blanks. All that is required is the compiler name **CC** and at least one **infile**. There can be as many input file names as you need. The compiler assumes that all fields up to the **-options** field are the names of input files. The output file name **outfile** if specified must come after the options. If you don't specify the output file name, it will be the same as the first (or only) input file name with .ASM or .MAC added as the extension.

Summary: Specifying File Names

A file name is specified as a CP/M unambiguous file name but you do not need to specify all of the parts -- the compiler will generally fill in the missing pieces. The general form of a file name is:

```
[d:][filename][.ext]
```

Although all parts are optional in different cases, you must always specify at least one of the parts.

d: is an optional drive specifier. If an explicit drive is not specified, Q/C will read input files from the currently logged drive and write the output file on the same drive as the first (or only) input file.

filename must always be given for input files. Output files need only a drive specifier if you simply want to put the output on a different drive from the input. In this case, the output file name will be the same as the first input file name with the default file extension added.

.ext is an optional file extension. If you don't give an extension, a default file extension will be supplied. If you want to specify a file which has no file extension, put a period after the file name and no extension will be added.

2.4 Compiler Options

Compiler options are specified by "switches" (a dash, then a letter) on the command line when you run the compiler. If you do not specify any options you will get the default settings shown below. Notice that many of these defaults can be changed by running the program QRESET included on your distribution disk. See section 1.5 "Using QRESET to Customize Q/C" for instructions on using QRESET.

-A Generate output for the Digital Research RMAC assembler using .ASM as the file extension for the output file. The default can be changed permanently with QRESET.
Note: this option is not in the Z80 version -- you always generate code for M80.
Default: generate output for M80.

-C Generate a commented assembler program including the C text and additional comments to indicate what the assembler code is doing. See Appendix C "Sample Compiler Output" for an example of what a file produced with the -C option looks like. If the output is for RMAC, the exclamation point (!) in C statements will be translated to a number sign (#). Otherwise the assembler thinks that the the rest of the C statement is a new assembler statement.
Default: no comments in the assembler program.

-D Debug mode. Send the output to the console rather than to a disk file so you can look at it immediately. This is very useful for debugging changes made to the compiler.
Default: output goes to a disk file.

-I Turn on automatic initialization of large arrays. Normally Q/C only does automatic initialization of global and static arrays which are no longer than 128 bytes. This speeds up the assembly and reduces the size of the .COM file. If you just want selected arrays initialized , you can force it without using -I. Simply initialize at least one element of the array and Q/C will set the rest to zero.
Default: don't initialize large arrays

-L Do a library generation run as part of building a new library. All globals defined at the beginning of the program will be placed in the normal output file. Each function will be written to a separate file named **function.MAC** or **function.ASM** where **function** is the actual name of the C function.
Default: don't do a library generation

-M Generate code for the Microsoft M80 or The Code Works CWA assembler using .MAC as the automatic file extension for the output file.
Note: this option is not in the Z80 version -- you always generate code for M80 and CWA.
Default: generate output for M80.

-O Specify a name for the output file. If you do not specify a file extension, the compiler will use the file extension appropriate for the assembler you are generating code for. If you specify only a drive, the output file name will be the same as the input file with the appropriate extension, and it will be written on the specified drive.
Default: the first input filename with the file extension ASM or MAC as appropriate.

-R Include the redirection capability in the compiled program. When you specify -R, you also get automatic closing of all buffered files at end of run. Notice that if your program does not need the redirection capability, you can reduce the size of the .COM file by excluding redirection. For more information on redirection and automatic closing of files, see Section 3.3 "I/O Redirection."
Default: do not include redirection capability.

-S Generate ROMable code and optionally specify where you want the stack to start. Specifying -S tells Q/C to do some things differently since the code generated will be loaded in ROM. Normally Q/C starts its stack immediately below the CP/M BDOS. ROMable code may have specific location of RAM where the stack must be, so specifying a hexadecimal address after -S tells Q/C where the stack must start. For example, if the stack should start at OEFFFH, say -SEFFF when you run Q/C. In addition, -S eliminates the calls to the library routines which parse the CP/M command line and which reboot CP/M at the end of the program. See Section 4.2 "Writing ROMable Programs" for more information.
Default: do not generate ROMable code.

-T Generate trace messages in your program. When you run your program every function will print its name in the messages

```
>function-name
<function-name
```

each time it is entered and exited. This is useful when your program mysteriously dies somewhere, and you don't want to trace it at the assembler language level to see where it is and how it got there.

If you don't want to trace all functions, you can turn tracing on and off with "smart" comments. If you have enabled tracing by specifying -T, then the compiler looks for comments of the form

```
/* $ +T [any text] */
```

where the white space is optional. Initially, tracing is on. -T turns tracing off and +T turns it back on. This allows you to trace only the functions you are interested in by placing "smart" comments around them.
Default: no trace messages.

-V Toggle the compiler between verbose and terse mode (see the section "Compiler Output" for an example of the effect of this option). If you change the default to terse mode using QRESET, then specifying -V will turn verbose mode on.
Default: compiler is in verbose mode.

When you specify options, all but -S can be typed together as well as separately. For example, the two commands

```
A>CC DISKDUMP -A -O B:
```
and
```
A>CC DISKDUMP -AO B:
```

will both compile **DISKDUMP.C**, generate assembler code for the Digital Research RMAC assembler because of the **-A** option, and change the name of the output file to **B:DISKDUMP.ASM** (forcing it to drive B) because of the **-O** option.

2.5 Compiler Output

Normally you will simply let the compiler output go to a disk file and then assemble it without looking at it. When the compiler is in "verbose" mode the output looks like:

```
A>CC PROGRAM
Q/C Compiler V3.x (Z80) Copyright (c) 1984 Quality Computer Systems
*** read    PROGRAM.C
*** include qstdio.h
*** resume  PROGRAM.C
=== main()
Symbol table entries left: 135  Memory unused: 2655 bytes
Literal space left: 873 bytes   Macro space left: 674 bytes
No errors found
A>M80 =PROGRAM
  etc.
```

If the compiler is in "terse" mode (because -V was specified or because QRESET changed the default), the only output you will see is the compiler summary at the end. Notice that a compile of a large size program of say 1000 lines will take several minutes, and there won't be any indication of what Q/C is doing during this time. In terse mode, the previous run looks like:

```
A>CC PROGRAM -V
Q/C Compiler V3.x (Z80) Copyright (c) 1984 Quality Computer Systems
Symbol table entries left: 135  Memory unused: 2655 bytes
Literal space left: 873 bytes   Macro space left: 674 bytes
No errors found
A>M80 =PROGRAM
  etc.
```

If you want to see what the assembler code for your program looks like for any reason, the **-C** option will generate some helpful assembler comments. See Appendix C "Sample Compiler Output" for an example of a simple C program and the assembler output using the **-C** option.

2.6 Error Messages

Unfortunately, none of us writes perfect C programs all the time. When Q/C finds something it doesn't like it prints a three line message on the console. For example, compiling the file **ERR.C** which contains the program:

```
/* error - force an error message */
int i, n[80];
error() {
        i = n[0;        /* bad subscript syntax */
        }
```

will generate the error message

```
<ERR.C> @ 4: Missing punctuation -- assumed present: ]
 i = n[0;
       ^
```

All error messages have this general form. First, the error message is printed showing the name of the disk file being read enclosed in angle brackets <>, followed by an at sign @ and the line number in the file. In our example **<ERR.C> @ 4** means the error is at line 4 in the file **ERR.C**. This allows you to go directly to the erroneous line in your editor. Next the line with the error is shown with all extra white space removed. Finally, a line with only a caret ^ is printed to show where the compiler was looking when it found the error.

The error message itself may be just a general message like "Must be lvalue", or it may include some specific information for this particular case after the message. The earlier example not only says that the needed punctuation is missing, but also that the compiler supplied the closing bracket "]" it was looking for.

If the error occurs inside an **#include** file, the error message will tell you. If the file **ERR.C** is used as an **#include** file in the source file **INCLERR.C** like this:

```
/* inclerr.c - force an error in a #include file */

#include "err.c"
```

the error message will change to

```
<INCLERR.C> @ 3: #include <ERR.C> @ 4: Missing punctuation --
        assumed present: ]
 i = n[0;
       ^
```

This says that the file being compiled is **INCLERR.C**, but that there is an **#include** command at line 3. The file being included is called **ERR.C** and the error is actually at line 4 of the file **ERR.C**. If you use nested include

files, you will see only the input file name from the command line and the include file currently being compiled. The intermediate include file names will not be shown.

Appendix B is an alphabetic listing of all the error messages given by Q/C along with an explanation of possible causes of the error.

Tips for Interpreting Error Messages

Normally the compiler has scanned beyond the error by the time it realizes that there is an error. In the previous example, `i = s[0;`, the caret "^" was pointing at the semicolon ";" because that is where Q/C was expecting to find the closing bracket "]". Since white space is ignored, if the semicolon were also missing, the compiler would have scanned to the next line looking for a non-white space character. In this case it would have reached the closing brace "}" which terminates the program before realizing that there was a problem. The line that it reported in the error message would be one line too far. Fortunately, this will usually happen only if you omit a required semicolon at the end of a line. In general, you should look to the left of the caret for the error, and occasionally on the line before the line shown.

2.7 Assembling a Program

After you compile a program, you must assemble the compiler output and link it with the function library to build a .COM file to run. In Chapter 1 "Getting Started" you saw a brief example. This section shows the process in more detail.

Using M80

Using the M80 assembler you compile, assemble, link and produce an executable .COM file, by typing the underlined parts:

```
A>CC progname -options
(compiler messages)

A>M80 =progname
(assembler messages)

A>L80 progname,CRUNLIB/S,progname/N/E
(linker messages)
```

At link time, the /S option causes L80 to search **CRUNLIB.REL** and include only those functions needed by your program.

Using RMAC [8080 version only]

With RMAC, compiling, assembling and linking looks like this (you type the underlined parts):

```
A>CC progname -options
(compiler messages)

A>RMAC progname $PZ-S
(assembler messages)

A>LINK progname,CRUNLIB[S,$SZ]
(load messages)
```

You will probably want to suppress the .PRN file and the .SYM file produced by RMAC. This is accomplished by the options $PZ-S. When you link, you want only those functions which are actually needed to be included in the .COM file. This is specified by the option S following the left bracket ([) which tells LINK to search CRUNLIB.REL and include only the needed functions. The remaining options suppress the listing and recording of the global symbol table.

3

Running Your Q/C Programs

When you run a Q/C program, it interacts with CP/M in various ways. First, you can pass parameters to a program from the CP/M command line. Second, you can redirect the standard input and output files on the command line so that they refer to disk or the printer rather than the console. This chapter tells you how to use these and other features.

3.1 Command Line Arguments

When you run your program, you can pass parameters to it by using the normal **argc, argv** mechanism. In the current release of Q/C there are three minor differences from the way Kernighan & Ritchie describe command line arguments.

To show how things work in Q/C, suppose you have a program called **FIND** which searches a file for a given string of characters. If you want to find all the **for** statements in the program **COMPARE.C** you would run the program like this

```
A>FIND COMPARE.C \Lfor
```

The first difference is that CP/M translates the command line to upper case so **FIND** receives the second argument as **\LFOR**. This means you have to use some kind of escape sequence to indicate that **for** is lower case. The \L in front of **for** is intended to tell **FIND** that the rest of the argument is lower case. This escape sequence is strictly for illustration. Q/C does not recognize \L as having any special significance in a command line argument. The program **FIND** must interpret \L as an indication that the rest of the argument is lower case.

The program **FIND** would start out like this

```
main(argc, argv)
int argc;
char *argv[];
        {
        char *filename, *string;
        filename = argv[1];
        string = argv[2];
```

As usual, **argv[1]** is a pointer to the second argument on the command line, so setting **filename** equal to **argv[1]** causes **filename** to point to the string **"COMPARE.C"**. If you only need to refer to the entire string, it is not necessary to define and use **filename** of course. For example, you can open the file that **FIND** is to search by saying

```
fopen(argv[1], "r")
```

since **fopen** only needs the pointer to the string containing the file name.

The second difference is that **argv[0]** points to a null string **""** rather than **"FIND"**, the name of the program being run, since CP/M only preserves the portion of the command line that follows the name of the program.

The final difference is that Q/C allows an argument to contain blanks or tabs by putting quotes around it. If you wanted **FIND** to locate all the endless **for** loops which you wrote as **for (;;)** with a blank between the keyword **for** and the left parenthesis, you would say

```
A>FIND COMPARE.C "\Lfor (;;)"
```

When Q/C finds an argument starting with a quote (") it makes everything up to the next quote or the end of the command line part of the argument. As before, the escape sequence \L tells FIND.C that the letters are lower case.

In summary, the three differences from standard command line argument passing are:

1. Since CP/M translates the command line to upper case, all the **argv** strings will be in upper case regardless of how you typed them on the command line.

2. **argv[0]** is a null string rather than the name of the program that is being run.

3. An argument enclosed in quotes can contain blanks or tabs.

3.2 Standard I/O Files

Q/C supports standard files which you can use without opening or closing much like standard C. These files include the standard input file **stdin**, the standard output file **stdout**, and the standard error file **stderr**. They are all normally assigned to the console (CP/M CON: device). The standard file names are declared in the file **QSTDIO.H** which should be included in any program using the I/O library. This file also defines the constants you normally need such as NULL and EOF.

The standard files can be used in several ways. The functions **getchar** and **gets** read from **stdin**, while **putchar** and **puts** write to **stdout**. Thus, the simplest copy program you can write is

```
/* copy.c - copy stdin to stdout */
#include <qstdio.h>
main()
        {
        int c;
        while ((c = getchar()) != EOF)
                putchar(c);
        }
```

This simple program can be useful as you will see in the discussion of I/O redirection.

You can also use the standard filenames in any of the character (buffered) I/O functions. For example

```
getc(stdin)
```

has the same effect as

```
getchar()
```

They both read the next character from **stdin**.

When you use the standard file names this way there are three requirements:

1. You must include **QSTDIO.H** in your program.

2. You must not open or close the standard file names.

3. You must not assign a value to the standard file names. In other words, don't do this

```
stdin = 5;      /* ILLEGAL */
```

3.3 I/O Redirection

Q/C supports I/O redirection from the command line. This means that the files **stdin** and **stdout** can be redirected from the console to disk, and that **stdout** can be redirected to the printer.

Notice that redirection is optional. If a program does not need redirection, you can reduce the size of the .COM file by excluding redirection. For example, a program which does only terminal I/O will not need the library functions which open and close files. If you exclude redirection and you do disk I/O with the buffered I/O functions, you will have to close the output files yourself. This can be done by ending your program with a call to the **exit** function or by closing the output files individually with the function **fclose**.

To see how redirection is done, consider the program **copy.c** described in the last section. If you run it like this

```
A>COPY
```

it will read from the console and write to the console. This is not very useful because it will just repeat everything you type. However, you can redirect **stdin** to a disk file by typing the less-than symbol "<" in front of the file name like this

```
A>COPY <OTHELLO.PRN
```

and print the file **OTHELLO.PRN** on the screen. To copy one file to another, type the greater-than symbol ">" in front of the destination file name. For example

```
A>COPY <OTHELLO.C >B:OTHELLO.C
```

will copy **OTHELLO.C** from drive A to drive B. The ">" in front of **B:OTHELLO.C** redirects **stdout** from the console to the file **B:OTHELLO.C**. In both of these examples, the disk files are opened and closed automatically; you never mention them in the program **COPY.C**.

Finally, you can redirect **stdout** to the printer like this

```
A>COPY <OTHELLO.C >LST:
```

Q/C recognizes **LST:** (or **lst:** or any combination of upper and lower case) as a reference to the CP/M LST: device (the printer). For a way to write to the printer without using **stdout**, see the description of the **fopen** function in Section 5.3 "Function Descriptions".

The arguments which do redirection are not passed to your program. They may be mixed in with other command line arguments, and they may appear in any order.

4

Advanced Q/C Topics

4.1 Interfacing with Assembly Language

Since C was designed for systems programming, it is seldom necessary to use assembly language. C programs can be written and debugged much more quickly than assembler programs, and are much easier to understand and change later.

Infrequently you will need to write a function in assembly language. For example, it may be necessary to speed up a heavily used function. For these rare occasions, Q/C provides the **#asm** and **#endasm** preprocessor directives. These allow you to embed assembly language in your C program. The best way to do this is to make the assembler routine a function which you call from your C program.

A Simple Example

This example shows the basic requirements for writing an assembly language subroutine which will be called as a C function. We will rewrite the standard library function **isdigit** which looks like this in C:

```
isdigit(c)
int c;
        {
        return (c >= '0' && c <= '9');
        }
```

isdigit checks the argument **c** to see if it is between 0 and 9 and returns true (non-zero) or false (zero). The same program in assembler is:

```
isdigit()
        {
#asm
        LXI     H,2     ;get arg off of stack
        DAD     SP
        MOV     A,M
        CPI     48      ;'0'
        JC      ?isdgl
        CPI     58      ;'9'+1
        JNC     ?isdgl
        ORA     H       ;set Z flag to zero
        RET
?isdgl: XRA     A       ;set Z flag to one
        MOV     H,A     ;return FALSE
        MOV     L,A
#endasm
        }
```

The output from the compiler will be:

```
        PUBLIC  isdigit?
isdigit?:
        LXI     H,2
        DAD     SP
        MOV     A,M
        CPI     48
        JC      ?isdgl
        CPI     58
        JNC     ?isdgl
        ORA     H
        RET
?isdgl: XRA     A
        MOV     H,A
        MOV     L,A
        RET
```

Since the function is defined in C, the compiler takes care of several housekeeping chores. First, it appends a ? to the function name to eliminate conflicting with assembler reserved words. Second, it generates the assembler pseudo-op PUBLIC to make this function name known to other files when you link. Although you can't see it, the compiler also enters the name **isdigit** in its symbol table. Then any calls in this same C program will know that **isdigit** is defined as a global name, so they won't cause an EXTRN pseudo-op to be generated erroneously.

Notice that the assembly language version does not include the argument **c** in the function definition. You must NOT define the argument(s) when you supply the entire function. If you do define them, the compiler will generate calls to entry and exit routines in the run-time library. These calls are unnecessary in this case and they can cause problems. Since the function returns a value, the compiler expects the Z flag to be set to correspond to

the return value. When the exit routine is called, it will wipe out your Z flag setting. A final note is that the compiler generates the closing **RET** statement.

The first few lines of the assembler code get the value of c off the stack. When you call **isdigit** in your C program, the compiler generates code that looks like this:

```
LHLD    c?
PUSH    H
CALL    isdigit?
```

So, when **isdigit** is entered the stack looks like this:

```
SP -->  return address
SP + 2  c
```

The label **?isdgl** used in this function was chosen to meet several requirements. It starts with a ? so that it doesn't conflict with any global names defined in C. C names must start with a letter or an underscore (_) which gets changed to an at sign (@) so they will never start with ?. The compiler generates its own labels in the form **?nnn** where **nnn** is a number (for example ?27). So as long as the second character of your labels is a letter, there will never be a conflict with compiler-generated labels. Finally, your labels must be unique within the first 6 characters to satisfy all the assemblers.

Once you know what you want to return (zero or non-zero) you must place it in HL if it is not already there. This is where the compiler expects the return value from a function call. Also, the compiler expects the Z flag to be set to one if the return value is zero, or set to zero if the return value is non-zero.

A Larger, Portable Example

The example on the next page shows how to write an assembler function which can be called from Q/C and how to call a Q/C function from assembler. It also shows how to maintain portability using conditional compilation when your function can also be written in C. This example is similar to the symbol table search in the Q/C compiler.

```
#ifdef PORTABLE                                                    (1)
findglb(sname)
char *sname;
        {register char *ptr;
        for (ptr = symtab; ptr < glbptr; ptr += SYMSIZE)
                if (streq(sname, ptr))
                        return ptr;
        return NULL;
        }
#else
findglb() {                                                        (2)
        extern char *glbptr, *symtab;                              (3)
        extern streq();
#asm 8080                                                          (4)
        PUSH    B       ;save calling program stack frame ptr      (5)
        LXI     H,4     ;get address of sname on stack             (6)
        DAD     SP
        MOV     C,M     ;put it in BC
        INX     H
        MOV     B,M
        LHLD    glbptr? ;address of last global symbol             (7)
        XCHG            ; in DE
        LHLD    symtab? ;address of first global symbol
?fglbl: MOV     A,H     ;is ptr < glbptr?                          (8)
        CMP     D
        JNZ     ?fglb2
        MOV     A,L
        CMP     E
?fglb2: JNC     ?fglb9  ;no, so we didn't find it
        PUSH    D       ;save glbptr
        PUSH    B       ;name is first arg to streq                (9)
        PUSH    H       ;ptr is second arg
        CALL    streq?  ;test the two strings for equality         (10)
        POP     H       ;clear args off the stack which also       (11)
        POP     B       ; restores the values in the registers
        JNZ     ?fglb8  ;if the strings matched, we're done
        LXI     D,15    ;otherwise, move to next symbol
        DAD     D       ;ptr += SYMSIZE
        POP     D       ;restore glbptr
        JMP     ?fglbl  ;loop and try again
?fglb8: POP     D       ;matched, finish clearing stack            (12)
        POP     B       ;restore caller's stack frame pointer      (13)
        RET
?fglb9: POP     B       ;restore caller's stack frame pointer
        LXI     H,0     ;no match, return NULL                     (14)
        XRA     A       ;set Z flag to indicate FALSE
        RET
#endasm                                                            (15)
        }
#endif
```

Explanation of the Example

First, a brief explanation of what the function is doing. **findglb** searches the global symbol table for the symbol name pointed to by **sname**. The C function **streq** is called to compare **sname** with the name stored in successive entries in the symbol table. If a match is found a pointer to this entry is returned. Otherwise NULL is returned.

In the following discussion the numbers in parentheses refer to the numbered lines in the example.

(1) One of the most important points illustrated in this example is the use of conditional compilation (the **#ifdef PORTABLE**) to retain a version of the subroutine which is portable to other machines. If you decide to move this routine to an 8086, for example, you simply **#define PORTABLE** ahead of the **#ifdef** and the portable C code will be compiled. When **PORTABLE** is not defined, the assembly code will be included in the output file giving a speed increase.

(2) You must not declare the arguments to the function when you are going to manage the stack yourself. Leaving out the arguments suppresses the calls normally generated by the compiler to entry and exit routines which maintain a constant stack frame using register BC (and in the Z80 version IX).

(3) If you refer to global variables defined in C you must declare them **extern** as shown here. Then the compiler will figure out whether assembler EXTRN pseudo-ops need to be generated depending on whether these names are defined in the current source file or elsewhere.

(4) and (15) The entire assembly language routine is surrounded by the preprocessor commands **#asm...#endasm**. This tells Q/C to copy everything in between directly to the output file. Only two things are done while **#asm** is in effect. An **#include** command will be cause the requested file to be copied, and assembly language comments will be stripped out. The presence of "8080" in the #asm command causes the Z80 version of Q/C to generate the M80 pseudo-op ".8080" at the beginning of your assembler code. It will also generate a ".Z80" when it reaches the **#endasm** command. The 8080 version of Q/C will ignore the "8080", however, allowing you to write one version of your program which can be compiled and assembled with either flavor of the compiler.

(5) In this version of Q/C, a constant stack frame pointer is maintained in register BC as mentioned in (2). This allows the compiler to reference all local automatic variables with a constant offset from BC. It also means that every function must preserve the contents of BC so that the stack frame pointer of the calling function will still be set when returning to it.
Z80 users NOTE: The index register IX must also be preserved.

(6) When you retrieve arguments you must remember that they are pushed onto the stack before the CALL to the function. The CALL then pushes the return address on the stack. When **findglb** is entered the stack looks like this

```
SP —> return address
SP + 2  sname
```

When there is more than one argument, the last argument pushed will be the first one above the stack pointer since they are pushed onto the stack in the order they are found in the call. The five lines starting here show how to get the argument **sname**. When BC is pushed onto the stack in step (4), the offset of **sname** from the stack pointer is changed to 4. So, the address of **sname** is computed in HL by adding the offset 4 to the stack pointer (SP). Then the two bytes at this location are loaded into the BC register pair.

(7) You can refer to a global variable like **glbptr** by using its name with a question mark (?) appended. Q/C appends a question mark to all global names so that they will not conflict with assembler reserved words such as HL and RET. Thus, if you reference a C global name you must add the question mark. Similarly, if you define a global name in assembler, you must add the question mark yourself if you want C to find it.

(8) All of the labels in this example start with ? to prevent them from colliding with labels generated for global variables in your C program. All internal labels generated by the compiler consist of a question mark followed by a number (like ?46), so if your assembler labels start with a question mark followed by an alphabetic character, you will never have labels which duplicate compiler-generated labels.

(9) Here we start pushing the arguments for **streq** onto the stack. Notice that they are pushed in the order they are seen in the call to **streq**. In this case all the arguments are passed as the actual value. If you need to pass an array, a function or a structure or union to the called routine, you must pass a pointer to the actual variable (meaning the address of the array, function, structure or union).

(10) To execute a C function, simply CALL its name with a question mark appended. See the discussion in (7).

(11) Now the arguments must be cleared back off the stack. You must do one POP for each argument. Notice that the calling routine is the one that clears the arguments off the stack. In general, you should not rely on restoring the values in the registers this way because the called function may have altered the values passed to it.

(12) Since we pushed the value of **glbptr** on the stack earlier we must POP it back off to restore the stack pointer to the way it was when **findglb** was entered. Otherwise the RET instruction will not find the return address.

(13) Before we return, the calling routine's stack frame pointer must be restored. See the discussion in (4).

(14) The next two lines show what Q/C expects to get back from the function. Since there is a return value, it must be loaded in the HL register pair and the Z flag must be set to reflect whether the value is zero or nonzero. This is the unmatched condition which returns NULL (zero).

Using Compiler Support Routines

One thing not illustrated by this example which can simplify your assembler language programming is the use of the compiler support routines located in **CRUNLIB.MAC**. Virtually all the C operators are implemented as calls to the support routines to perform the 16 bit operation. Unary operators expect their operand in HL and binary operators expect the left operand in DE and the right operand in HL. The result is always returned in HL.

To show how these routines are used, suppose you need to multiply the two global integers xi and yi and put the result in zi. In C this would be zi = xi * yi; and in assembler you write

```
LHLD    xi?     ;load left operand
XCHG            ;move it to DE
LHLD    yi?     ;load right operand in HL
CALL    ?mult   ;do xi*yi with the library routine
SHLD    zi?     ;the result is returned in HL
```

Q/C Calling Conventions

Here is a summary of conventions you must follow when you write in assembly language:

1. Arguments are always passed as 16 bit values. **char** arguments are sign extended to 16 bits (the compiler support routine **?sxt** can be used to do this). Everything else is already 16 bits.

2. Each argument is pushed onto the stack in the order it is found, i.e. from left-to-right.

3. The value of the 8080/Z80 stack pointer (SP) and the Q/C constant stack frame pointer (BC) must be preserved. If you change either SP or BC, you must restore them before returning. The values in all other registers may be destroyed.
   Z80 users NOTE: The index register IX must also be preserved.

4. If there is a return value, it must be in the HL register pair. The Z flag must be set to indicate whether the return value is zero or non-zero. If there is no return value, the contents of HL and the setting of the Z flag are undefined (in other words you don't have to set them in any particular way).

4.2 Writing ROMable Programs

When you write programs which will be loaded in ROM, there are several things which must be done differently. The main difference is that everything in the code segment (CSEG) will end up in ROM where it can't be changed, while everything in the data segment (DSEG) will be in RAM where it won't be initialized. Another difference is that ROM programs normally don't run under CP/M so that you don't need or want to load the library routines which parse the CP/M command line to build **argc** and **argv** and which reboot CP/M at the end of the program.

You tell Q/C you want to generate ROMable code by compiling with the -S compiler option. Following the -S you specify the address that you want the stack pointer (SP) initialized to. For example, if you are compiling the program **ROMPROG.C** whose stack will start at 0DFFFH, compile it with the command

```
A>CC ROMPROG -SDFFF
```

When you specify this option, the compiler places all strings (for example "abcd") in the CSEG so they will be initialized. Of course this means that you cannot change these strings which is normally allowed. Also, if you want to set up tables you can define global variables and initialize them. These will also go in the CSEG and also cannot be changed. All uninitialized global variables will not get the default initialization and will be placed in the DSEG so they will end up in RAM. You must not define any local **static** variables because they won't be handled correctly.

If you will not be using the I/O functions or the memory allocator **malloc** in the library, you do not have to include the standard header file **QSTDIO.H** in your program. This will eliminate unneeded library variables from being placed in the DSEG. If you do this, you may need to define the symbolic constants **TRUE** and **FALSE** which normally are defined in **QSTDIO.H**.

If you do not want any DSEG variables brought in from the library you must rework the library module **CRUNTIME.MAC**. Change the definition of the Q/C "register" variables **r?1?-r?5?** from **DEFS 2** to **DEFS 0** and eliminate the end-of-memory symbols defined immediately after them. Then assemble **CRUNTIME.MAC** and replace the module "crunti" in **CRUNLIB.REL** as described in Appendix E "Maintaining the Function Library". If you make this change, be sure you do not define any local variables as **register**.

4.3 Compiling a Large Program in Parts

When a program is large it may be convenient to compile parts of it separately. With relocating assemblers like M80 and RMAC this works nicely. Each part of the C program generates a .MAC or .ASM file which is assembled to produce a relocatable .REL file. When you are ready to build the .COM file, you simply link all the .REL files with L80 or LINK.

If one part of the program changes, you compile only that part and

assemble the compiler output to produce a new .REL file. This .REL file can then be linked with the existing .REL files to create the new .COM file.

To make this concrete, suppose your program consists of two parts called PART1.C and PART2.C, and that the relocatable files PART1.REL and PART2.REL exist from previous compilations and assemblies. Now if you decide to change PART1.C and build a new version of your program which you call BIGPROG.COM, you give the following commands using M80:

```
A>CC PART1
  (compiler messages)
A>M80 =PART1
  (assembler messages)
A>L80 PART1,PART2,CRUNLIB/S,BIGPROG/N/E
  (linker messages)
```

If you are using RMAC this looks like:

```
A>CC PART1
  (compiler messages)
A>RMAC PART1 $PZ-S
  (assembler messages)
A>LINK BIGPROG=PART1,PART2,CRUNLIB[S,$SZ]
  (linker messages)
```

5

Q/C Function Library

The C language does not include any input or output. The designers decided that all I/O should be done using library functions. Also, while C has many powerful operators, it does not support frequently used functions such as finding the length of a string of characters, or copying one string to another. A positive effect of these design decisions is that C is a fairly small language that is relatively easy to implement. But even its designers needed to do I/O, not to mention character handling!

So, very early in the history of C there came to be libraries of commonly used functions. As time passed, the folks at Bell Labs found that different groups were experiencing (unnecessary) problems due to small incompatibilities in their function libraries. After a period of evolution, all concerned agreed on what is called the "Standard I/O Library". This library has been quite stable for several years, with small revisions when Version 7 of UNIX appeared, and others with UNIX System III.

An important point to remember is that the Standard I/O Library now exists in several environments, including UNIX, UNIX look-alike operating systems, and even CP/M. Thus, C programs that rely only on the Standard I/O Library are very portable.

5.1 Comparison with the Standard I/O Library

The Q/C function library includes essentially all of the Standard I/O Library and various system functions which provide access to CP/M disk I/O and other system facilities. Most of the system functions simulate UNIX system calls. All functions which are not unique to CP/M are intended to be identical to or compatible with their UNIX counterparts. Q/C function library features include:

Similarities

- All buffered sequential I/O functions
- Command line redirection of buffered I/O
- Automatic opening of **stdin**, **stdout**, and **stderr**
- Automatic closing of all buffered files at end of job when the program is compiled with the redirection switch (-R) on
- The formatted print facility **printf**, **fprintf** and **sprintf**
- The formatted input facility **scanf**, **fscanf** and **sscanf**
- Memory allocation using **malloc** and **free**

Differences

- Random I/O (via **seekr** and **tellr**) is done only at the system level
- System level I/O must be done in multiples of 128 bytes
- System level I/O cannot be redirected
- Redirection and **fopen** recognize the CP/M LST: device
- Newline characters are converted to carriage return/line feed (CR/LF) pairs for compatibility with CP/M text files
- CP/M EOF (^Z) is recognized as end-of-file

The last two differences are intended to allow the great majority of C programs which process text files to work the same under CP/M and UNIX. If you don't want any tampering done during file I/O, you can open the file for binary I/O.

5.2 Overview of the Library

Table 5-1 groups the functions according to use, and gives a one line summary for each one. All functions except **bdos**, **bdosl**, **mpm**, **setjmp**, **longjmp**, **in** and **out** are written in C (or in assembly language and C).

Console I/O

This group of functions is normally used when you want to communicate with the user at the console. Since these functions actually do their input and output using **stdin** and **stdout**, they can be redirected to disk files from the command line that runs the program. In this case the disk files will be automatically opened and closed, so you can do disk I/O with a minimum of effort by using these functions. Of course, you're limited to one input and one output file. However, in many cases this is all you need.

Character (Buffered) Disk I/O

This group of functions together with the previous group make up most of the Standard I/O Library. These functions are normally used to do disk I/O. However, if you tell these functions to use **stdin**, **stdout**, or **stderr** the console will be used unless you have redirected **stdin** or **stdout**.

These functions allow you to work with one character or line at a time. All buffer management is done for you. Buffer space is allocated when the file is opened, and buffers are automatically filled or emptied when necessary.

In addition, you have the option of opening files for normal (text) I/O or binary I/O. If a file is opened for text I/O, CR/LF pairs are changed to newline characters on input. The newline character is changed back to a CR/LF pair on output. This means that CP/M files created by a normal text editor can be manipulated by a Q/C program using the normal C convention where a line is terminated by a newline character. Also, the CP/M EOF character ^Z is recognized in input files and is added to the end of output files when they

are closed.

Files opened for binary I/O have absolutely no tampering done. On input, you get each character exactly as it was read. A CR/LF pair comes in as the two characters '\r', '\n'. The CP/M EOF character ^Z is returned as the decimal value 26. You will not get the EOF indication until there is no more data to read. Also, no sign extension is done. Thus, if the character 0xFF is read, it will be returned as the decimal value 255 rather than -1 (which is EOF). On output, each character is written exactly as given to the output function. When you close the file, a ^Z is not added.

Examples of uses for binary I/O are reading the input for a file dump routine and writing escape sequences to a printer to control special features such as graphics.

Files used by this group of functions are identified by a file pointer (fp) which is defined like this

```
FILE *fp;
```

FILE is defined in the file **QSTDIO.H** which should be included in all Q/C programs. When you open a file, you get back a file pointer for that file. All the other functions in this group are given this file pointer as an argument to identify the file.

Low-level (System) Disk I/O

These functions, which simulate UNIX system calls, give you access to the CP/M sequential and random disk I/O routines. You can read or write one or more CP/M 128-byte logical records with minimal overhead by using these functions. You must provide the buffer space to hold all the records being read or written, but the library functions will maintain the CP/M file control block (fcb) for each file.

These routines are used only in special cases, such as file copy programs, where efficiency is the most important consideration. You lose portability when you use these functions, and your program has to do its own buffer management.

Notice that I/O at this level can not be redirected as it can under UNIX. If you try to use a file descriptor of 0 (**stdin** under UNIX), 1 (**stdout**), or 2 (**stderr**) you will get an error return value.

Files used by the functions in this group are identified by a file descriptor (fd) which is simply a small integer. When you open a file, you get back a file descriptor for the file. When you call any of the other functions in this group, you give them the file descriptor as an argument to tell them which file to use.

CONSOLE INPUT/OUTPUT

getchar()	Read a character from the standard input file
getkey()	Check for keyboard input
gets()	Read a string from the standard input file
qprintf()	Short version of printf
printf()	Write formatted print to the standard output file
putchar()	Write a character to the standard output file
puts()	Write a string to the standard output file
scanf()	Read formatted data from the standard input file

CHARACTER (BUFFERED) INPUT/OUTPUT

clearerr()	Clear the error indicator for a file
fclose()	Close a file
feof()	Check whether end-of-file has been found
ferror()	Check whether an I/O error has occurred
fflush()	Flush an output file buffer
fgets()	Read a string from a file
fileno()	Get the internal file descriptor number
fopen()	Open a file
fprintf()	Write formatted output to a file
fputs()	Write a string to a file
fread()	Read data items from a file
fscanf()	Read formatted data from a file
fwrite()	Write data items to a file
getc()	Read a character from a file
getw()	Read a word from a file
putc()	Write a character to a file
putw()	Write a word to a file
setbsize()	Set the size of a user-supplied buffer
setbuf()	Provide a user-supplied buffer
ungetc()	Push a character back onto an input file

LOW-LEVEL (SYSTEM) INPUT/OUTPUT

close()	Close a file
creat()	Create a new file or reuse an existing file
open()	Open an existing file
read()	Read a file in multiples of 128 bytes
seekr()	Change read/write pointer
tellr()	Report read/write pointer
write()	Write a file in multiples of 128 bytes

CHARACTER TESTING

isalnum()	Is the character a letter or a number?
isalpha()	... a letter?
isascii()	... an ASCII character
iscntrl()	... a control character
isdigit()	... a number?
islower()	... a lower case letter?
isprint()	... printable?
ispunct()	... punctuation?
isspace()	... white space?
isupper()	... an upper case letter?

Table 5-1. Q/C Function Library

STRING AND CHARACTER HANDLING

atoi()	Convert a numeric string to an integer
chlower()	Convert an upper case letter to lower case
chupper()	Convert a lower case letter to upper case
index()	Find the first occurrence of a character in a string
itob()	Convert an integer to a string in various bases
rindex()	Find the last occurrence of a character in a string
sprintf()	Format a string
sscanf()	Get formatted data from a string
strcat()	Append one string to the end of another string
strcmp()	Compare two strings
strcpy()	Copy one string into another string
strlen()	Calculate the length of a string
strmov()	Copy one string into another string
strncat()	Append at most n characters from one string to another
strncmp()	Compare at most n characters in two strings
strncpy()	Copy exactly n characters from one string to another
tolower()	Convert a character to lower case
toupper()	Convert a character to upper case

SYSTEM

bdos()	Do a CP/M system call and return a double byte
bdosl()	Do a CP/M system call and return a single byte
exit()	Close the files and return to CP/M
in()	Input a byte from an 8080/Z80 port
out()	Output a byte to an 8080/Z80 port
makfcb()	Build a CP/M file control block
mpm()	Do an MP/M system call
unlink()	Delete a file from the CP/M directory

MEMORY ALLOCATION

calloc()	Allocate and zero array space
free()	Return space to memory allocator for reuse
malloc()	Allocate a block of memory space which can be freed
maxsbrk()	Report memory space available from **sbrk**
moat()	Set the size of the stack reserve space
sbrk()	Permanently allocate a block of memory space

MISCELLANEOUS

imax()	Find the maximum of two integers
imin()	Find the minimum of two integers
longjmp()	Do a non-local jump
peek()	Look at a byte of memory
poke()	Change a byte of memory
setjmp()	Set up for a non-local jump
wpeek()	Look at a word of memory
wpoke()	Change a word of memory

Table 5-1. Q/C Function Library

System Functions

This group of functions provides you with various capabilities for interacting directly with CP/M or MP/M and the hardware. You can make system calls, build file control blocks, delete files from disk and do I/O to the 8080/Z80 ports.

Memory Allocation

These functions provide a simple memory allocation scheme. Memory can be allocated by calling **malloc** or **calloc** and later returned for reuse by calling **free**. If you don't want the overhead associated with this scheme, you can get memory by calling **sbrk**. Your executable program will be smaller, but memory obtained this way cannot be freed for reuse.

All the space between the top of the program and the stack is available for allocation except for a buffer zone just below the stack. This buffer zone, called the moat, protects your program from having the stack grow down into it. This is not foolproof, of course, because the stack may grow larger than the moat. The moat is originally set to 1000 bytes, but this can be changed with the **moat** function.

5.3 Function Descriptions

The rest of this chapter describes each of the functions in the library in a standard format. The descriptions are arranged alphabetically by function name. Other than a few closely related functions, each function appears on a separate page. Only the sections that are needed appear in the descriptions.

The format used for each function description is shown on the next page.

Name
function name - one line description of the function

Synopsis
Defines the calling sequence, the arguments and indicates the type of the return value if it is something other than **int**.

Description
Describes what the function does.

Returns
Tells what the return values are. Symbolic constants (for example, NULL and EOF) are defined in the file **QSTDIO.H** which should be included in every program.

Example (optional)
Gives an example of the use of this function.

Remarks (optional)
Discusses any points of interest or potential problems in using this function.

Portability (optional)
Identifies those functions which are not in the Standard I/O Library. Also points out functions which are peculiar to this implementation for CP/M or which differ from their UNIX counterpart. Notice that if you use only those functions which are in the Standard I/O Library, you will minimize the effort of moving your program to another standard implementation of C.

See Also (optional)
Lists related functions which you might want to look at.

Bugs (optional)
Tells about serious problems with the way this function works.

Name
atoi - convert a numeric string to an integer

Synopsis
```
atoi(s)
char *s;
```

Description
Converts a string containing a signed decimal number in the range -32768 to +32767 to its integer equivalent. Leading white space will be skipped, and a + or - sign may precede the number. atoi will continue converting until a character other than '0' through '9' is found.

Returns
The value returned is the integer equivalent of the number in s.

Example
Several examples are

```
atoi("123")    returns  123
atoi("  -5")   returns  -5
atoi("12abc")  returns  12
atoi("abc")    returns  0
atoi("123456") returns  an undefined integer
```

Remarks
No check is made to determine that the number will actually fit in an integer.

See Also
itob

Name

bdos - do a CP/M system call and return a double byte value
bdosl - do a CP/M system call and return a single byte value

Synopsis

```
bdos(c, de)
int c, de;

bdosl(c, de)
int c, de;
```

Description

Do a CP/M bdos call to location 5H using **c** as the function number and **de** (when needed) as the argument. **c** is loaded in register C and **de** is loaded in the register pair DE.

Returns

bdos returns an integer which is the double byte value placed in register HL by CP/M. **bdosl** returns an integer whose low-order byte is the single byte value placed in register A by CP/M and whose high-order byte is zero.

Example

If you want to stop a C program immediately without any of the usual cleanup done by exit(), you can warm boot CP/M using system call zero by saying

```
bdos(0, 0);
```

Remarks

If you are using CP/M, the function **bdos** can be used for all calls to CP/M and the return value will be correct. If you are using a CP/M-compatible system, you should call the the appropriate function depending on whether you expect a single or double byte return value. See Appendix F for more information.

If you need to do disk I/O yourself for any reason, **makfcb** will build a CP/M file control block (fcb) for you.

Portability

These functions are not in the Standard I/O Library.

See Also

makfcb, mpm

Bugs

Because Q/C pushes the arguments to a function on the stack in the same order it finds them, you must always include the argument **de** even if it is not used (just say **bdos(1, 0)** for example). If you don't, **bdos** will not find the first argument and exciting things may happen. Always be CAUTIOUS when using this function.

Name
char *calloc - allocate and zero array space

Synopsis
```
calloc(nelem, elemsize)
int nelem, elemsize;
```

Description
Allocates memory space for an array of **nelem** elements where **elemsize** is the size of an element. The space allocated is initialized to zero.

Returns
A pointer to the beginning of the space allocated or NULL if there is not enough space available.

Example
Suppose you need a array of 100 structures and you don't want the space included in your .COM file. Instead define a pointer to the structure and get the space from **calloc**:

```
struct xyz *a_of_xyz;

if ((a_of_xyz = calloc(100, sizeof(struct xyz))) == NULL)
        printf("Can't allocate a_of_xyz\n");
```

Remarks
If you store outside the space given to you by **calloc** you can cause all sorts of serious errors including crashing CP/M. Also, you can cause very mysterious errors if you don't check to see that you really got the space you requested. Since **calloc** returns NULL which is zero, your pointer to the new memory will contain zero. Then when you start storing into your block of memory you will actually be writing over CP/M information in low memory.

See Also
free, malloc

Name
close - close a file

Synopsis
```
close(fd)
int fd;
```

Description
Closes the file associated with file descriptor **fd**. This frees the file descriptor for use by another file. If the file is open for output and no errors have occurred while writing the file, a CP/M close will be done to record the file permanently in the file directory.

Returns
The return value is 0 if the close is successful or -1 if the file is not open or CP/M can't close it.

Example
Suppose the name of the output file is in the array **outfile**, and that **outfd** contains the file descriptor given to this file when it was opened. Then you can close the file and test for a bad close like this

```
char outfile[15];
int outfd;
...
if (close(outfd) == -1) {
        printf("Can't close: %s\n", outfile);
        exit(1);
        }
```

Portability
This function is not in the Standard I/O Library. It simulates a UNIX system call.

See Also
open, creat

Name

chlower - convert an upper case letter to lower case
chupper - convert a lower case letter to upper case

Synopsis

```
chlower(c)
int c;

chupper(c)
int c;
```

Description

If **c** is an upper case letter 'A' through 'Z', **chlower** will convert it to its lower case form 'a' through 'z'.

If **c** is a lower case letter 'a' through 'z', **chupper** will convert it to its upper case form 'A' through 'Z'.

Returns

chlower returns the lower case form of **c** when **c** is a letter and **c** otherwise.

chupper returns the upper case form of **c** when **c** is a letter and **c** otherwise.

Example

Say you have the file name "copy.c" in an array pointed to by **pfile** and you want to convert it to upper case. The following piece of code

```
for( ; *pfile; ++pfile)
        *pfile = chupper(*pfile);
```

will convert it to "COPY.C". The period in the middle of the name is not affected.

Remarks

These functions were added to the Q/C Library because the Standard I/O Library functions **tolower** and **toupper** will change some non-alphabetic characters as well as the ones desired.

Portability

These functions are not in the Standard I/O Library.

See Also

tolower, toupper

Name
clearerr - clear the error indicator for a file

Synopsis
```
clearerr(fp)
FILE *fp;
```

Description
Clears the error flag for the file pointed to by **fp**.

Returns
No return value.

Remarks
Normally, if an error occurs the error flag remains set until the file is closed and all further calls to the I/O functions will be ignored. If there is some I/O which must be done, you can reset the error flag with this function.

See Also
ferror

Name

creat - create a new file or reuse an existing file

Synopsis

```
creat(filename, pmode)
char *filename;
int pmode;
```

Description

Opens the file named by **filename** for use. If the file does not exist, it is created. If it does exist, it is deleted and then recreated which effectively discards the previous contents of the file. **pmode** is the protection mode for a new file under UNIX. It has no meaning in Q/C and is included only for compatibility. Your call will normally be **creat(filename, 0644)**. If your program is run under UNIX, the **pmode** of **0644** says that you can read or write this file, but everyone else can only read it.

Returns

A file descriptor (**int fd**) if successful or -1 if not.

Example

To open the file named "PROGRAM.C" for output (or reuse the file if it already exists) say

```
int outfd;
...
outfd = creat("PROGRAM.C", 0644);
```

outfd is the file descriptor which will tell **write** and **close** which file to work with.

Portability

This function is not in the Standard I/O Library. It simulates a UNIX system call.

See Also

open, close

Name

exit - close all open files and return to CP/M
_exit - return immediately to CP/M

Synopsis

```
exit(n)
int n;

_exit()
```

Description

exit closes any buffered files which are open for output and reboots CP/M. If **n** is not zero, this is considered to be an error termination. In this case, **exit** will also delete the CP/M submit file A:$$$.SUB if present. This will terminate the submit stream which ran this program.

_exit does a CP/M warm boot by calling bdos function zero. No other action is taken.

Returns

Does not return.

Example

The **main** procedure in Q/C ends with the statement

```
exit(nerrors);
```

where **nerrors** is the number of errors the compiler found in your C program. If you are using a submit file to compile and assemble your program, it will be deleted if the compile is unsuccessful. This prevents the assembler from trying to assemble a bad file.

Portability

These functions are not in the Standard I/O Library. They simulate UNIX system calls.

Name
fclose - close a file

Synopsis
```
fclose(fp)
FILE *fp;
```

Description
Closes the file pointed to by fp thus freeing the file pointer for use by another file. If the file is open for output, a CP/M EOF (^Z) will be added, any characters in the buffer will be written, and a CP/M close will be done to make the file permanent. If the file was open for binary output, the CP/M EOF character will not be added. If the last 128 byte sector is not filled, whatever is currently in the buffer will be written on the file.

Returns
Returns EOF if the close is not successful. The return value for a successful close is not specified, so you should only check for EOF.

Example
To close the file pointed to by outfp say

```
fclose(outfp);
```

See Also
fopen, fflush

Name
feof - check whether end-of-file has been found

Synopsis
```
feof(fp)
FILE *fp;
```

Description
Checks whether end-of-file has been read on the file pointed to by **fp**.

Returns
Returns non-zero if end-of-file has been read and zero otherwise.

Example
To check whether end-of-file has been read on the file pointed to by **input** say

```
FILE *input;
...
if (feof(input)) {
        /* do end-of-file processing */
        }
```

Remarks
The buffered I/O functions all return **EOF** to indicate an error condition. This means that input functions such as **getc** return the same value for an error and for end-of-file. You can determine which condition actually occurred by calling **feof** and **ferror**.

See Also
ferror

Name
ferror - check whether an I/O error has occurred

Synopsis
```
ferror(fp)
FILE *fp;
```

Description
Checks whether an error has occurred on the file pointed to by **fp**.

Returns
Returns non-zero if an error has occurred and zero otherwise.

Example
To check whether an error has occurred on the file pointed to by **output** say

```
FILE *output;
...
if (ferror(output)) {
        /* do error processing */
        }
```

Remarks
The error condition can be set for a number of reasons besides actual hardware problems. For example, if the file was not opened successfully, or if an attempt was made to write an input file or read an output file, the error flag will be set.

See Also
feof, fopen

Name
fflush - flush an output file buffer

Synopsis
```
fflush(fp)
FILE *fp;
```

Description
If the file pointed to by fp is open for output, any data in the buffer will be written. If the last 128 byte sector is not full, whatever is already in the buffer is written. If the file is not open for output, this is considered an error and nothing will be written.

Returns
Returns EOF if an error occurs.

Example
To write the contents of the current buffer for the file pointed to by **outfp** say

```
fflush(outfp);
```

See Also
fopen, fclose

Name

fgets - read a string from a file

Synopsis

```
char *fgets(s, n, fp)
char *s;
int n;
FILE *fp;
```

Description

Reads a line from the file pointed to by **fp** into the string **s**. A maximum of **n - 1** characters will be moved. Any characters remaining in the line will be available for the next call. If the file is not open for binary I/O, a CR/LF pair is converted to a newline character '\n'.

When the file is open for binary I/O, end-of-file is reported only at the physical end of the file (no more sectors on the disk to read). If the file is open for normal I/O, reading a CP/M EOF (^Z) will also be considered end-of-file. When end-of-file is detected part way through a line read from a disk file, the partial line will be returned. The next call to **fgets** will report the end-of-file.

If the file is not open for input, it is considered an error and nothing will be put in **s**.

Returns

The value returned is **s** or NULL upon end-of-file or error.

Example

If you want to read lines from either the console or a disk file by using redirection use the following code

```
#define MAXLINE 81              /* allow for '\0' */
char buffer[MAXLINE];
...
while (fgets(buffer, MAXLINE, stdin) != NULL) {
        /* do something */
        }
```

Remarks

If **stdin** is being read from the console, CP/M function 10 (Read Console Buffer) is used. This means that all the usual CP/M editing characters will work. However, the only time end-of-file will be recognized is if a CP/M EOF (^Z) is the first character typed on the line.

See Also

gets

Name
fileno - get the internal file descriptor number

Synopsis
```
fileno(fp)
FILE *fp;
```

Description
Gets the file descriptor number which the buffered I/O functions use when they call the system I/O functions to do the actual I/O. This is the number returned by the system I/O function **open**.

Returns
The internal file descriptor number.

Remarks
This function is in the Standard I/O Library and is included more for completeness rather than any great usefulness.

See Also
open

Name

fopen - open a file

Synopsis

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

Description

Opens the file given by **filename** for buffered I/O. **filename** is a CP/M file name which may include a drive (for example B:INPUT.DAT) or you can specify "lst:" to write to the printer. **mode** is "r" to read, "w" to write, or "a" to append to the file. "a" will create the file if it does not exist. Mode specifications may be in upper or lower case.

Normally the file will be treated as CP/M text meaning newline characters will be converted to carriage return/line feed (CR/LF) pairs, and ^Z will be treated as end-of-file. For a complete description of the tampering being done, see **getc**, **putc**, and **fclose**. If you want to do I/O without any tampering, open the file for binary I/O by specifying a mode of "rb", "wb" or "ab".

Opening a text file in append mode ("a") positions the file at the CP/M end-of-file (^Z) so the first character written overwrites the ^Z. Opening a binary file ("ab") positions the file at the beginning of the next CP/M sector following the last sector already written.

The open will fail if:

(1) **filename** is not a valid CP/M file name or "lst:" or if **filename** is opened for reading and can't be found.
(2) **mode** is not "r", "w", "a", "rb", "wb" or "ab".
(3) The maximum number of buffered files is already open.

Returns

The value returned is a file pointer (**FILE *fp**) for a successful open or NULL if any errors are found.

Example

To open the file "STDSUB.LIB" for input and check the result say

```
FILE *libfp;
if ((libfp = fopen("STDSUB.LIB", "r")) == NULL)
        /* print error message */
```

Remarks

You can supply your own buffer by calling **setbuf**. If you don't, buffer space is obtained by calling **sbrk** the first time the the file is read or written. This means that you only use buffer space for the files which are open. Q/C comes with the library set up to support 10 files open at once, but there is very little space penalty for increasing this. See the appendix "Maintaining the Function Library" to change this limit.

See Also

fclose, setbuf, setbsize

Name
fprintf - write formatted output to a file

Synopsis
```
fprintf(fp, format, arg1, arg2, ...)
FILE *fp;
char *format;
```

Description
The arguments **arg1**, **arg2**, ... are formatted according to the specifications given in **format** and written to the file with file pointer **fp**. This works as described in Kernighan & Ritchie with the exception that none of the specifications for **long** and **float** are available.

If the file is not open for output, nothing will be written.

Returns
No return value.

Example
The function **cantopen** which reports the name of a file which can't be opened and then terminates the run can be written like this

```
cantopen(filename)
char *filename;
        {
        fprintf(stderr, "Can't open: %s\n", filename);
        exit(0);
        }
```

Remarks
As the example above shows, this function can be used to print error messages which must appear on the console. Even if **stdout** is redirected to a file, **stderr** will always be the console.

See Also
printf, qprintf, sprintf, puts, fputs

Name

fputs - write a string to a file

Synopsis

```
fputs(s, fp)
char *s;
FILE *fp;
```

Description

Writes the string **s** to the file pointed to by **fp**. If the file is open for normal (text) I/O, a newline '\n' will be expanded to a CR/LF pair to meet the CP/M text convention. If it is open for binary I/O, every character in **s** will be written exactly as received.

If the file is not open for output, it is an error and nothing will be written.

Returns

No return value.

Example

You can write the text line "This sentence no verb.\n" to a file in either of these two ways

```
fputs("This sentence no verb.\n", outfile);
```

or

```
fputs("This sentence", outfile);
fputs(" no", outfile);
fputs(" verb.\n", outfile);
```

Remarks

Notice if you do not end a line with a newline character, nothing will be appended so you can build up a line with several calls to **fputs**.

See Also

puts

Name
fread - read data items from a file

Synopsis
```
fread(ptr, size, nitems, fp)
int nitems
FILE *fp;
```

Description
Reads **nitems** of data each with length **size** from the file associated with file pointer **fp**. and places them in the area pointed to by **ptr**. If the file is open for text I/O the CR/LF pairs are changed to newline characters and end-of-file is recognized when a CP/M EOF (^Z) is encountered. If the file is opened for binary I/O you get everything unchanged.

If the file is not open for input, it is an error and nothing is read.

Returns
The return value is the number of items read or zero at end-of-file or if an error occurs.

Remarks
Usually, **fread** is used to read data written by **fwrite**. No conversion is done on input. This means, for example, that **int** data must be recorded as two byte binary integers on the file. If you want to read text files (ASCII data) and do conversions you should use **fscanf**. **fgets** should be used if you want to read text lines which end with carriage return/line feed pairs.

See Also
fgets, fscanf, fwrite, read

Bugs
Since CP/M does not record the exact end-of-file, files opened for binary I/O will not get an end-of-file indication until the end of the last sector is read. This means your program will need a special data item which it can recognize as end-of-file as the last item in the data file.

Name
free - return space to the memory allocator for re-use

Synopsis
```
free(p)
char *p;
```

Description
The space pointed to by p is returned to the pool of free memory which can be allocated by the functions **calloc** and **malloc**. p must point to an area which was originally obtained from **calloc** or **malloc** or the integrity of the entire system is threatened.

Returns
No return value.

Example
Suppose you have obtained space for a table from **malloc**. When the space is no longer needed it can be returned for some other use like this:

```
char *table;            /* pointer to the table */
...
table = malloc(SIZE);   /* get the space for table */
...
free(table);            /* space no longer needed */
```

See Also
calloc, malloc

Name
fscanf - read formatted data from a file

Synopsis
```
fscanf(fp, format, arg1, arg2, ...)
FILE *fp;
char *format;
```

Description
Reads characters from the file pointed to by **fp**, interprets them according to the specifications given in **format**, and stores them in the arguments **arg1**, **arg2**, etc. This works as described in Kernighan & Ritchie with the following exceptions:

(1) None of the specifications for **long** and **float** are available.

(2) The UNIX V7 use of the "h" conversion is followed. This treats "h" as a modifier of other integer conversions, so the valid uses are **%hd**, **%ho**, and **%hx**.

(3) Also, the UNIX V7 use of the "c" conversion is followed. This means that %c puts a single character in the matching argument which must be a pointer to character. However, **%nc** puts **n** characters into the matching argument which must be a pointer to an array of characters.

Returns
The value returned is the number of format items successfully matched and assigned to the arguments. EOF is returned upon end-of-file or error.

Example
You can read a person's name from a file like this:

```
char firstname[10], initial[2], lastname[20];

fscanf(fp, "%s %1s. %s", firstname, initial, lastname);
```

With the input "Brian W. Kernighan", **firstname** will contain "Brian", **initial** will contain "W", and **lastname** will contain "Kernighan".

Remarks
The arguments to **fscanf** must be pointers or you will not get back the values assigned. If no items are matched, the count returned is zero which is different from EOF. If end-of-file is found in the middle of the format, the return value is the number of items matched to this point. The next call to **fscanf** will return EOF.

See Also
fgets, gets, scanf, sscanf

Bugs
Doesn't implement the UNIX V7 %[...] conversion.

Name

fwrite - write data items to a file

Synopsis

```
fwrite(ptr, size, nitems, fp)
int nitems
FILE *fp;
```

Description

Writes **nitems** of data each with length **size** from the area pointed to by **ptr** to the file associated with file pointer **fp**. If the file is open for text I/O the newline characters are changed to CR/LF pairs and a CP/M EOF (^Z) is added when the file is closed. If the file is opened for binary I/O everything is written unchanged.

If the file is not open for output, it is an error and nothing is written.

Returns

The return value is the number of items actually written or zero if an error occurs.

Remarks

fwrite writes the data with no conversion so **int** data is recorded as binary integers on the output file. If you want to write text files (ASCII data) and do conversions you should use **fprintf**. **fputs** should be used if you want to write text lines which end with carriage return/line feed pairs.

See Also

fputs, fread, fscanf, write

Bugs

Since CP/M does not record the exact end-of-file, files opened for binary I/O will have garbage in the last CP/M sector after the final data item. This means your program will need to write a special item after the last data item which can be recognized as end-of-file when the file is read back in.

Name
getc - read a character from a file

Synopsis
```
getc(fp)
FILE *fp;
```

Description
Reads the next character from the file pointed to by **fp**. CR/LF pairs are changed to the newline character '\n' and CP/M EOF (^Z) is recognized as end-of-file when the file was opened for normal (text) input. If the file is open for binary input, all characters are returned just as they are read. They are not sign-extended, so the character with value 0xFF can be distinguished from EOF. End-of-file will be reported only when there is no more data to read.

If the file is not open for input, it is an error and nothing is read.

Returns
The return value is the character read or EOF.

Example
To read a single character from the file pointed to by **fp** and test for end-of-file say

```
int c
...
if ((c = getc(fp)) != EOF)
        /* do something */
```

Notice that c should be defined as **int** to be sure that the special return value EOF can be distinguished from any legitimate character which might be read from the file.

Remarks
Once end-of-file has been reached or an error has occurred, all subsequent calls will return EOF. Notice that if you do not supply your own file buffer by calling **setbuf**, the first call to **getc** will cause it to get buffer space from the memory allocator **sbrk**. If no space is available, **getc** will return EOF to indicate the error. Since EOF can indicate reading end-of-file or an error, you must use the library functions **feof** and **ferror** to distinguish which meaning is intended.

See Also
feof, ferror, getchar

Name
getchar - read a character from the standard input file

Synopsis
getchar()

Description
Reads the next character from the standard input file **stdin**.

Returns
The value returned is the character read or EOF.

Example
A classic example of the use of **getchar** from Kernighan & Ritchie is

```
int c;
while ((c = getchar()) != EOF)
        putchar(c);
```

This copies the standard input file to the standard output file one character at a time until end-of-file is reached.

Remarks
EOF is returned when a CP/M end-of-file (^Z which is '\32') is read. If **stdin** has been redirected to a disk file, EOF will also be returned when the physical end of a disk file is reached.

If **stdin** is assigned to the console (which is the default), a ^C typed as the first character on the line will warm boot CP/M.

When choosing between **getchar** and **getc** notice that **getchar** lets you read a disk file without declaring a file pointer or explicitly opening the file. You simply redirect the standard input file **stdin** on the command line when you run the program. On the other hand, you can only specify one input file this way.

See Also
feof, ferror, getc, getkey

Name
getkey - check for keyboard input

Synopsis
getkey()

Description
Checks to see if a console key has been pressed and gets the character if one is available.

Returns
The character typed or EOF if no character is available.

Example
This function provides a good way to get a different seed value for a random number generator every time a program is run. Simply ask a question and then increment the seed until the user responds.

```
int seed, answer;
seed = 1;
printf("Do you want instructions (y or n)?");
while ((answer = getkey()) == EOF)
        ++seed;
```

The while loop will check to see if a key was pressed and store the result in **answer**. As long as there is no response, **getkey** will return EOF and **seed** will be incremented. When a key is pressed, the loop will end. **answer** will have the response, and **seed** will be set to some random value.

Remarks
The advantage of this function over other console input functions such as **getchar** is that you get control back if no character has been typed. Any other function will wait until a character is ready. Notice that **getkey** cannot be redirected.

Portability
This function is not part of the Standard I/O Library.

See Also
getchar, getc

Name

gets - read a string from the standard input file

Synopsis

```
char *gets(s)
char *s;
```

Description

Reads a line from **stdin** into the string **s**. A carriage return/line feed (CR/LF) combination is considered the end of the line. The CR/LF is discarded and a null character '\0' is appended to conform to C conventions.

Returns

The value returned is **s** or NULL upon end-of-file or error.

Remarks

If **stdin** is assigned to the console, the CP/M Read Console Buffer function (#10) is used. This means that all of the usual CP/M editing characters will work just as if you were typing a line to CP/M. In this case, the only way that an end-of-file will be recognized is if the first character typed on the line is a ^Z.

See Also

fgets

Bugs

Since you can't specify the maximum length of the string to be read, it is possible to overwrite whatever follows **s**. **gets** has no way of knowing how big **s** was defined so it just stuffs in everything up to the CR/LF. If there is a possibility of reading a line which is too long, you can use **fgets** to read **stdin** like this

```
fgets(s, MAXSIZE, stdin);
```

and limit the number of characters read to MAXSIZE - 1.

Name
getw - read a word from a file

Synopsis
```
getw(fp)
FILE *fp;
```

Description
Reads the next word from the file pointed to by **fp**. The word is built up by two successive calls to **getc** so the same actions described there for text vs. binary files apply. The word is read low byte first and then high byte.

If the file is not open for input, it is an error and nothing is read.

Returns
The return value is the word read or **EOF** on end-of-file or error. Since **EOF** is just an integer value which may be returned normally you must call **feof** and **ferror** in this case to determine what actually happened.

Example
To read a single word from the file pointed to by **fp** and test for end-of-file say

```
int word;
...
if ((word = getw(fp)) == EOF && feof(fp))
        printf("End of file reached\n");
```

See Also
putw

Name
in - input a byte from an 8080/Z80 port

Synopsis
```
in(port)
int port;
```

Description
Gets a byte from the port indicated by port.

Returns
The return value is the byte read zero-extended to convert it to an integer.

Example
If port 32 is connected to some interesting device you can get the value currently available by saying:

```
#define DEVICE 32
unsigned value;

value = in(DEVICE);
```

Remarks
This function does not use any self-modifying code so it can be used in a program which is placed in ROM. The Z80 version uses the Z80 instruction "IN A,(C)" so the port number is loaded in register C. The 8080 version builds the instruction "IN port" on the stack so that "port" does not need to be put into the function's code.

Portability
This function is not part of the Standard I/O Library. It is included only to provide a C interface to the 8080/Z80 hardware.

See Also
out

Name
imax - find the maximum of two numbers
imin - find the minimum of two numbers

Synopsis
```
imax(m, n)
int m, n;

imin(m, n)
int m, n;
```

Description
imax/imin determines the maximum/minimum of the two numbers m and n by doing a signed comparison.

Returns
The value returned by imax/imin is the larger/smaller of m and n.

Example
If a contains -20 and b contains 15 then the statement

```
c = imax(a, b);
```

will set c to 15.

Portability
These functions are not in the Standard I/O Library.

Name
index - look for a given character in a string

Synopsis
```
char *index(s, c)
char *s, c;
```

Description
Looks for the first occurrence of the character c in the string s.

Returns
The value returned is a pointer to the first occurrence of c in s or NULL if c is not found in s.

Example
Programs which read an input file and produce an output file often use the input file name with a different file extension as the output file name. A text formatter reading filename.TXT as the input might use filename.PRN as the output when no output file name is specified. The following code fragment copies the input file name to the output file name and looks for a period in the name. If it finds one, it copies PRN after the period. Otherwise, it adds .PRN (with the period) to the end of the name.

```
char infile[15], outfile[15], *p;
...
strcpy(outfile, infile);          /* start with input filename */
if ((p = index(outfile, '.')) != NULL) /* look for period */
    strcpy(p + 1, "PRN");         /* add "TXT" after period */
else                              /* no period found */
    strcat(outfile, ".PRN");
```

See also
rindex

Name

isalnum - is the character a letter or a number?
isalpha - ... a letter?
isascii - ... an ASCII character
iscntrl - ... a control character
isdigit - ... a number?
islower - ... a lower case letter?
isprint - ... printable?
ispunct - ... punctuation?
isspace - ... white space?
isupper - ... an upper case letter?

Synopsis

```
is____(c)
int c;
```

Description

Each of these routines is called with an integer value and a test is done to see if it belongs to the specified class. **isascii** is defined for all integers, but the other tests are defined only if **isascii** is true, or for the special value EOF.

isalnum - Is the character a letter 'a' through 'z' or 'A' through 'Z', or a number '0' through '9'?

isalpha - Is the character a letter 'a' through 'z' or 'A' through 'Z'?

isascii - Is the character an ASCII character, which means the numeric value of the character is in the range '\0' through '\177'?

iscntrl - Is the character a control character, which means its numeric value is in the range '\0' through '\37' or is '\177'? (CP/M would call these ^@ through ^_ and DEL or RUBOUT.)

isdigit - Is the character a number '0' through '9'?

islower - Is the character a lower case letter 'a' through 'z'?

isprint - Is the character printable? This means will you see it if you try to print it. A blank is considered printable.

ispunct - Is the character punctuation? This means is it one of the characters !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~

isspace - Is the character white space, which means one of the characters blank (' '), newline ('\n'), carriage return ('\r'), formfeed ('\f'), or tab ('\t')?

isupper - Is the character an upper case letter 'A' through 'Z'?

Returns

Non-zero if true or zero if false.

Example

To test whether the string **symname** contains a valid Q/C name (meaning the first character is alphabetic or an underscore) say

```
if (isalpha(symname[0]) || symname[0] == '_')
```

Name
itob - convert an integer to a string in various bases

Synopsis
```
char *itob(n, s, b)
int n, b;
char *s;
```

Description
Converts the integer **n** to its ASCII representation in base **b** and places it in string **s**. **n** can be signed or unsigned. The base can be -10 for signed decimal or any value from 2 to 36 for unsigned numbers. For bases greater than 10, capital letters starting with A are used to represent digits greater than 10.

Returns
The value returned is a pointer to **s**.

Example
Some examples of using **itob** are:

```
itob(6, s, 2)       puts  "110"    in s
itob(6, s, 10)      puts  "6"      in s
itob(35, s, 8)      puts  "43"     in s
itob(35, s, 36)     puts  "Z"      in s
itob(65535, s, 10)  puts  "65535"  in s
itob(65535, s, -10) puts  "-1"     in s
itob(65535, s, 16)  puts  "FFFF"   in s
```

To print the signed decimal representation of **n** on **stdout** say

```
puts(itob(n, s, -10));
```

Remarks
The string **s** must be long enough to hold the longest number you expect plus one position for the end-of-string character '\0'. If you specify signed decimal (b == -10), be sure to allow one additional position for a negative sign. For the usual cases (bases 8, 10, and 16), you should use **sprintf()**, as it is portable.

Portability
itob is not in the Standard I/O Library.

See also
atoi, sprintf

Name
longjmp - do a non-local jump

Synopsis
```
#include <setjmp.h>
longjmp(env, val)
jmp_buf env;
int val;
```

Description
longjmp causes a jump to the location where the function **setjmp** was last called with the argument **env**. The previous environment which is saved in **env** is restored and the integer **val** is sent to **setjmp** as its return value.

Returns
This function does not return. Control is transferred to the last call to **setjmp** which returns **val**.

Example
Say you are deep in your full-screen editor program and you find you are out of memory. You want to return to the top of program, report the error and quit.

```
#define MEM_ERR 1
#include <setjmp.h>
jmp_buf hold_env;
...
deepfunc() {
        ...
        if ((p=malloc(ENOUGH)) == NULL)
                longjmp(hold_env, MEM_ERR);
```

Remarks
You must have called **setjmp** with the same **jmp_buf** variable before you call **longjmp**. Otherwise, **longjmp** may go anywhere and you will probably crash your system.

Portability
This function is not considered part of the Standard I/O Library. However, it is identical to the library function normally available to C programs under UNIX.

Name
makfcb - build a CP/M file control block (fcb)

Synopsis
```
makfcb(filename, fcb)
char *filename, *fcb;
```

Description
Builds a CP/M file control block (fcb) for the file **filename** which can be any valid CP/M file name. The filename is converted to upper case. Explicit drive names and the CP/M wild card characters '*' and '?' are allowed. **fcb** must be a least 36 bytes long to accommodate CP/M 2.X random access functions. The first 12 bytes will contain the drive code and file name, and the remaining 24 bytes will be zeroed. **makfcb** considers the following conditions to be errors:

(1) drive code not alphabetic, e.g. 1:PROGRAM.C

(2) file name portion longer than 8 characters, e.g. DISKLIBRARY.C

(3) file type longer than 3 characters, e.g. INPUT.DATA

(4) control characters in the file name, e.g. PR^OGRAM.C

Returns
Returns -1 if an error is found.

Example
To build an fcb that matches any C program file on the currently logged drive, say

```
char fcb[36];
...
makfcb("*.c", fcb);
```

The name portion of the **fcb** (**fcb[1]** through **fcb[11]**) will contain the characters ????????Cbb where "b" indicates a blank.

Remarks
This function is typically used when you make a CP/M system call using **bdos** and the second argument is an fcb.

Portability
This function is not in the Standard I/O Library.

Name

malloc - allocate a block of memory space which can be freed

Synopsis

```
char *malloc(n)
unsigned n;
```

Description

Allocates a block of **n** bytes from the space available between the top of the program and the bottom of the stack. A certain amount of space (called the moat) which is directly below the stack cannot be allocated. This is intended to leave space for stack growth so the stack does not grow down into the program memory space.

Returns

The value returned is a pointer to the beginning of the space allocated or NULL if there is not enough free space available.

Example

To allocate a table of 1000 bytes and make **ptable** point to the beginning of the space say

```
if ((ptable = malloc(1000)) == NULL)
        /* report not enough space available */
```

You should always check the return value to see that you <u>really</u> got the space.

Remarks

The size of the moat is initially set to 1000 bytes, but you can change this with the library function **moat**.

If you store outside the space given to you by **malloc** you can cause all sorts of serious errors including crashing CP/M. Also, you can cause very mysterious errors if you don't check to see that you really got the space you requested. Since **malloc** returns NULL which is zero, your pointer to the new memory will contain zero. Then when you start storing into your block of memory you will actually be writing over CP/M information in low memory.

See also

calloc, free, maxsbrk, moat

Name
maxsbrk - report memory space available from sbrk

Synopsis
unsigned maxsbrk()

Description
Determines how much memory space is available for sbrk allocate.

Returns
The value returned is the amount of memory between the top of the memory allocated and the bottom of the moat.

Example
If you want to get all the memory available for a large buffer say:

```
bufsize = maxsbrk();
```

Since the stack pointer bounces up and down, you should probably reduce this number by a few bytes (say 10 or 20) to be sure you can really get this much when you actually call sbrk to get the space.

Portability
This function is not in the Standard I/O Library.

See also
malloc, moat

Name
moat - set the size of the stack reserve space

Synopsis
```
unsigned moat(size)
unsigned size
```

Description
The moat is the amount of free memory space directly below the current stack which cannot be allocated to allow for stack growth. This value is initially set to 1000 bytes, but **moat** lets you change this.

Returns
The value returned is the old moat size.

Example
To change the moat size to 500 bytes say

```
moat(500);
```

Remarks
Normally, you should not set the moat size to less than 500 bytes. All local variables except **register** variables and all function arguments are pushed onto the stack, so the stack can grow quickly. Local arrays will make the stack grow very quickly!

Portability
This function is not in the Standard I/O Library.

See also
malloc, maxsbrk

Name
mpm - do an MP/M system call

Synopsis
```
mpm(c, de, a)
int c, de, *a
```

Description
Do an MP/M system call to location 5H using c as the function number and de as the argument. c is loaded in register C and de is loaded in the register pair DE.

Returns
Returns an integer which is the double byte value placed in register HL by MP/M. Also returns the value MP/M places in register A in the integer pointed to by a. This integer will contain the contents of register A in the low-order byte and zero in the high-order byte.

Example
To call MP/M and get both the value in register pair HL and the value in register A say

```
int a, c, de, hl;

hl = mpm(c, de, &a);
```

Portability
This function is not in the Standard I/O Library.

See Also
bdos, bdosl

Bugs
You must always include the argument de even if it is not used by MP/M. You can just pass a constant zero like this

```
mpm(c, 0, &a);
```

Name
open - open an existing file

Synopsis
```
open(filename, rwmode)
char *filename;
int rwmode;
```

Description
Opens the file named by **filename**. **rwmode** can be:

0 = read
1 = write
2 = read and write

It is an error to **open** a file which does not exist. An error will also occur if **filename** or **rwmode** is invalid, if CP/M cannot open the file, or if the maximum number of files is already open.

Returns
The value returned is a file descriptor (**int fd**) or -1 if an error was detected.

Remarks
This function is used to open files for system I/O using the **read** and **write** functions. The maximum number of files allowed to be open at one time is 10. This can be increased by recompiling the disk library functions. See the Appendix E "Maintaining the Function Library".

rwmode = 2 indicating that the file can be read or written is normally used for a file which will be accessed randomly and positioned using the function **seekr**. Initially the file is positioned at the first record in the file which is CP/M random record number 0.

Portability
This function is not in the Standard I/O Library. It simulates a UNIX system call.

See also
creat, close

Name
out - output a byte to an 8080/Z80 port

Synopsis
```
out(value, port)
int value, port;
```

Description
Writes the low-order byte of value to the port indicated by port.

Returns
The return value is the byte written zero-extended to convert it to an integer.

Example
To send the value 0x80 to port 19 say:

```
out(0x80, 19);
```

Remarks
This function does not use any self-modifying code so it can be used in a program which is placed in ROM. The Z80 version uses the Z80 instruction "OUT A,(C)" so the port number is loaded in register C. The 8080 version builds the instruction "OUT port" on the stack so that "port" does not need to be put into the function's code.

Portability
This function is not part of the Standard I/O Library. It is included only to provide a C interface to the 8080/Z80 hardware.

See Also
in

Name
qprintf - short version of printf

Synopsis
```
qprintf(format, arg1, arg2, ...)
char *format;
```

Description
The arguments **arg1**, **arg2**, ... are formatted according to the specifications given in **format** and written to **stdout**. This works as described for the function **printf** except that the only conversion specifications allowed are **%d** (decimal), **%u** (unsigned decimal), **%o** (octal), **%x** (hexadecimal), **%c** (character), and **%s** (string). The width, precision, justification, and padding specifications available in **printf** are missing here.

Returns
There is no return value.

Example
qprintf can print an error message on the console like this:

```
char filename[15];

qprintf("Can't open: %s\n", filename);
```

If filename contains "INPUT.DATA" the message on the console will be:

```
Can't open: INPUT.DATA
```

Remarks
This function is useful when you are doing simple formatted output to the console and you need to conserve the amount of memory used.

Notice that **qprintf** is a proper subset of **printf**. This means that if you use **qprintf** in a program, you can simply change all references to **printf** and your program will still work correctly.

Portability
This function is not in the Standard I/O Library. It is provided only to aid in writing compact programs to fit in the 64K address space provided on 8080/Z80 machines. You should use **printf** in any program where portability is required.

See Also
printf

Name

peek - look at a byte of memory
poke - change a byte of memory

Synopsis

```
peek(address)
unsigned address;

poke(address, value)
unsigned address;
int value;
```

Description

peek lets you look at the contents of a byte in memory.
poke changes the value stored in a byte of memory.

Returns

The value returned by **peek** is the value of the byte at **address**. **poke** returns the value that was previously stored in the byte being changed. The return value is not sign-extended in either case. The high-order byte will always be zero.

Example

To see what is in the CP/M IOBYTE at location 3H say

```
char iobyte;
iobyte = peek(3);
```

To change it so that the LIST device is the line printer (LPT:) regardless of what it previously was say

```
poke(3, ((iobyte & 0x3F) | 0x80))
```

The left-most two bits of iobyte are forced to binary 10 (LPT:). This value is then put in location 3H.

Remarks

These functions are useful to work with one or at most a few bytes in memory. If you want to do more than this, you should define a character pointer and do indirection on it. For example, the **peek** example above would become

```
char *p, iobyte;
p = 0x3;        /* make p point to location 3H */
iobyte = *p;    /* look at the contents of 3H */
```

Portability

These functions are not in the Standard I/O Library.

See Also

wpeek, wpoke

Name
printf - write formatted print to the standard output file

Synopsis
```
printf(format, arg1, arg2, ...)
char *format;
```

Description
The arguments **arg1**, **arg2**, ... are formatted according to the specifications given in **format** and written to **stdout**. This works as described in Kernighan & Ritchie with the exception that **long** and **float** formatting is not available. Thus the conversion characters recognized are **d** (decimal), **u** (unsigned decimal), **o** (octal), **x** (hexadecimal), **c** (character), and **s** (string). The width, precision, justification, and padding specifications work just as described in the book.

Returns
There is no return value.

Example
printf can print a simple string like this:

```
printf("Nothing fancy here\n");
```

prints

```
Nothing fancy here
```

In this case the format is "Nothing fancy here\n". Since it has no conversion specifications, no other arguments are expected or will be used.

Normally the format will have conversion specifications. For example, you can see the internal value of a character like this:

```
printf("The ASCII code for %c is %d.\n", 'A', 'A');
```

prints

```
The ASCII code for 'A' is 65.
```

The %c conversion treats 'A' as a character, so it prints as A. The %d, however, treats it as its decimal value 65 and prints that.

(continued on next page)

This example shows what the number stored in the variable **u** looks like when it is printed in different bases by using the different conversion specifications.

```
static char mesg[] = "Different representations of";
static unsigned u = 65535;
printf("%s %u = 0%o, %d, 0x%x\n", mesg, u, u, u, u);
```

prints

```
Different representations of 65535 = 0177777, -1, 0xFFFF
```

Finally, to show the effects of specifying width and fill characters, suppose we wanted to print amounts stored as cents in an integer called **cost**. If the current value in **cost** is 1305 then the call

```
printf("$%3d.%02d", cost/100, cost%100);
```

prints

```
$ 13.05
```

cost/100 is 13 which is printed in a 3 character field, right-adjusted and filled with leading spaces. **cost%100** is 5 which is printed right-adjusted in a 2 character field and filled with leading zeros.

See Also

fprintf, qprintf, sprintf

Name

putc - write a character to a file

Synopsis

```
putc(c, fp)
char c;
FILE *fp;
```

Description

Puts the character **c** on the file pointed to by **fp**. The newline character '\n' will be changed to a CR/LF pair to conform to the CP/M text convention if the file is open for normal (text) output. If it is open for binary I/O, each character will be written exactly as received.

It is an error to write to a file which is not open for output, and in this case nothing will be written.

Returns

The value returned is **c** or **EOF** if an error occurs.

Example

You can write to the file pointed to by **outfp** and check for an error like this

```
if (putc(c, outfp) == EOF) {
        printf("Output disk full\n");
        exit(1);
        }
```

Remarks

The file being written must be opened by calling **fopen** and closed by calling **fclose**. The file pointer **fp** used in the call to **putc** and **fclose** is the value returned by **fopen**.

If you do binary I/O to a printer, you must supply your own carriage return character ('\r') for each newline (linefeed) character if your printer will not do this automatically.

See also

fopen, fclose, putchar

Name
putchar - write a character to the standard output file

Synopsis
```
putchar(c)
char c;
```

Description
Writes the character c on the standard output file **stdout**. If c is the newline character '\n', it is expanded to a carriage return/line feed (CR/LF) pair to conform to the CP/M convention for text.

Returns
The value returned is c.

Remarks
putchar and **getchar** can be used together to easily write programs with one input and one output file. (See **getchar** for a very simple copy program.) No file definitions are needed and the files are opened and closed automatically. If you want, you can also redirect the output file (**stdout**) to the printer for more flexibility. Notice that you must include the redirection capability with the compiler -R switch to do any of these things.

See also
putc, getchar

Name
puts - write a string to the standard output file

Synopsis
```
puts(s)
char *s;
```

Description
Writes the string **s** to **stdout** followed by a newline character '\n'. Any newline character is changed to a CR/LF pair to conform to the CP/M text conventions. Since **puts** always appends a newline, if **s** ends with a newline, you'll end up with two of them.

Returns
There is no return value.

See also
fputs

Name
putw - write a word to a file

Synopsis
```
putw(word, fp)
int word;
FILE *fp;
```

Description
Writes **word** to the file pointed to by **fp**. The word is written by two successive calls to **putc** so the same actions described there for text vs. binary files apply. The word is written low byte first and then high byte.

If the file is not open for output, it is an error and nothing is written.

Returns
The return value is the word written or EOF if an error occurs. Since EOF is just an integer value which may be returned normally you must call **feof** and **ferror** in this case to determine what actually happened.

Example
To write a single word from the file pointed to by **fp** and test for an error say

```
int value;
...
if (putw(value, outfile)) == EOF && ferror(outfile))
        printf("Error writing output file\n");
```

See Also
getw

Name

read - read a file in multiples of 128 bytes

Synopsis

```
read(fd, buffer, n)
int fd, n;
char *buffer;
```

Description

Reads at most **n** bytes from the file with file descriptor **fd** into the location pointed to by **buffer**. For CP/M, **n** must be a multiple of 128. **buffer** is an area defined in your program.

Reading starts at the current value of the read/write pointer. Normally, this is the next CP/M record after the last record read or written. However, the read/write pointer may be changed by calling **seekr**.

When end-of-file is reached you may not get **n** bytes, but **read** tells how many bytes it actually read. The next call will return 0 since there is nothing left to read.

It is an error if the file is not open for input. In this case nothing will be read or placed in **buffer**.

Returns

The value returned is the actual number of bytes read (<= n) so 0 indicates end-of-file. Returns -1 if an error occurs or if **n** is not a multiple of 128.

Example

To read 8 CP/M records (1024 bytes or 1K) at once, say

```
#define BUFSIZE 8*128
char filebuf[BUFSIZE];
...
nread = read(infile, filebuf, BUFSIZE);
```

nread will be set to the number of bytes actually read.

Remarks

To use **read**, you must have opened the file with **open**. The file descriptor **fd** is the return value from the call to **open**.

Under UNIX, **read** is a direct entry to the operating system which lets you read any number of bytes. Here it is an entry to CP/M sequential disk I/O. Since CP/M does its I/O in 128-byte records, you are restricted to multiples of this size. When you use **read**, CP/M transfers the data directly into your buffer so there is very little overhead.

Portability

This function is not in the Standard I/O Library. It simulates a UNIX system call.

See also

open, close, getc, seekr

Name
rindex - find the last occurrence of a character in a string

Synopsis
```
char *rindex(s, c)
char *s, c;
```

Description
Looks for the last occurrence of the character c in the string s.

Returns
The value returned is a pointer to the last occurrence of c in s or NULL if c is not found in s.

Example
Suppose you want to know if the final characters of the string currently contained in s are "xyz". You could check like this:

```
char *p, s[80];
...
if ((p=rindex(s, 'x')) != NULL && strcmp(p, "xyz") == 0) {
    /* do something */
    }
```

See also
index

Name

sbrk - allocate a block of memory space

Synopsis

```
char *sbrk(n)
unsigned n;
```

Description

Allocates a block of n bytes from the space available between the top of any previously allocated memory and the bottom of the stack. A certain amount of space (called the moat) which is directly below the stack cannot be allocated. This is intended to leave space for stack growth so the stack does not grow down into the program memory space.

Returns

The value returned is a pointer to the beginning of the space allocated or -1 if there is not enough free space available.

Example

To allocate a table of 1000 bytes and make **ptable** point to the beginning of the space say

```
char *ptable;

if ((int)(ptable = sbrk(1000)) == -1)
        /* report not enough space available */
```

You should always check the return value to see that you really got the space. Otherwise you will end up destroying the CP/M information in low memory when you store into "your" newly-acquired space. The cast is necessary in the test so the test is done properly since **ptable** is a pointer. Pointers are unsigned so their value is never considered to be negative.

Remarks

The size of the moat is initially set to 1000 bytes, but you can change this with the library function **moat**.

Notice that space obtained from **sbrk** can never be returned for reuse. If you need the space only temporarily and would like to return it for re-allocation by calling **free**, use **malloc**.

Q/C uses **sbrk** to get its table space, thus reducing the size of the executable file CC.COM considerably. Also, the buffered I/O functions such as **getc** get their buffer space by calling **sbrk**. This means your program only requires enough memory to support the number of files that are actually open.

See also

malloc, maxsbrk, moat

Name
scanf - read formatted data from the standard input file

Synopsis
```
scanf(format, arg1, arg2, ...)
char *format;
```

Description
Reads characters from **stdin**, interprets them according to the specifications given in **format**, and stores them in the arguments **arg1**, **arg2**, etc. This works as described in Kernighan and Ritchie with the following exceptions:

(1) None of the specifications for **long** and **float** are available.

(2) The UNIX V7 use of the "h" conversion is followed. This treats "h" as a modifier of other integer conversions, so the valid uses are **%hd**, **%ho**, and **%hx**.

(3) Also, the UNIX V7 use of the "c" conversion is followed. This means that **%c** puts a single character in the matching argument which must be a pointer to character. However, **%nc** puts n characters into the matching argument which must be a pointer to an array of characters.

Returns
The value returned is the number of format items successfully matched and assigned to the arguments. EOF is returned upon end-of-file or error.

Example
You might read a person's name from the console like this:

```
char firstname[10], initial[2], lastname[20];

scanf("%s %1s. %s", firstname, initial, lastname);
```

If you type Dennis M. Ritchie on the console, **firstname** will contain "Dennis", **initial** will contain "M", and **lastname** will contain "Ritchie".

Remarks
The arguments to **scanf** must be pointers or you will not get back the values assigned.

When your input is from the console, the only way that end-of-file will be recognized is if the first character typed on the line is ^Z. If no items are matched, the count returned is zero which is different from EOF.

See Also
fscanf, gets

Bugs
Doesn't implement the UNIX V7 %[...] conversion.

Name
seekr - change read/write pointer

Synopsis
```
seekr(fd, offset, mode)
int fd, mode;
unsigned offset;
```

Description
Changes the read/write pointer for the file with file descriptor **fd** so that the next record will be read from or written to a different location in the file. **offset** says how far to move the read/write pointer measured in CP/M 128 byte records. The offset is measured from the beginning of the file if **mode** is 0, the current record if **mode** is 1, or end-of-file is **mode** is 2. Although **offset** is treated as an unsigned number, the correct result will be obtained if you specify a negative offset when **mode** is 1 or 2.

Returns
The value returned is -1 if an error occurs.

Example
To position a file at the last record before end-of-file say

```
seekr(fd, -1, 2);
```

Remarks
This function does not return the new read/write pointer location because without long integers there is no way to distinguish between random record 65535 and the error return value -1. If you need to know the current read/write pointer location, use the function **tellr**.

Portability
This function is not in the standard I/O library. It is intended to work in a similar fashion to the UNIX system call **lseek** which lets you position a file to a particular byte.

See Also
tellr

Name

setbsize - set the size of a user-supplied buffer

Synopsis

```
setbsize(fp, bufsize)
FILE *fp;
int bufsize;
```

Description

Sets the size of the buffer for the file pointed to by **fp** to **bufsize**. If **bufsize** is not a multiple of 128, it will be rounded down to the next lower multiple. For example, if **bufsize** is 250, only 128 bytes of your buffer will be used. If **bufsize** is less than 128 or if no user buffer was supplied, a standard size system buffer will be allocated.

Returns

No return value.

Remarks

setbsize must be called after calling **setbuf** and before reading or writing the file.

Portability

This function is not in the Standard I/O Library.

See Also

fopen, setbuf

Name

setbuf - provide a user-supplied buffer

Synopsis

```
setbuf(fp, buffer)
FILE *fp;
char *buffer;
```

Description

The **char** array **buffer** is used instead of a system-supplied buffer for the file pointed to by **fp**. If **buffer** is NULL, a system buffer will be allocated. The library routines assume that **buffer** is the same size as a standard system buffer (512 bytes if you haven't modified the Q/C library). This can be overridden by calling **setbsize**.

setbuf must be called after **fopen** but before the file is read or written.

Returns

No return value.

Example

To use a 1K buffer to write the file BIGFILE.TXT say

```
#define BIGBUFSIZE 1024
...
FILE *output;
char buffer[BIGBUFSIZE];
...
if ((output = fopen("BIGFILE.TXT", "w")) == NULL) {
        printf("Can't open BIGFILE.TXT\n");
        exit(0);
        }
setbuf(output, buffer);          /* supply large buffer */
setbsize(output, BIGBUFSIZE);    /* say how big it is */
...
(Now you can start writing to BIGFILE.)
```

Remarks

It is sometimes useful to supply a larger or smaller buffer for certain files. For example, the Q/C compiler uses a 128 byte buffer to read #include files because they are usually fairly short. This saves space at execution time without any real sacrifice in speed. You can tell the library routines your buffer is a different length with the function **setbsize**.

See Also

fopen, setbsize

Name

setjmp - prepare for a non-local jump

Synopsis

```
#include <setjmp.h>
setjmp(env)
jmp_buf env;
```

Description

Saves the current stack environment in the variable **env**. A later call to **longjmp** (which is usually in a different function) will return here as if **setjmp** were returning with the value **longjmp** supplies. All local variables in this function will have the same value they had at the last call to **setjmp**. The function which contains the call to **setjmp** must not have returned in the meantime.

Returns

When you call **setjmp** it returns zero. When you call **longjmp** later, **setjmp** appears to return the value supplied by **longjmp**.

Example

In this example, the call to **setjmp** saves the environment and returns 0. This causes the switch statement to select case 0 and processing begins. If everything works **process** returns and the break statement transfers control to the cleanup at the end of the program. If an error occurs somewhere in **process** or below, calling **longjmp** will return control to the point where **setjmp** returns. The value returned (which should indicate the error) is placed in **err_code** and then the switch causes the appropriate message to be printed and any required fixup action to be performed.

```
#include <setjmp.h>
jmp_buf hold_env;
main() {
        int err_code;
        ...
        err_code = setjmp(hold_env);
        switch (err_code) {
        case 0: process();      /* normal processing */
                break;          /* all went well */
        case 1: printf("Error #1 - program ending\n");
                /* do any fixup needed for this error */
                break;
        ...
                }
        /* Do some cleanup and quit */
        exit(err_code);
        }
```

Remarks

setjmp is used in conjunction with **longjmp** to provide a way of returning from deep within a program when some catastrophic error occurs. This eliminates the need to pass error flags up through many layers of function calls.

See also

longjmp

Name
sprintf - format a string

Synopsis
```
sprintf(s, format, arg1, arg2, ...)
char *s, *format;
```

Description
The arguments **arg1**, **arg2**, ... are formatted according to the specifications given in **format** and put in the string **s**. This works just as described in Kernighan & Ritchie except that the specifications for formatting **long** and **float** variables are not present. See the description of **printf** for more details.

Returns
No return value.

Example
If **s** is a character array and **n** contains 23, the call

```
sprintf(s, "LABEL%03d:", n)
```

will put "LABEL023:" in **s**.

Remarks
If you put the value zero into the middle of the string **s**, this will be considered the end of the string since strings are terminated by a zero '\0'. Any time **s** is used, the characters after the zero will not be seen.

This call

```
sprintf(s, "%s %c %s", "Early end", 0, "Lost");
```

will build a string which is effectively "Early end " because the **0** loaded by the **%c** will terminate the string. The characters " Lost" will be in **s** but won't be seen.

See also
fprintf, printf

Name
sscanf - get formatted data from a string

Synopsis
```
sscanf(s, format, arg1, arg2, ...)
char *s, *format;
```

Description
Takes characters from the string **s**, interprets them according to the specifications given in **format**, and stores them in the arguments **arg1**, **arg2**, etc. This works as described in Kernighan & Ritchie with the following exceptions:

(1) None of the specifications for **long** and **float** are available.

(2) The UNIX V7 use of the "h" conversion is followed. This treats "h" as a modifier of other integer conversions, so the valid uses are **%hd**, **%ho**, and **%hx**.

(3) Also, the UNIX V7 use of the "c" conversion is followed. This means that **%c** puts a single character in the matching argument which must be a pointer to character. However, **%nc** puts **n** characters into the matching argument which must be a pointer to an array of characters.

Returns
The value returned is the number of format items successfully matched and assigned to the arguments. If the first character in the string does not match the format, zero is returned. If **s** is the null string "", then EOF (which is different from zero) is returned.

Example
The following example shows a string containing a date broken up into month, day and year.

```
char date[10], month[3], day[3], year[3];

sscanf(date, "%2s/%2s/%2s", month, day, year);
```

If **date** contains "01/02/84" then after calling **sscanf** **month** will contain "01", **day** will contain "02" and **year** will contain "84".

Remarks
The arguments to **sscanf** must be pointers or you will not get back the values assigned.

See Also
scanf, sscanf

Bugs
Doesn't implement the UNIX V7 **%[...]** conversion.

Name
strcat - append one string to the end of another string

Synopsis
```
char *strcat(s1, s2)
char *s1, *s2;
```

Description
Concatenates the strings **s1** and **s2** by copying **s2** to the end of **s1**.

Returns
A pointer to the beginning of **s1**.

Example
To add the file extension ".ASM" to filename say

```
strcat(filename, ".ASM");
```

If filename contained "PROG" before the call, it will contain "PROG.ASM" afterwards. If filename contained "PROG.C", it would contain "PROG.C:ASM" afterwards.

Remarks
s1 must be long enough to hold the string it currently contains plus **s2** or whatever follows **s1** will be overwritten. Since only the calling program knows how long **s1** is, it must do the checking.

See also
strcpy, strncpy, strmov.

Name
strcmp - compare two strings

Synopsis
```
strcmp(s1, s2)
char *s1, *s2;
```

Description
Compares **s1** and **s2** character by character to determine if they are equal or which string's ASCII representation is higher and lower. Examples are:

```
strcmp("abc", "abc");   /* s1 == s2 */
strcmp("abl", "abc");   /* s1 < s2 since 'l' < 'c' */
strcmp("ab$", "Abc");   /* s1 > s2 since 'a' > 'A' */
```

Returns
strcmp returns zero to indicate the strings are equal, a negative number to indicate that **s1** is less than **s2**, and a positive number if **s1** is greater than **s2**.

Example
The following piece of code indicates how you might use **strcmp** in a sort routine

```
if (strcmp(s1, s2) > 0) {       /* is s1 > s2? */

        /* code to reverse the position of s1 and s2 */

        }
```

Remarks
strcmp is most often used just to test if two strings are the same.

See also
strncmp

Name
strcpy - copy one string into another string

Synopsis
```
char *strcpy(s1, s2)
char *s1, *s2;
```

Description
Copies **s2** into **s1**.

Returns
A pointer to the beginning of **s1**.

Example
To copy the string "Hold this string" into the character array **holdstr** say

```
char holdstr[81];
strcpy(holdstr, "Hold this string");
```

Remarks
The previous contents of **s1** are lost. **s1** must be long enough to hold **s2** or whatever follows **s1** will be overwritten. Since only the calling program knows how long **s1** is, it must do the checking.

See Also
strcat, strncat, strmov.

Name
strlen - calculate the length of a string

Synopsis
```
strlen(s)
char *s;
```

Description
Counts the number of characters in the the string **s** up to but not including the end-of-string character '\0'.

Returns
The length of the string **s**.

Example
To find the length of the second command line argument passed to your C program say

```
arg2len = strlen(argv[2]);
```

If you had run the program by typing

```
A>copy prog.c prog.bak
```

arg2len would be set to 8, the length of "prog.bak".

Name
strmov - copy one string into another

Synopsis
```
char *strmov(s1, s2)
char *s1, *s2;
```

Description
s2 is copied to **s1**.

Returns
The value returned is a pointer to the new end of **s1**. In other words, you get back a pointer to the end-of-string character '\0' at the end of **s1**.

Example
After this series of calls to **strmov**:

```
char mesg[20], *p;
p = strmov(mesg, "one,");
p = strmov(p, "two,");
strmov(p, "three!");
```

mesg will contain "one,two,three!".

Remarks
Since you get back a pointer to the end of the string each time, you can do several calls to **strmov** one after another and fill a character array as shown in the example above.

Portability
strmov is not in the Standard I/O Library.

See also
strcpy, strncpy, strcat, strncat

Name

strncat - append at most n characters from one string to another

Synopsis

```
char *strncat(s1, s2, n)
char *s1, *s2;
int n;
```

Description

Copies at most n characters from string **s2** to the end of **s1**. If **s2** contains less than n characters, the copy will stop with the end-of-string character '\0'. In either case, **s1** will be properly null-terminated.

Returns

A pointer to the beginning of **s1**.

Example

To add the file extension **ext** to filename but copy at most 3 characters say

```
char filename[15], ext[10];
strncat(filename, ext, 3);
```

If **filename** contains "PROG." and **ext** contains "BASIC" before the call, then **filename** will contain "PROG.BAS" afterwards.

Remarks

s1 must be defined long enough to hold the string it currently contains plus a least **n** additional characters or what follows **s1** may be overwritten. Since only the calling program knows how long **s1** is, it must do the checking.

See also

strcat, strcpy, strncpy, strmov.

Name

strncmp - compare at most n characters in two strings

Synopsis

```
strncmp(s1, s2, n)
char *s1, *s2;
int n;
```

Description

Compares **s1** and **s2** character by character up to the end of the shorter string or **n** characters to determine if they are equal or which string's ASCII representation is higher and lower. If one string is shorter than the other and its length is less than **n**, then it will compare low.

Returns

strncmp returns zero to indicate the strings are equal at least up to n characters, a negative number to indicate that **s1** is less than **s2** at least within the first n characters, and a positive number if **s1** is greater than **s2** in the first n characters.

Example

```
strncmp("abc", "abx", 4)    returns < 0 since 'c' < 'x'
strncmp("abc", "abx", 3)    returns < 0 since 'c' < 'x'
strncmp("abc", "abx", 2)    returns 0 since "ab" == "ab"
strncmp("abc", "ab", 3)     returns > 0 since "abc" > "ab"
strncmp("abc", "ab", 2)     returns 0 since "ab" == "ab"
```

See Also

strcmp

Name
strncpy - copy exactly n characters from one string to another

Synopsis
```
char *strncpy(s1, s2, n)
char *s1, *s2;
int n;
```

Description
Copies exactly **n** characters into **s1**. If the length of **s2** is less than **n**, **s1** will contain a copy of **s2** padded with null characters ('\0'). If the length of **s2** is greater than **n-1**, **s1** will not be properly null-terminated.

Returns
A pointer to the beginning of **s1**.

Example
If **s** is defined by

```
char s[6] = "xxxxx";
```

then after the call

```
strncpy(s, "ab", 4);
```

s will contain 'a', 'b', '\0', '\0', 'x', '\0'. An additional null character is copied after "ab" to make 4 characters. After the further call

```
strncpy(s, "abcdefg", 6);
```

s will contain 'a', 'b', 'c', 'd', 'e', 'f' and will not be null-terminated.

Remarks
The previous contents of **s1** are lost. **s1** must be at defined to be at least **n** characters or whatever follows **s1** will be overwritten. Since only the calling program knows how long **s1** is, it must do the checking.

See Also
strcpy, strcat, strncat, strmov.

Name
tellr - returns the current read/write pointer location

Synopsis
```
unsigned tellr(fd)
int fd;
```

Description
Gets the read/write pointer for the file with file descriptor **fd**. This is the next 128 byte record which will be read by **read** or written by **write**. For CP/M this is a number from 0 to 65535.

Returns
The record number in the read/write pointer.

Remarks
The read/write pointer is just the CP/M random record number in the file control block (fcb) for this file.

Portability
This function is not in the Standard I/O Library. It is similar to the UNIX system call **tell** which tells you the next byte which will be read or written.

See Also
seekr

Name

tolower - convert a character to lower case
toupper - convert a character to upper case

Synopsis

```
tolower(c)
int c;

toupper(c)
int c;
```

Description

tolower converts the character 'c' from upper to lower case by adding 32 (which is 'a' - 'A') to it.

toupper converts the character 'c' from lower to upper case by subtracting 32 (which is 'a' - 'A') from it.

Returns

The value returned by **tolower** is c + 32 and by **toupper** is c - 32.

Example

If you say

```
printf("toupper('d') = %c", toupper('d'));
```

you get:

```
toupper('d') = D
```

but it you say

```
printf("toupper('D') = %c", toupper('D'));
```

you get:

```
toupper('D') = $
```

See also

chlower, chupper

Bugs

Notice that these functions don't check to see if c is actually a letter of the right case. They always do the conversion without checking to see if it makes sense. This is the way they are defined in UNIX, so they are provided in this form. For a version of these functions which does what you would expect, see **chlower** and **chupper**.

Name
ungetc - push a character back onto an input file

Synopsis
```
ungetc(c, fp)
int c;
FILE *fp;
```

Description
Pushes the character **c** back onto the file pointed to by **fp**. The next call to **getc** will return this character. Only one character can be pushed back, and **EOF** cannot be pushed back.

If the file is not open for input, it is an error and nothing is done.

Returns
The value returned is **c** or **EOF** if the character cannot be pushed back.

Example
If you are reading a number from a file, you may not know you have reached the end until you find a character which is not numeric:

```
while (isdigit(c = getc(fp))) {
        /* do something */
        }
ungetc(c, fp); /* push back the non-numeric character */
```

Remarks
fp must be either **stdin** or the value returned by **fopen** when the file was open for input.

See also
fopen, getc

Name
unlink - delete a file from the CP/M directory

Synopsis
```
unlink(filename)
char *filename;
```

Description
Deletes the file **filename** from the CP/M directory. **filename** must be a valid CP/M file name.

Returns
The value returned is 0 if successful and -1 otherwise.

Example
To delete the file "CHAPTER1.TXT" on the C: drive regardless of what the currently logged drive is say

```
unlink("C:CHAPTER1.TXT");
```

Portability
This function is not in the Standard I/O Library. It simulates a UNIX system call.

Name

wpeek - look at a word of memory
wpoke - change a word of memory

Synopsis

```
wpeek(address)
unsigned address;

wpoke(address, value)
unsigned address;
int value;
```

Description

wpeek lets you look at the contents of a word in memory.
wpoke changes the value stored in a word of memory.

Returns

The value returned by **wpeek** is the value of the word at **address**. **wpoke** returns the value that was previously stored in the word being changed. In both cases, the contents of **address** is the low order byte of **value**, and the contents of **address**+1 is the high order byte of **value**.

Example

If you wanted to see the location of the CP/M bios by looking at the jump address at location 2H you could say

```
unsigned bios;
bios = wpeek(0x2);
```

Remarks

These functions are useful to work with one or at most a few words in memory. If you want to do more than this, you should define a pointer and do indirection on it. Using pointers, the **wpeek** example above becomes

```
unsigned bios, *pbios;
pbios = 0x2;            /* point to bios address at 2H */
bios = *pbios;          /* get the bios address */
```

Portability

These functions are not in the Standard I/O Library.

See also

peek, poke

Name

write - write a file in multiples of 128 bytes

Synopsis

```
write(fd, buffer, n)
int fd, n;
char *buffer;
```

Description

Writes at most **n** bytes from the location pointed to by **buffer** to the file with file descriptor **fd** at the current read/write pointer location. For CP/M, **n** must be a multiple of 128. **buffer** is an area defined in your program. Normally, the next sequential CP/M record is written. If you change the read/write pointer by calling **seekr**, then you will start writing records sequentially from that point.

It is an error if the file is not open for output. In this case nothing will be written.

Returns

The value returned is the number of bytes actually written or -1 if an error has occurred. After an error occurs, all subsequent calls will return -1.

Remarks

The file being written must be opened by calling **creat** or **open** and closed by calling **close**. The file descriptor **fd** passed to **write** and **close** is the return value from the call to **creat** or **open**.

The error return value (-1) may mean that the file is not open for output or you did not specify a multiple of 128 bytes to be written. Most often though, it means that the disk you are writing on is full.

Under UNIX, **write** is a direct entry to the operating system which lets you write any number of bytes. Here it is an entry to CP/M disk I/O. Since CP/M does its I/O in 128-byte records, you are restricted to multiples of this size. When you use **write**, CP/M transfers the data directly from your buffer so there is very little overhead.

Portability

This function is not in the Standard I/O Library. It simulates a UNIX system call.

See Also

creat, close, open, putc(the buffered I/O equivalent of write), seekr

6

Compiler Internals

This chapter explains the major features of the Q/C compiler's internal operations. It is as accurate as possible, but like any program, the source code itself is the final authority on how Q/C actually works.

6.1 Overview

In the following sections you will follow the compiler through a C program from the highest level view to the lowest. First all preprocessing is done. This includes recognizing and acting upon preprocessor commands and preprocessing your C program code. You can think of this as being done before compilation. Since this is a one-pass compiler, however, each line is preprocessed and then compiled before going on to the next line.

The compiler starts its analysis with external declarations which include the function definitions. Global variable definitions cause assembler code to be generated immediately to reserve storage space and initialize the variable. Function definitions are considerably more complicated. Analysis proceeds through the arguments of the function, the local declarations in the function, and the C statements which make up the function. The statements are separated into expressions, and then the expressions are broken down into their operators and operands. At this point the compiler has reached its lowest level view of the C program. It is now ready to generate the assembler code which will do what you have specified in your C program.

If you look at the **main** function in Q/C you will see the overall flow of the compiler. Most of the functions called by **main** call other functions in turn to do their work. Initial housekeeping is performed by **init**. The signon message is printed, 1200 bytes of stack space are reserved from the memory allocator, and various global variables are initialized. Next, **getoptions** determines the compiler options requested and the names of input and output files. **getspace** obtains compiler table and buffer space from the memory allocator. Then, **zeromem** sets all memory between the top of the program and the bottom of the stack to zero. This allows the compiler to calculate the amount of unused memory at the end of compilation. **getinfil** opens the first or only input file and **getoutfil** opens the output file. **gettable** initializes some table entries and chains together all the free space in the structure member table. Finally, **kill** clears the input line and compilation begins.

The program is brought in a line at time by **inline** and the function **preprocess** looks for preprocesser commands. Conditional compilation is handled here by compiling or ignoring lines of C program code under the control of **#if...else...endif** commands. For all lines which are compiled, **procline** removes comments and extra white space, checks for matching apostrophes on character constants and matching quotes on strings, and also does any replacement of **#define** text.

The second level driver is **parse**. It works through many other functions to produce assembler code equivalent to your C program. After parsing is complete, the compiler does its cleanup. The literal pool for the last function is dumped, the output file is closed, and the compiler summary is printed.

Before these processes are described, let's step back and see how the finished product will look. You may want to review the CP/M Interface Guide for more details on how CP/M lays out memory. The executable part of the program starts at the beginning of the CP/M transient program area (TPA). The external and static variables are included in this part. At the top of the TPA (just below the CP/M FDOS) is the Q/C stack. Automatic local variables and arguments to functions are placed on the stack causing it to grow down towards low memory. The space between the top of the program and the bottom of the stack is free space from which the library functions **malloc** and **sbrk** can allocate memory space for the C program.

File buffer space is allocated from the free space. This has two advantages. The .COM file for your program does not include the space for file buffers so it takes less space on disk and it loads faster. Also, when your program is running no memory space is tied up by file buffers that aren't needed.

The compiler uses this capability to get all of its table space at execution time. The main tables it uses are:

(1) a symbol table to keep track of all identifiers (functions and variables) and their characteristics;
(2) a structure member table which contains a list of pointers to the symbol table entries for the members of each type of structure;
(3) a type table which contains an entry describing each unique type of identifier currently defined;
(4) a macro pool to hold the **#define** definitions and their replacement text;
(5) a literal pool which contains all the strings defined in the program;
(6) a table for loops and **switch** statements to keep track of where **break** and **continue** statements must go; and
(7) a table containing the location and value associated with the **case** labels in a **switch** statement.

The definitions of all of these tables appear in the header file CSTDDEF.H along with the values that the various fields may hold. Now that you have the big picture, let's see the details.

6.2 Preprocessing

As each line of C text is brought in, it is scanned first by the function **preprocess** for #preprocessor commands. The commands **#include** and **#asm/#endasm** are straightforward. The **#include** command causes the function **doinclude** to switch to the include file and remember what the input file was. This process may be nested to a depth of three include files. Q/C uses the function **doasm** to copy assembly code surrounded by the **#asm** and **#endasm** commands directly to the output file. Only two special actions are taken inside these commands. The **#include** command is recognized, and all assembly language comments are stripped out to reduce the size of the output file.

The conditional compilation commands **#if**, **#ifdef**, **#ifndef**, **#else**, and **#endif** work together. When one of the #if commands is found, the required test is done. The global variable **condif** is set to PROCESS or SKIP depending on the result of the test. As each line of the program is read, **preprocess** checks **condif** to see if the line should be compiled or skipped.

If an **#else** command is found, the current setting of **condif** is reversed. If **condif** is not already set, an error message is printed. The variable **condelse** is set true at this point so the compiler can verify that only one **#else** occurs for each **#if**.

When the **#endif** command is found, the compiler checks to see that **condif** is set. If it is, the variables **condif** and **condelse** are set false to signal that conditional compilation is no longer in effect. If **condif** is not set, there is nothing to end so an error message is printed.

These commands can can be nested up to six levels. At any of the nested levels, the variable **condif** can take on a third value, IGNORE. This occurs when the higher level condition is SKIP. Since all input is being skipped because of the higher level condition, the compiler must not change the setting of **condif** when a **#else** is found. Once the nested condition is terminated by a **#endif**, then **condif** reverts to value it had at the higher level.

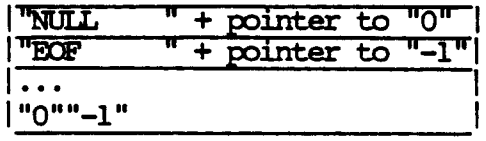
Each time you use the **#define** command, the name and replacement text are stored in a table called the macro pool by the function **addmac**. When the compiler scans your C text in function **procline**, it will call **findmac** for each symbol name it finds. If the name is one for which you have defined some replacement text, the replacement will be done before the line is compiled.

As an example, suppose you give the two definitions

```
#define NULL  0
#define EOF   -1
```

These will be entered in the macro pool as follows. At one end the macro pool is divided into 10 byte fixed-length slots which have room for a legal C symbol name (maximum of 8 characters) plus a 2 byte pointer to the location of

the replacement text for this symbol. By keeping the names a fixed distance apart, these names can be searched quickly by simply doing a serial search through the table until either the name is found or the end of the names already defined is reached. The replacement text is entered at the other end of the macro pool as a null-terminated string. The pointer stored with the macro name points to the beginning of this string so it can be retrieved. After the two definitions above the macro pool will look like



The **#undef** command finds the definition of the specified name in the macro pool (if it exists) and changes the name to the null string. This prevents subsequent references to this name from finding the entry, but it does not free the space used by the entry.

6.3 Type Handling

At the highest level, your program consists of definitions of identifiers -- either global variables or function definitions. Identifiers have two basic properties: their storage class and their type. The storage classes possible for an identifier depend on whether it is declared globally (outside any function) or locally (within a function). Because of this, storage classes will be discussed at the different possible levels.

The types possible for an identifier are the same at both levels with one exception. A function cannot be local to (i.e. known only inside) another function. Therefore, type handling is done only on one level. The type table contains a single entry for each type of identifier currently active. When there are no variables of a given type defined, the type table entry can be reused. This only occurs, however, when the type table fills. In the following discussion, all symbolic constants (for example T_INT) are defined in the header file **CSTDDEF.H**.

Type Table

The type table is an array of structures with entries of the form:

```
struct typeinfo {
        char            t_code,
        int             t_size,
        int             t_refs,
        union baseinfo {
                struct memtab *memlist;
                struct typeinfo *p_type;
                }       *t_base;
        struct typeinfo *t_next;
        };
```

The first field, **t_code**, contains one of the fundamental C types such as T_INT for an **int** variable, or one of the derived types such as T_ARRAY for an array. The second field contains the size of one instance of this type. Thus an integer variable will have a size of 2, whereas an array of 5 integers will have a size of 10. The next field, **t_refs**, is simply a count of how many references there are to this table entry. When this number drops to zero, the entry can be reused.

Skipping the field **t_base** temporarily, **t_next** ties all entries for a given fundamental or derived type together in a list. An array of structures called **basetypes** contains the head of each list. As an example, the list of array entries (meaning those with **t_code** containing T_ARRAY) starts at *basetype[T_ARRAY]. The field **t_next** in this entry points to the next array entry and the final array entry contains a pointer back to **basetype** signalling the end of the list.

To simplify type processing, the function **inittypes** initializes the type table with entries for all fundamental types (and for the derived type function returning integer since it is so common). The **t_refs** field is set to one so that these entries will never be removed from the type table even if there is currently no active identifier of the given type. Global pointers are defined for each of these permanent entries in the global definition file **CGLBDEF.C** with names like **chartype** and **inttype**.

The derived type entries are built up using the **baseinfo** union **t_base**, while fundamental types contain NULL in the this field. All derived types except structures use the **typeinfo** pointer **p_type** to link the type information. For example, the derived type "pointer to character" is represented by an entry whose type is pointer and whose base type is **char**. Thus the entry has T_PTR in **t_code** and a pointer to the entry for the fundamental type **char** in **t_base**. More complicated types are built up by chaining more entries together through the base type field.

Structures use the **memtab** pointer **memlist** in **t_base** to point to the list of members for this type of structure. The member lists are contained in another table called the structure member table.

Filling in the Type Table: Some Examples

Let's create some type table entries to make this discussion clear. We'll concentrate on character variables to show the possibilities, but the same types of entries are made for signed and unsigned integers. Initially the type table contains one entry for the type "character" pointed to by **chartype**. If the type table is located at address 1000 (decimal) it looks like:

Address	t_code	t_size	t_refs	t_base	t_next
1000	T_CHAR	1	1	NULL	&basetypes[T_CHAR]
...					

The ellipsis (...) represents the other permanent entries. Now suppose a simple character variable is defined like

```
char c;
```

Since there is already an entry for the fundamental type "character", the only change is to increment **t_refs** making it 2.

Next suppose a pointer to character variable is defined by

```
char *pc;
```

A new entry is added to the table making it look like:

Address	t_code	t_size	t_refs	t_base	t_next
1000	T_CHAR	1	3	NULL	&basetypes[T_CHAR]
...					
1045	T_PTR	2	1	1000	&basetypes[T_PTR]

This entry is located at 1045 (decimal). It has a type of T_PTR and a size of 2 bytes since pointers are 16 bit addresses, and it has one reference to it. This time the base type is "character" so **t_base** contains 1000, the address of the fundamental type. Since the type "character" now has another reference to it, its **t_refs** field is incremented to 3. There are no other pointer types defined yet, so **t_next** points back to the head of the list of pointers.

Now suppose an array of 10 characters is defined by:

```
char ac[10];
```

A new type entry is created giving:

Address	t_code	t_size	t_refs	t_base	t_next
1000	T_CHAR	1	4	NULL	&basetypes[T_CHAR]
...					
1045	T_PTR	2	1	1000	&basetypes[T_PTR]
1054	T_ARRAY	10	1	1000	&basetypes[T_ARRAY]

t_refs for the fundamental type "character" is now 4. An entry has been added for an array with size 10 bytes whose base type is "character" located at address 1000.

If a array of 10 pointers to character is defined by:

```
char *apc[10];
```

the type table looks like:

Address	t_code	t_size	t_refs	t_base	t_next
1000	T_CHAR	1	4	NULL	&basetypes[T_CHAR]
...					
1045	T_PTR	2	2	1000	&basetypes[T_PTR]
1054	T_ARRAY	10	1	1000	1063
1063	T_ARRAY	20	1	1045	&basetypes[T_ARRAY]

The entry for "pointer to character" at location 1045 now has 2 references to it. The first entry for type array (T_ARRAY) now points to location 1063 as the next entry of type array. Finally, only one new entry was needed for type array of 10 pointers to character since an entry already existed for the derived type "pointer to character". Notice that the new entry has a size of 20 bytes vice the 10 for the preceding entry since pointers to character take 2 bytes each. Also, **t_base** of the new entry points to location 1045 which is itself a derived type pointing to the fundamental type "character" at location 1000.

Parsing Derived Types

Declarations are identified by the function **isdecl** which returns a pointer to a type table entry (among other things) when it finds a declaration. Normally this will be one of the fundamental types such as **chartype** for character variables. When the identifier being declared is a derived type like "array of character", the function **dodecl** parses the identifier. It uses a type parsing stack whose important part is defined as:

```
struct typestack {
        char    t_code;
        int     t_size;
        };
```

dodecl looks for the pointer operator "*", function operator "()", array operator "[]", and parentheses which simply alter the normal associations of

the other operators. **dodecl** calls itself recursively to parse the declarations in the proper order. Each time it finds one of the operators it is looking for, it calls **pushtype** which records the code and size in the **typestack** structure given above.

As an example, the last type discussed in the previous section, "array of 10 pointers to character", would generate the type stack entries:

t_code	t_size
T_PTR	2
T_ARRAY	10

After the parsing is complete, the function **loadtype** uses the information in the type stack to build the list of type table entries for this type as discussed in the preceding section.

Structure Member Table

The structure member table records the members contained in each type of structure currently defined. As I mentioned earlier, each structure entry in the type table points to the list of its members in the structure member table. This allows Q/C to check for valid initialization of structures and to define the right type and size variable for each structure member.

The member table is an array of structures with entries:

```
struct memtab {
        struct st       *p_sym;
        struct memtab   *nextmem;
        };
```

Initially all the space in unused, and the entire table is chained together by the **nextmem** pointer. The final entry contains NULL in this field. Q/C keeps the address of the beginning of the member table in the global pointer **memtab** and the head of the free space list in the pointer **freemem**.

As structure declarations are parsed by **decltag**, the member table is filled in by calling the function **addmem**. When a member is added to the symbol table, **p_sym** is assigned the address of the symbol table entry. The **nextmem** pointer in the previous member entry is set to point to the current member entry. This chaining of entries is necessary because structures can be declared inside of other structures. In this case, **decltag** is called recursively and starts parsing the inner structure declaration. When it is through and the inner member list has been completed, it continues to work on the outer list. The final entry in the list contains NULL in the pointer **nextmem**.

Local structure declarations are only in effect until the end of the function, so they must be removed from the member table. This is done by calling the function **delmem** to return the member list to the free space chain.

delmem changes the last entry in the list so that it points to the current head of the free space list, and then changes the head of the free space list to the beginning of the list being deleted.

6.4 Globals/Functions

Typically, your program begins with the declaration of global (external) variables. These are followed by the definition of the functions which make up your C program. The second level driver function, **parse**, searches through your program looking for global declarations and function definitions until end-of-file is reached.

Global Variables

parse calls **isdecl** to look for declaration keywords specifying a storage class (such as **static**), a type (such as **char**) or both. **isdecl** will find all keywords and return the storage class if any and a pointer to a type table entry if a type or **typedef** name was found. If no storage class specifier is given, the default global (SC_GLOBAL) is used. Similarly, if no type specifier is given, the type defaults to **int** (the pointer **inttype** discussed in Section 6.3 "Type Handling"). If neither specifier is found, **parse** assumes it has found the definition of a function which returns an integer. In this case, both defaults are taken.

Notice that **typedef** definitions are normally given at the beginning of the program and thus are actually global declarations. **typedef** is treated just like any of the usual storage classes. Once a **typedef** definition has been given, the name defined can be used in place of the ordinary type specifiers such as **int**.

parse calls **declglb** and passes it the storage class and type pointer. **declglb** determines the names and characteristics (such as pointer) of each variable being declared by calling **declvar** for each name in the declaration list. If the name is a function, then **isfunc** is called to either process the function definition or determine that this is simply a declaration of the type the function returns. Finally **addglb** is called to add the identifier to the symbol table for future use.

If a variable is being defined, **doinit** is called to initialize it. When no explicit initialization is given, or when an array is not completely initialized, **definit** puts in zeros.

Symbol Table (Part 1: Globals)

Every global and local variable declared in the C program has an entry in the symbol table. Global variables, unlike local variables, remain in the symbol table until the end of the program since their scope extends over the entire program. Entries for local variables will be discussed in section 6.5 "Local Declarations".

The symbol table entry contains the name of the variable, its characteristics, and any additional information needed to find it when it is used in an expression. The symbol table is an array of structures with the entries defined as:

```
struct st {
        char    st_sc;          /* storage class */
        struct typeinfo
                *st_type;       /* pointer to type table */
        int     st_info;        /* see Table 6-2 */
        char    st_name[NAMESIZE];
        char    st_idset;       /* tag/member or variable */
        };
```

Table 6-1 shows the possible values for each field in an entry. These values are shown as symbolic constants which are defined in the file **CSTDDEF.H** on the Q/C disk. The information field **st_info** is used in a number of different ways depending on the what kind of variable is represented in this entry and what its storage class is. Table 6-2 shows all the uses of the information field.

The use of most fields in the symbol table entry should be clear from Table 6-1. The storage class field **st_sc** can have the values SC_GLOBAL, SC_ST_GLB or SC_EXTERN depending on how the global variable is declared. Given the declarations

```
int i;
static int j;
extern int k;
```

the first declaration says i is being defined and that storage space should be reserved. The second declaration says that j is being defined as a global in this file but is not known in any other file. The final declaration says that k has been defined elsewhere, but be aware that it exists because this program will refer to it. The compiler puts SC_GLOBAL in the storage class field of the symbol table entry for i, SC_ST_GLB for j, and SC_EXTERN for k.

For all global variables defined in this file, the information field **st_info** contains the symbolic constant DECL_GLB to indicate that this name is known throughout the file. If the variable is only declared as **extern** either explicitly or by making a function call, then the information field can also contain DECL_LOC. This indicates that the name is only known in the current function, but that the compiler should compare all references to this name to see that they agree on the type of the variable.

Field	Possible Values	Comments
st_sc	SC_GLOBAL	A plain global definition
	SC_ST_GLB	A static global definition
	SC_EXTERN	A global which has only been declared
	SC_STATIC	A local **static** variable
	SC_AUTO	A local **auto** variable
	SC_ARG	An argument to a function
	SC_REG	A local **register** variable
	SC_MEMBER	A member of a structure
	SC_TYPE	A structure tag or **typedef** name
st_type	pointer	Points to the type table entry describing this type of variable
st_info	Table 6-2	Depends on **st_type** and **st_id** fields
st_name	Any valid symbol name	Null terminated string containing the name of this symbol
st_idset	ID_VAR	A ordinary variable or **typedef** name
	ID_STRUCT	A structure/union tag or member

Table 6-1. Possible values in a symbol table entry.

st_idset	st_sc	st_info
ID_VAR	SC_GLOBAL/ SC_ST_GLB	DECL_GLB — known throughout this source file
	SC_EXTERN/ SC_TYPE	DECL_GLB — known throughout this source file DECL_LOC — known only in the current function
	SC_AUTO/ SC_ARG	Offset in local stack frame for a variable or internal label number for a statement label
	SC_STATIC	Internal label number assigned to this variable
	SC_REG	Register number assigned to this variable
ID_STRUCT	SC_TYPE	DECL_GLB — known throughout this source file DECL_LOC — known only in the current function
	SC_MEMBER	Offset from the beginning of the structure

Table 6-2. Uses of the information field (**st_info**) in the symbol table.

Function Definition

When Q/C sees a declaration of a function, it calls **isfunc** to decide whether this is the definition or just a declaration of the function name and the type it returns. A function definition is distinguished by the presence of an argument list in parentheses or an opening brace "{" following the empty parentheses. **isfunc** returns FALSE if it is only a declaration. If it is the definition of the function, however, it analyzes the function, generates assembler code, and then returns TRUE. **isfunc** does its work by calling a number of other functions.

The next few sections discuss arguments, local variable declarations, and then the statements which make up the function.

6.5 Arguments

When a function is called, the arguments are pushed onto the stack by the calling routine in the order they are seen. To retrieve the arguments, the function being called must know this order. This is determined by the way the arguments appear in the argument list. If the function definition begins

```
function(arg1, arg2)
```

then it knows that **arg1** was pushed on the stack first and **arg2** next. Since the call pushed the return address on the stack after that, the stack looks like this

```
SP —> return-address
      arg2
      arg1
```

when the function is entered. To retrieve an argument, it is only necessary to know its offset from the stack pointer (SP).

Symbol Table (Part 2: Arguments)

Information about the arguments is recorded in the symbol table similarly to the way it is recorded for global variables. The arguments of a function (along with any variables declared inside the function) are local to the function, however. This means that their names are known only inside the function. In fact, if the local name is the same as a global name, the local name takes precedence and the global name is temporarily not available.

To accomplish this, global variables are entered at one end of the symbol table and remain there permanently. Local variables are entered starting at the other end of the symbol table and remain only until the compiler is through compiling the function. When the next function is being compiled, its local variables will reuse the local symbol table entries, and they will be the only local variable names known.

By sharing the symbol table in this way, it does not need to be partitioned rigidly into a global and a local section. Since the symbol table will not be filled until there are no more entries available for either type of variable, the amount of space allocated for the symbol table can be smaller. A program with an unusually large number of either type of variable will not cause one part of the table to overflow while the other part still has plenty of room.

Since the location of the arguments is determined by the order of the argument list, and the characteristics are determined by the declaration list, the symbol table entry is built in two parts. The argument list is processed by **procarg** which adds each argument to the local symbol table. Then **declarg** analyzes the argument declarations. It determines their characteristics (such as integer, pointer, etc.) and also checks for multiple declaration of arguments.

As each argument is found in the argument list, its name is recorded in the local portion of the symbol table. The type pointer **st_type** is set to NULL to indicate that the argument has not been declared yet. When a variable is found in the declaration list, its type entry is checked. If it is NULL, the entry is changed to point to the appropriate type table entry. This also indicates that this name has been declared. If the type pointer is not NULL, this is a multiple declaration, so an error message is printed.

6.6 Local Declarations

Q/C expects all local declarations to be given at the beginning of the function body which means immediately after the first opening brace {. This is a deviation from standard C which allows local declarations at the head of any compound statement or in other words after any opening brace. This restriction simplifies compilation and code generation considerably. The compiler only has to be concerned with one level of local variables rather than a number of levels each of which may supersede variables declared in the higher levels.

It also simplifies managing the stack. Since space for automatic local variables is reserved on the stack, this space must be returned whenever local variables are deallocated. If variables may be declared in any compound statement then the space for these variables must be freed whenever the block is left. This means that **break**, **goto**, **continue**, and **return** statements as well as simply exiting a block must all be concerned with freeing varying amounts of stack space.

The function **procdecl** finds all the declaration statements at the beginning of a function. This is done by calling **isdecl** repeatedly just as globals and arguments are handled. In the case of local variables all storage classes are allowed so no special error checking is needed. Defaults are set for storage class and type if needed. Then **declloc** is called to process the variables that are being declared.

declloc calls **declvar** for each name it encounters. **declvar** checks for

valid symbol names, duplicate declarations, computes the size of each variable, and determines its characteristics. For **static** variables, it also does initialization by calling **doinit**. This works just as described in section 6.4 under the heading "External Declarations". After **declvar** is through, **decllocc** calls **addloc** to add the symbol table entry for each variable.

Symbol Table (Part 3: Locals)

Once again Tables 6-1 and 6-2 show the complete range of possible values for the symbol table entry of a local variable. The discussion here is broken down by storage classes since each class is handled differently. Labels are considered separately since they do not really belong in any storage class.

Externals

Any variables declared as **extern** are added to both the global and local portions of the symbol table. Function declarations are always considered to be **extern**. In the global entry, the information field is set to DECL_LOC. this allows the compiler to check for consistent usage throughout the source file while still limiting the scope of the local declaration to an individual function.

Automatic Variables

Automatic (**auto**) variables are given space on the stack. To retrieve an automatic variable, its offset from the beginning of the stack frame for this function is recorded in the information field of the symbol table entry.

Register Variables

Register variables are held in a special place where they can be retrieved easily. Other than this, they act like automatic variables. Only variables which occupy two bytes (which means integers and pointers) will be put in a register. Also, there are only five registers available, so not all variables declared as **register** will end up in a register. If a variable is placed in a register, the register number will be recorded in the information field of the symbol table entry.

Static Variables

If a local variable is declared **static**, the compiler reserves space for it in memory inside the function. This space is given a compiler-generated internal label so it does not conflict with any other local or global variables with the same name. The internal label number is recorded in the information field of the symbol table entry.

Labels

Statement labels exist only to give the **goto** statement a place to go. The compiler has several things to concern it when you use labels. A label is declared the first time it is seen whether it appears in a **goto** statement or

as the label on a C statement. At this time it is assigned an internal label number, and it is added to the symbol table. Its type points to the **labeltype** entry in the type table. If it appears in a **goto**, its storage class is set to SC_NONE. Any further appearances in **goto** statements cause its symbol table entry to be checked, and its internal label number to be retrieved for use in the assembler output. When it used as the label on a statement, however, its storage class is made SC_AUTO and it does not change after this.

This allows the compiler to check for duplicate statement labels and undefined statement labels. If a label appears on a statement and the symbol table entry says its storage class is SC_AUTO, this means it has already appeared as the label of a statement so it is a duplicate. At the end of the function, the compiler checks all of the local symbol table entries. If it finds an entry with a type of label and a storage class of SC_NONE, this is a label which was used in a **goto** statement but never appeared as the label of a statement.

6.7 Statements

The local declarations are followed by the statements which make up the body of the function. After **isfunc** finishes with the declarations, it calls **compound** to process the rest of the function body. **compound** in turn calls **statement** repeatedly until the highest level compound statement which started the function is completed. The function **statement** is very straightforward. It calls **chklabel** to see if the statement has a label. Then it looks for the keyword (such as **if**, **for**, etc.) to see what type of statement it has.

There is a function to process each statement type. For example, **if** statements are handled by the function **doif**. These functions check the syntax to see that the required keywords and punctuation are present. They also save any required information, such as **case** values for **switch** statements, in tables so the assembler code can be generated.

These functions do not do any code generation themselves. They call on a number of lower level functions for this. As an example, the call **jumpcond(FALSE, label)** produces the assembler code to test the value of an expression and then jump to **label** if the expression is false. **label** is an internal label generated by the compiler.

Statement Expansion

The compiler breaks the logic of each C statement type into simple combinations of tests and jumps since this is what can be done in assembler language. Each C statement is shown in its general form and then as the compiler implements it in terms of tests and jumps. Tests are shown like a simple C **if** statement which has no **else**. Jumps are shown as **goto**s. The labels are all internal labels generated by the compiler so they start with **?** which is the usual convention in the assembler code generated. The terms **expression** and **statement** represent normal C expressions and statements which appear in the locations shown.

Loops (**while**, **do...while**, and **for**) and the **switch** statement must also provide for the **continue** and **break** statements. In the expansions **?cont** is the label for a **continue** statement to jump to, and **?brk** is the label for a **break**.

if (expression) statement

```
            if ( ! expression )
                    goto ?1
            statement
?1:
```

if (expression) statement1 else statement2

```
            if ( ! expression )
                    goto ?1
            statement1
            goto ?2
?1:         statement2
?2:
```

while (expression) statement

```
?cont:  if ( ! expression )
                goto ?brk
        statement
        goto ?cont
?brk:
```

do statement while (expression)

```
?1:     statement
?cont:  if ( expression )
                goto ?1
?brk:
```

for (expression1 ; expression2 ; expression3) statement

```
        expression1
?1:     if ( ! expression2 )
                goto ?brk
        goto ?2
?cont:  expression3
        goto ?1
?2:     statement
        goto ?cont
?brk:
```

```
switch ( expression )
case constant-expression: statement
 ...
default: statement

                expression
                goto ?1
        ?2:     statement       /* case 1 */
                ...             /* more cases */
        ?N:     statement       /* default case */
                goto ?brk
        ?1:     [call the run-time library routine to pick case]
                [argument list of case values and matching labels]
        ?brk:
```

```
break

                goto ?brk
```

```
continue

                goto ?cont
```

The two statement types not shown have no real expansion. A **return** statement is simply an assembler return. A **goto** is done as a jump to a compiler-generated label. These labels are discussed under the heading Symbol Table (Part 3: Locals).

6.8 Expressions

The C statements are made up of either simpler C statements or keywords and expressions plus punctuation. The expressions are made up from primaries and operators. If you look at the compiler source code you will see that a great deal of it is devoted to parsing expressions.

The major complicating feature of parsing expressions is applying the operators in the correct order. Each operator works on one or more operands. For example, in the expression

```
a + b
```

the operator is **+** and its operands are **a** and **b**. If an expression could just be scanned from left to right determining the operands and doing the specified operations, there would be almost nothing to it. However, as shown in Chapter 2 of Kernighan & Ritchie, there is a particular order in which operators are applied. In the expression

```
a + b * c
```

the multiplication must be done before the addition so in effect this expression is

```
a + (b * c)
```

As if this weren't hard enough, some of the operators have the same priority and then they have to be applied either left-to-right or right-to-left. A common example of this is stepping through an array of characters using a pointer

```
*++p
```

Both operators * and ++ have the same priority but they are grouped right-to-left. This means that this expression is evaluated as

```
*(++p)
```

First the value of the pointer p must be retrieved, then it must be incremented, and finally the character it points to is loaded.

Because of this priority scheme, the operands of any particular operator can be expressions involving any of the operators having a higher priority. To make this concrete consider the expression

```
a * b + c * d
```

Since multiplication has a higher priority than addition a * b and c * d will be done first. Then those two quantities will be added to get the value of the expression. In effect this expression is

```
(a * b) + (c * d)
```

The operands of the addition operator + are the expressions a * b and c * d.

Since Q/C makes one pass through your C program from left to right, it must remember which variables are having what operations performed on them and in what order. This is done by using the technique of recursive descent parsing.

6.9 Recursive Descent Parsing

Recursive descent parsing uses the ability to call subroutines recursively to record the variables, operators, and order of operations in the expression being parsed on the stack. The record is composed of local variables in the routines which parse the different operators and the order in which the routines are called. Recursion is used when an inner expression is enclosed in parentheses. Then the parser calls itself to evaluate the inner expression first.

The functions which make up the recursive descent parser are **expression**, **heir1** through **heir15**, and **primary** which are all in CC4.C. If you look at the table in Chapter 2 of Kernighan & Ritchie which shows the order of evaluation of operators, you'll notice that some operators are at the same level. The

functions **heir1** through **heir15** each handle the operators at one level in this table. The direction of increasing priority is from **heir1** to **heir15**. Thus **heir1** parses the sequence (comma) operator, while **heir15** handles function references () and array references []. The variables or constants involved in these expressions are identified by the function **primary**.

A Parsing Example

Whenever C syntax calls for an expression the function **expression** is called to parse the expression. When the argument **load** is TRUE, the expression must be evaluated and the value be available for further use. If **load** is FALSE, the expression is evaluated but the final value is not loaded if it is not a by-product of the code to evaluate the expression. **expression** calls **heir1** which calls **heir2** and so on. All of these functions work in a similar way. The operators must be recognized and their operands must be identified.

To see how this is done in Q/C, let's parse the simple expression

```
a + b * c
```

discussed in the last section. Recall that multiplication has a higher priority than addition, so the expression is evaluated as if it were written

```
a + (b * c)
```

You may want to look at a listing of the functions **heir1** through **heir15** and **primary** as the expression is parsed.

Each routine starting with **heir1** calls the next level routine to see if there are any higher priority operators to be evaluated first. **heir12**, which handles addition, is looking for an expression like this

```
expression1 + expression2
```

where + is the addition operator, **expression1** and **expression2** are the operands of the addition operator, and the result will be the value of **expression1** plus the value of **expression2**. The general form of the expression **heir12** is looking for is

```
operand operator operand
```

Thus the first thing **heir12** needs if indeed it is looking at an addition expression is the operand **expression1**. The operands of a given operator are expressions possibly involving any higher priority operators as discussed in the previous section. To find an expression involving any of the higher priority operators, **heir12** calls **heir13**. When the lower level subroutines return to **heir12**, they will have already parsed the expressions involving higher priority operators such as multiplication.

In this case, the series of calls reaches **primary** which identifies the

variable a. At this point the compiler moves its attention to the next item in the expression — the addition operator +. **primary** returns to **heir15** which does not find any of the operators it is looking for, so it returns to **heir14**. This continues until **heir12** is reached. **heir12** is looking for a + or a -, and a + is what it finds. It has now identified the left operand (a) and the operator (+) of an addition expression. It now needs the right operand. Since this operand may be an expression involving higher priority operators, **heir12** once again calls **heir13** to get an operand.

This time when **primary** is reached it finds the variable b. Control is once again returned to **heir15**. It sees the multiplication operator * which it is not interested in so it returns. When **heir13** is reached, however, it is looking for an *. It now recognizes that it has found b * which is the left operand and operator of a multiplication expression. It needs a right operand to complete the expression, so it calls **heir14** to evaluate the right operand.

Once again **primary** is reached and this time identifies the variable c. When control returns to **heir13** it now has an entire multiplication expression

```
b       *        c
operand operator operand
```

so its job is done (We temporarily ignore the need to generate assembler code to do the multiplication).

heir13 returns to **heir12** which now has its complete addition expression

```
a       +        b * c
operand operator operand
```

The multiplication has already been done, so **heir12** has the correct operands — the variable a and the expression b * c.

This completes the parsing of the original expression. The operations specified have been found and applied in the correct sequence according to the order of operations given in Kernighan & Ritchie. Any expression, no matter how complicated, can be parsed just the way it was in this example.

Recording the Parse Results

Now that you have seen the parsing you may be wondering how the different level routines tell each other what they have found. The method is fairly simple. Each time one routine calls another to find an operand, it passes a structure with four elements as an argument to the called routine. The called routine (or some lower level called routine) fills in the structure.

The structure is defined as:

```
struct operand {
        char            op_load;
        struct st       *op_sym;
        int             op_val;
        struct typeinfo *op_type;
        };
```

The entry **op_sym** contains a pointer to the symbol table if the expression being parsed consists of a variable which has not yet been loaded. Otherwise, it contains NULL. **op_type** simply contains a pointer to the type table entry which describes the type of the expression parsed so far.

The more interesting entries are **op_load** and **op_val**. **op_load** can take on a number of values (all defined in CSTDDEF.H). Each value sets a different bit, and it is possible for more than one bit to be set.

EXPRESSION	- an expression whose value is already loaded
LOADVALUE	- the value of the expression must be loaded
LOADADDR	- the address of the variable must be loaded
LVALUE	- the expression is an lvalue
CONSTANT	- a constant expression suitable for use as an initializer, an array dimension, or a case value
CONSTADDR	- the address of a global or static variable plus a constant offset suitable for use as an initializer
CONSTOFF	- a local variable whose address plus a constant offset must be loaded
ASGCONST	- a constant which appeared on the right side of an assignment statement but which wasn't loaded

The entry **op_val** holds different values depending on the contents of **op_load**. When **op_load** is CONSTANT or ASGCONST, it contains the value of the constant. For CONSTADDR or CONSTOFF, it contains the offset to be added to the address of the variable.

6.10 Code Generation

The last section mentioned the fact that assembler code must be generated as soon as the parsing of an expression is completed. This is required because Q/C is a one pass compiler. Once compilation has moved past an expression, the information that was recorded on the stack indicating the variables involved and the order of operations is lost. The only permanent record of what was to be done is the assembler code generated to carry out the specified operations.

Overview

Basically, there are two kinds of assembler code generated. For some things, a series of assembler statements is generated at that point in the program to carry out the operations specified by the C code. This is called inline code. Other operations are done over and over the same way, so the

compiler calls on a small group of subroutines when these operations are needed. These routines, which have been mentioned before, are called the compiler support routines.

The compiler support routines are all written in assembler code and reside in the file **CRUNTIME.MAC**. They consist of routines to do 16 bit arithmetic, routines to load and store variables, and routines to perform most of the C operators (for example comparison operators such as >=). The compiler generates calls to these routines automatically when they are needed, so you will not be aware of them unless you look at the assembler code generated.

To simplify accessing arguments and local variables on the stack, Q/C uses a constant stack frame pointer. This means that the compiler maintains its own stack frame using the BC register pair (and the index register IX in the Z80 version). Regardless of where the stack pointer (SP register), local variables and arguments are at a constant offset from the stack frame pointer. Typically, all functions now start and end with a call to a library routine which manages the activation record for the function. These calls are generated if any of the following services are needed:

- allocate and free stack space for local **auto** variables
- save and restore the Q/C "registers" for local **register** variables
- save and restore the constant stack frame pointer (register BC/IX)

Q/C ensures that your global names do not conflict with assembler reserved words or with compiler-generated labels by adding a ? to the end of your names, and by starting all compiler-generated labels with a ?.

An Example

Before discussing this any further, let's see what code generation looks like. As a example, the C code

```
int i, j, k;
i = j * k;
```

will generate the assembler code:

```
PUSH    B       ;save the address of i for the assignment
LXI     H,2     ;load the offset of j from the stack frame ptr
DAD     B       ;compute the address of j
CALL    ?g      ;load the value of j in the HL register pair
XCHG            ;put the value of j in the DE register pair
LXI     H,4     ;load the offset of k from the stack frame ptr
DAD     B       ;compute the address of k
CALL    ?g      ;load the value of k in HL
CALL    ?mult   ;multiply HL by DE placing the result in HL
POP     D       ;retrieve the address of i
CALL    ?p      ;store HL at the address contained in DE
```

The comments are not part of the generated code. They only help to explain what is taking place in the example.

The local **auto** variables **i**, **j**, and **k** are all stored on the stack. To reference them, their address is computed by adding an offset to the address contained in the stack frame pointer (BC). Recall that this offset is recorded in the information field of the symbol table entry for each local **auto** variable.

When Q/C needs to retrieve an **auto** variable, it first computes the address and then calls a compiler support routine to do the actual load. In the example **i** is the first local variable so its offset is zero from the beginning of the stack frame. The addresses of **j** and **k** are computed, and since they are integers, a call is generated to the routine **?g** which loads a 16 bit integer from the address contained in the HL register pair and places it in HL. Once the variables **j** and **k** are in the proper location, a call is made to **?mult** which multiplies DE (containing **j**) by HL (containing **k**) and places the result in HL. Finally, the assignment to **i** is done. The **POP D** loads the address of **i** (which was saved by the PUSH B instruction earlier). Then the assignment is done by calling **?p** which stores the contents of HL (**j * k**) at the address contained in DE.

The Z80 version can do this with less code by using the index register IX to access the variables. In this case the code generated is:

```
LD      L,IX+2  ;load the value of j
LD      H,IX+3
XCHG            ;put the value of j in DE
LD      L,IX+4  ;load the value of k
LD      H,IX+5
CALL    ?mult   ;multiply HL by DE placing the result in HL
LD      IX+0,L  ;store the result in i
LD      IX+1,H
```

This discussion cannot begin to describe all the possible code generation situations. The major features of code generation will be covered however. In this discussion the following terms will be used:

unary operator	- an operator with one operand (like * in *p)
binary operator	- an operator with two operands (like + in a + b)
left operand	- first operand of a binary operator (a in a + b) (abbreviated **lop** in comments in the compiler)
right operand	- right operand of a binary operator (b in a + b) (abbreviated **rop** in comments in the compiler)
primary register	- the register pair HL (abbreviated **preg**)
secondary register	- the register pair DE (abbreviated **sreg**)

Whenever a variable is referenced either it or its address is loaded into the primary register. Each of the four storage classes are handled differently. Table 6-2 summarized the information which is retained in the symbol table so each type of variable can be located.

Auto Variables

As you saw in the earlier example, **auto** variables are loaded by using their offset from the stack frame pointer to compute their address. Then a compiler support routine is called to do an indirect load using the address in the primary register in the case of 8080 code. The routine **?gc** retrieves the character at the address in the primary register and sign extends to 16 bits in the primary register. The routine **?g** retrieves the 16 bit integer at the address in the primary register and loads it into the primary register. Similarly, the routine **?p** stores an integer. Z80 code loads and stores the variables directly using the indexed load instructions.

Static Variables

Local variables whose storage class is **static** are assigned an internal label number and have storage reserved and initialized to zero like this:

```
?1:     DB      0       ;this is a character variable
?2:     DW      0       ;this is an integer variable
```

To retrieve these the code generated is

```
        LDA     ?1      ;load the character into A
        CALL    ?sxt    ; and sign extend into HL
        LHLD    ?2      ;load the integer into HL
```

Storing is done by

```
        MOV     A,L     ;put low-order byte of expression into A
        STA     ?1      ; and store the character
        SHLD    ?2      ;store the integer
```

Register Variables

Local variables whose storage class is **register** are held in special "register" areas defined in the compiler support library. These are five two-byte variables called **r?1?** through **r?5?**. They are considered register variables because of the speed and small amount of code needed to load or store them. The code to load them is simply

```
        LHLD    r?1?    ;load a register variable
```

and storing is

```
        SHLD    r?1?    ;store a register variable
```

The function which called this function may be using the "registers" also, and those variables must be available upon returning from this function. To handle this, the "registers" are saved by pushing them onto the stack when the function is entered and then restored by popping them back off the stack when

leaving the function.

Global Variables

Global variables have storage reserved in the program but outside of any function. Their name in the assembler output is just their C name (or at least the first 8 characters) with a question mark appended to keep the name from colliding with assembler reserved words. Thus the code to reserve space for an integer variable named **xi**, load it, and store it is

```
        PUBLIC  xi?     ;make xi known outside this file
xi?:    DW      0       ;define a global integer variable xi
        LHLD    xi?     ;load the global xi
        SHLD    xi?     ;store the global xi
```

For a character variable (**xc**), the code is

```
        PUBLIC  xc?     ;make xc known outside this file
xc?:    DB      0       ;define a global character variable xc
        LDA     xc?     ;load xc into A
        CALL    ?sxt    ; and sign extend into HL
        MOV     A,L     ;load low-order byte of expression into A
        STA     xc?     ; and store into xc
```

If the global variable is defined with storage class **static**, the **PUBLIC** directive is not generated which makes this name known only within the current file.

Register Usage

As any expression is evaluated the code generated normally places the current value of the expression in the primary register. Unary operators expect their operand to be in the primary register, and they place the result of applying the operand in the primary register.

Binary operators load their left operand in the primary register. If the right operand is a simple scalar variable (meaning no subscripts or other higher priority operator expressions), the left operand will be switched to the secondary register and the right operand loaded in the primary register. Then a compiler support library routine is called to perform the operation. The result is placed in the primary register for any lower priority operator which might be using this expression as its operand.

When the right operand is an expression, the left operand is pushed onto the stack to save it while the right operand expression is being evaluated. When the right operand has been evaluated its value is in the primary register. Now the left operand is popped into the secondary register and the library routine which performs this operation is called as before.

All of the compiler support routines work this way. Unary operators expect their operand in the primary register and place the result in the

primary register. Binary operators place their right operand in the primary register and their left operand either on the stack or in the secondary register. Each binary operator library routine has two entry points. One entry pops the left operand into the secondary register; the other expects it to be there already.

The statement parsing routines in CC3.C and the expression parsing routines in CC4.C call on a number of functions in CC6.C and CC7.C to do the actual assembler code generation.

6.11 Code Optimization

Q/C uses a number of techniques to improve the speed and shorten the length of the code generated. The five main areas are stack space management, logical tests, register usage, recognition of special cases, and peephole optimization.

Stack Space Management

The allocation and freeing of stack space for local variables is simplified by the Q/C requirement that all local variables be declared at the beginning of a function. This allows the compiler to parse all local declarations and determine how much local space is needed. It then generates a call to a library entry routine which reserves the space for all auto variables, preserves the Q/C "registers" if they are needed, preserves the calling routine's stack frame pointer, and sets the new stack frame pointer.

Since all the locals are declared at one time, the compiler always frees the same amount of space when you return from a function. This allows the compiler to generate a single call to a library exit routine at the end of the function. All **return** statements jump to this common code.

Logical Tests

Whenever a C statement such as **if** or **while** requires that an expression be tested, the expression is tested to see if it is true (non-zero) or false (zero). Since the result of an expression is placed in the primary register, this test can be done with the code

```
MOV     A,H
ORA     L
JZ      falselabel
```

The test in assembler code is based on the setting of the Z flag. If Z is on, the result is zero or false. If it is off, the result is non-zero or true. A very simple example is the C statement

```
if (i > 5)
        function();
```

which calls **function** if i is greater than 5 and does nothing otherwise. This

could be translated as

```
        LHLD    i
        XCHG
        LXI     H,5
        CALL    ?gt     ;test DE > HL and set HL to 0 or 1
        MOV     A,H
        ORA     L
        JZ      ?1      ;bypass the call if test was false
        CALL    function
?1:
```

The comparison operators (==, >, etc.) and the logical operators (&&, etc.) all return TRUE (1) or FALSE (0) as their result. In this case the compiler support routine **?gt** is called to see if DE is greater than HL. The test after the call determines whether **?gt** found the expression to be true or false.

To improve code generation, Q/C ensures that all of the operators mentioned above set the Z flag as well as placing the 0 or 1 in the primary register. This allows the code to be shortened to

```
        LHLD    i
        XCHG
        LXI     H,5
        CALL    ?gt
        JZ      ?1
        CALL    function
?1:
```

In general, if one of these operators has just been done, the compiler remembers it and skips the two unnecessary assembler statements **MOV A,H** and **ORA L**.

Register Usage

Q/C typically does not load a variable until it finds out how it is being used. In the example above, the add operation is parsed by the function **heir12**. The discussion of the recursive descent parser in section 6.9 showed that after **heir12** recognizes the partial expression **i +**, it calls the routines which parse higher priority operators to get its right operand. When these routines return to **heir12** and tell it that the right operand is **j**, **heir12** knows that no higher priority operators were found following **j**. This means that it can now generate the code to do **i + j** regardless of what follows **j**. Since **i** does not need to be preserved after the addition is done, the code can be shortened to

```
LHLD    i
XCHG
LHLD    j
DAD     D
```

A further improvement can be made when the right operand is a constant. The expression

```
i + 5
```

could be translated very much like the previous example as

```
LHLD    i
XCHG
LXI     H,5
DAD     D
```

Since addition is commutative, however, it can be computed as either i + 5 or 5 + i (Notice that Kernighan & Ritchie explicitly allows this in Appendix A, section 7). This allows the code to be shortened to

```
LHLD    i
LXI     D,5
DAD     D
```

For operators which are not commutative, like division, the first form including the XCHG instruction must be used to preserve the order of the operands. Thus

```
i / 5
```

must be translated by

```
LHLD    i
XCHG
LXI     H,5
CALL    ?div    ;library routine to do HL = DE / HL
```

since i / 5 is not the same as 5 / i.

Special Cases

We'll look at one example of special cases -- addition and subtraction of a constant <= 3. This is improved over the example using constants given above by using the INX and DCX instructions. The expression

```
i + 1
```

is translated

```
LHLD    i
INX     H
```

All of the above improvements in doing addition also help in accessing the elements of an array since array[subscript] is equivalent to *(array + subscript) and in accessing the members of a structure since the

member's address is the address of the structure plus the offset of the member.

Peephole Optimization

Peephole optimization is a technique which looks at a small portion of the generated assembler code for patterns of instructions which can be replaced by a more efficient group of instructions. This allows the parsing and code generation to be less complicated while still producing reasonably good code.

Normally the generated assembler code is held in a buffer, and this buffer is scanned for patterns to be replaced. Q/C uses a simpler scheme (at least, simpler for the few patterns it is looking for). Whenever an assembler instruction is generated which is the beginning of a pattern, the global variable **peepflag** is set to indicate which pattern. The generated code is held in a special buffer called **peepbuf**. If the following instructions complete the pattern, the replacement assembler code is written on the output file. If an instruction is generated which is not part of the pattern, the instructions in **peepbuf** are written on the output file and compilation continues.

The simplest pattern which Q/C recognizes is multiple unconditional jumps:

```
JMP ?1
JMP ?2
...
```

This pattern may be appear because of the way code is generated for **if...else** and **switch** statements. Since the second and subsequent jumps can never be reached, they are dropped. Notice that this means that jumps which are dropped cannot have a label. Otherwise they might be reached from somewhere else in the program.

This is generally true of any pattern. There cannot be any labels in the middle of the pattern, or the entire pattern might not be executed every time. Then the replacement pattern would not do the same thing as the original code.

The second pattern typically arises from C code like this

```
x = y;
if ( x ... )
```

where a global, static or register variable is referenced in successive lines. In the pattern

```
SHLD    name
LHLD    name
```

the LHLD instruction can be dropped because **name** is already in HL.

The third pattern typically comes from C code like this

```
if ( test )
    break/continue/return
```

This generates the assembler code

```
        ( test )
        JZ      ?1
        JMP     ?2
ccl:    ...
```

This can be improved by reversing the test and doing the jump for the **break**, **continue**, or **return** statement directly. The replacement pattern is

```
        ( test )
        JNZ     ?2
```

The constant stack frame pointer makes it easier to access the first **auto** variable defined in a function. Since BC contains the address of the beginning of the local variable space, the peephole optimizer can often improve the generated code significantly.

For example, if a function starts like this:

```
func()
{       char array[80];
        register char *p;
```

then an assignment which generates the code

```
;   p = array;
        LXI     H,0
        DAD     B
        SHLD    r?1?
```

becomes

```
        MOV     L,B
        MOV     H,C
        SHLD    r?1?
```

Also, if **array** is used as an argument to a function the code

```
;   puts(array);
        LXI     H,0
        DAD     B
        PUSH    H
        CALL    puts?
```

becomes

```
        PUSH    B
        CALL    puts?
```

Appendix A

How Q/C Differs from Standard C

Appendix A in Kernighan & Ritchie's book The C Programming Language is the official reference manual for the C language. Besides giving a complete description of the C language, it also describes the implementation dependent details of various C compilers.

Since Q/C is a proper subset of standard C, Appendix A of Kernighan & Ritchie describes Q/C as well. A few features are missing, however, so this appendix is a supplement to Appendix A in Kernighan & Ritchie. The missing features are listed here, and implementation dependent details of Q/C are documented. The section numbering is the same as Kernighan & Ritchie, but only those sections which are different are included here.

Here is a summary of the major differences between Q/C and standard C. The current release of Q/C does not support:

1. variable types **long**, **float** and **double**

2. parameterized **#define** commands

3. initialization of **auto** or **register** variables

4. local declarations in compound statements

5. bit fields

1. Introduction

Q/C is Quality Computer Systems' implementation of C for 8080/Z80 CP/M systems. It compiles C programs into assembler language for input to Digital Research's RMAC assembler, Microsoft's MACRO-80 (M80) assembler, and The Code Works' CWA Z80 assembler.

2. Identifiers (Names)

The restrictions on external names (function names and external variables) are:

RMAC	6 characters, 1 case
M80	6 characters, 1 case
CWA	6 characters, 1 case

The CWA assembler can be reset to recognize up to 8 characters and to distinguish between upper and lower case.

2.3 Keywords

The following names are reserved as keywords:

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

If you declare one of these as a variable name, use one as a statement label, or redefine one with **#define** you will get an error message.

2.4.2 Explicit long constants

No longs.

2.4.4 Floating constants

No floats.

2.6 Hardware characteristics

Currently Q/C on the 8080/Z80 uses the following sizes:

char	ASCII 8 bits
int	16 bits
short	16 bits

4. What's in a name?

Q/C recognizes all four storage classes, but it requires all local variables to be declared at the beginning of a function. Thus, the concept of a block does not exist. All **automatic**, **static**, and **register** variables declared in a function exist throughout that function.

Types **long**, **float**, and **double** are not implemented. This means that all arithmetic types are integral types.

6.1 Characters and integers

Characters are always sign-extended when they are converted to integers, so they act the same as the description for the PDP-11. For example, the value of a character variable is in the range -128 to +127.

6.2 Float and double

No floating types.

6.3 Floating and integral

No floating types.

6.6 Arithmetic conversions

The "usual arithmetic conversions" reduce to:

> First, any operands of type **char** are converted to **int**.
> Then, if either operand is **unsigned**, the other is converted to **unsigned** and that is the type of the result.
> Otherwise, both operands must be **int**, and that is the type of the result.

7. Expressions

The result of division by zero and mod by zero is zero.

7.1 Primary expressions

All constants are of type **int**.

Argument passing is handled just as described except that there are no floats or doubles.

7.5 Shift operators

Right shifts are logical (0-fill) if **E1** is **unsigned**. Otherwise, right shifts are arithmetic (sign bit is copied into vacated bits).

8.1 Storage class specifiers

Formal parameters and local variables may be declared as **register**, but only variables declared as **int** or as pointers will be put in registers. Also, only the first five **register** declarations in a function will be effective.

8.2 Type specifiers

Types **float**, and **double** are not available. If you declare a variable to be **long int**, you will get a warning, and it will be compiled as a 16 bit **int**.

8.3 Declarators

The only restriction here is that when you declare a function there must be only one set of parentheses showing the location of an argument list, and it must appear last. This means, for example, that the declaration

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

will successfully declare **apfpi** to be a 3 by 3 array of pointers to functions returning pointers to integers. On the other hand, the declaration

```
int (*fpfi())();
```

will fail to declare **fpfi** to be a function returning a pointer to a function which returns an integer. In general, you cannot declare a function which returns a pointer to a function or a pointer to an array.

8.5 Structure and union declarations

Bit fields are not implemented.

8.6 Initialization

External and **static** variables can be initialized when they are declared. If you do not explicitly initialize them, they will be set to zero if they are less than 129 bytes long. This restriction keeps the size of .COM files manageable. If you want large arrays to start at zero, use the compiler switch **-I**.

You cannot initialize **auto** and **register** variables. Since initialization is done each time the function is entered, you can get the same effect by writing assignment statements at the beginning of the function.

8.7 Type names

Type names are available with the same restriction on functions described in Section 8.3 "Declarators."

9.2 Compound statement, or block

Local variables may be declared only at the beginning of a function, not at the beginning of any compound statement. This means that the concept of a block does not exist in Q/C. The syntax of compound statements reduces to:

```
compound-statement:
        {statement-list_opt}

statement-list:
        statement
        statement statement-list
```

Since local variables cannot be declared, no initialization is needed.

9.7 Switch statement

No check is made to insure that the cases are unique. If there is more than one statement with the same **case** value, the last one will be selected. The **default** case is checked to ensure that there is only one, however.

9.10 Return statement

Since all functions in Q/C return a two-byte value, a character variable will be sign-extended to an integer if it appears as the expression in a **return** statement.

10.1 External function definitions

The function-declarator must meet the restriction given in Section 8.3 "Declarators." Basically, this means that a function cannot return a pointer to a function or to an array.

11.1 Lexical scope

Local identifiers may be declared only at the beginning of the block constituting a function.

12.1 Token replacement

The parameterized **#define** is not implemented.

12.2 File inclusion

filename must be a CP/M unambiguous file name. As usual, it may include a drive name.

12.5 Assembler Code Inclusion (new section not in Kernighan & Ritchie)

Q/C allows you to include assembler code in your C program by surrounding it with the **#asm** and **#endasm** preprocessor commands. Anything between these commands is copied directly to the output file. The only change is that assembler comments are stripped to reduce the size of the output file. You can use the **#include** command inside of this construction, but **#define** commands will not be recognized, and no text replacement will take place.

#asm has an optional argument "8080" which allows you to embed 8080 assembler code in your program and have it work correctly with either the 8080 or Z80 version of Q/C. When you say **#asm 8080**, the 8080 version simply ignores this argument since 8080 code is being generated anyway. The Z80 version, however, inserts a .8080 pseudo-op ahead of your code and a .Z80 after it. Notice that this only works with the M80 assembler, not with The Code Works CWA assembler which can only assemble Zilog mnemonics.

The assembler code is essentially invisible to the compiler. If you want to use assembler code where the compiler is expecting a C statement, you will have to surround it with braces or put a semicolon (null statement) after it. An example is

```
        while (*s++) {  /* send string to port 5 */
#asm
                MOV     A,L
                OUT     5
#endasm
                }
```

This feature has two uses. You can use it to communicate directly with hardware or machine software, and you can write heavily-used functions in assembler for speed. An example of the second use is the functions **streq** and **astreq** in the compiler. If you are concerned about portability, this feature should not be used.

14.4 Explicit pointer conversions

In this implementation for the 8080/Z80, pointers are represented as 16 bit unsigned integers. There are no alignment requirements for **chars** or **ints**.

15. Constant expressions

Q/C will usually recognize constant expressions in statements and evaluate the constant expression at compile time. To insure that a constant expression will be recognized just enclose it in parentheses. For example

```
return (c + ('a' - 'A'));
```

will compile the same as

```
return (c + 32);
```

16. Portability considerations

In Q/C the order of evaluation of function arguments is left to right. Multi-character constants are assigned to a word left to right.

17. Anachronisms

Q/C does not support any obsolete features. In particular, =op for assignment operators is not recognized. The example given

```
x=-1
```

will assign -1 to **x** rather than decrement it. The other obsolete assignment operators produce an "Invalid expression" message.

Appendix B

Q/C Error Messages

This appendix lists all of the Q/C error messages alphabetically and gives a short explanation and possible causes.

The compiler sometimes has a hard time recovering when it does find an error, so you may get a group of messages for one error. If you see a cluster of messages which doesn't seem to make sense, try correcting the problem indicated in the first message and see if you get a good compile.

Already defined: name

The symbol **name** has already been declared. The declaration is ignored. Notice that an ordinary variable can have the same name as a structure or union tag or member. However, a tag and a member cannot have the same name.

Argument can't be that type

Structures, unions, and functions cannot be passed as an argument. A pointer to any of these types is valid, however.

Can only initialize global and static variables

If this is an **extern** declaration, the variables declared cannot be initialized because no storage is reserved here. Initialization must be done where the variables are defined. This release of Q/C cannot initialize local **auto** and **register** variables.

Can't add pointers

The only thing that can be added to a pointer is a scalar.

Can't be a member

A member of a structure or union cannot be a function or an instance of itself. Notice that a pointer to the type being defined is valid.

Can't close output file

An error has occurred while writing the output file. Most likely the disk is full.

Can't initialize unions

The C language definition does not allow unions to be initialized.

Can't pass structures or unions

Structures and unions cannot be passed as arguments to a function. Q/C assumes you meant to pass a pointer and compiles the argument as if it were preceded by the address operator (&).

Can't subtract pointer from scalar

A pointer cannot be subtracted from a scalar. Subtracting a scalar from a pointer or subtracting a pointer from the same type of pointer are the only allowed combinations.

Can't subtract unlike pointers

Two pointers can be subtracted only if they point to objects of the same type.

Can't open: filename

The file name specified can't be opened. Compilation is ended at this point. The most common cause of this error is misspelling the file name. Other possible causes are running out of memory space to allocate a buffer for the file and trying to create a file on a disk with no more available directory entries.

Can't subscript

You can only subscript a variable if it has been declared as an array or a pointer.

Can't #undef - not defined

The name you are trying to #undef is not currently defined. Either it never appeared in a #define statement or it was undefined in a previous #undef statement.

char cannot hold address

Since **char** variables are only 8 bits long, they cannot hold a 16 bit address. If this were a pointer-to-char, it would be a legal initialization.

else not matched with if

There is no **if** statement active to match with this **else**. You may have forgotten to put braces ({}) around the statements making up the compound statement in the last **if**.

Expected comma

The only punctuation allowed between names is the comma.

Function can't return aggregate

A function can only return a scalar or a pointer to more complicated types.

#if nested too deeply

Q/C currently supports six levels of nested #if preprocessor commands.

Illegal address

You are trying to find the address of something that does not have an address. Only lvalues have addresses. If the lvalue is a **register** variable, you cannot take its address because machine registers normally do not have an address.

Illegal function or declaration

At this level the compiler is expecting either function definitions or external declarations. This is neither a valid symbol name nor a valid declaration specifier.

Illegal symbol name

The compiler is expecting a valid symbol name which starts with a letter or an underscore (_) and consists only of letters, underscores, and numbers.

Illegal use of a keyword: keyword

The keyword shown after the colon is being used improperly. Keywords cannot be used as variable names, statement labels, or **#define** macro names. Notice that the **#define** replacement text can be a keyword however.

Illegal use of label: name

The name shown has been declared a label. The only valid uses of label names are in **goto** statements or as the label on a statement. If this is a statement label, you may have forgotten the colon (:) which must follow the label name.

#include nested too deeply

Q/C currently supports three levels of nested include files.

Inconsistent declaration: name

This external name was previously declared with a different type (like **int** before and **char** now), or it is an entirely different kind of variable (like using an external variable name as the name of a function as well).

Inconsistent use of pointers in conditional expression

In a conditional expression (**?:**), if one result expression is a pointer the other must be a pointer to the same type of object or it must be NULL (0).

Initializer must be constant expression

An initializer must be a constant expression as defined in Appendix A, section 15 of Kernighan and Ritchie.

Invalid expression

The compiler was expecting a variable name or a constant, but it found something else. It may be an invalid name. Names must start with a letter or an underscore (_) and consist only of letters, underscores, and numbers.

Invalid storage class

You have either specified a storage class which is not allowed for external variables (only locals can be **auto** or **register**), or you have specified more than one storage class in the same declaration statement.

Invalid type

You have given an illegal combination of type specifiers in this declaration. The only types allowed are **char**, **short**, **int**, **unsigned**, **short int**, and **unsigned int**.

#line number must be decimal

The constant following #line must be a decimal number.

Line too long

After expanding any **#define** replacement text, the input line is longer than the size of the buffer. The size can be increased by changing the symbolic constant **LINESIZE** in **CSTDDEF.H** and recompiling the compiler.

Macro (#define) pool is full

The space allocated for holding **#define** names and their replacement text is exhausted. The macro pool can be enlarged by running QRESET which is described in section 1.5. If you don't have enough memory for a larger macro pool, you will have to reduce the number or size of your definitions. If you have definitions which are only used in certain parts of the program, it may

be possible to split your program into separately compiled parts with smaller sets of definitions in each part.

Member can't have storage class

When you define a structure the storage class is given to the entire structure. Only the type (like **int**) of the members can be specified. The storage class specification is ignored.

Member has another meaning

This member has all ready been defined a different way. When a member is declared in more than one structure, its type (like **int** or pointer to **char**) and its offset from the beginning of the structure must be the same. Another possibility is that the name being declared has already been defined as a structure or union tag.

Member table full

The space allocated for keeping track of structure members is full. The member table can be enlarged by running QRESET which is described in section 1.5.

Missing apostrophe

The character constant does not end with an apostrophe. The end of the line is considered the end of the constant.

Missing delimiter: " or <

The filename in an **#include** statement must be enclosed in quotes "" or angle brackets <>.

Missing #endif

The end of your program was reached without finding a **#endif** to match the last **#if**, **#ifdef**, or **#ifndef**.

Missing punctuation — assumed present: punctuation

The compiler was expecting the punctuation character shown. The compilation is continued as if the punctuation were given.

Missing quote

The string constant has no ending quote. The end of the line is considered the end of the string. If you are trying to continue the string from one line to another, the last character on the first line must be a backslash \.

Missing semicolon

This statement must end with a **semicolon.**

Missing while in dowhile

The **do ... while** statement is missing its **while** condition.

Multiple default cases

A **default** case has already been defined for this **switch** statement.

Must be constant expression

This is one of the places where C requires a constant expression as defined in Appendix A, section 15. A bad array size is set to 1. A bad **case** value causes the case to be ignored. A **#if** will be compiled as if the constant expression evaluated to zero.

Must be lvalue

The expression requires an lvalue in the location indicated. If this is unclear, review sections 5 and 7 in Appendix A of Kernighan & Ritchie.

Must match a #if command

A **#else** or **#endif** must be preceded by an **#if, #ifdef,** or **#ifndef.**

Name not in argument list

The last symbol found in the argument declarations is a name not included in the argument list.

Need explicit array size

The size of the array must be given when an external array is declared without the storage class **extern.** A size of 1 is assumed.

Negative array size illegal

The size given for an array must be positive. The absolute value of the size specified is used.

No active loop statement

The **continue** statement must appear within one of the loop statements **while, do ... while,** or **for** since it means to start the next time around the loop.

No active switch statement

case statements must be inside a **switch** statement. You may have forgotten the braces ({}) surrounding multiple **case** statements.

No active switches or loops

The **break** statement must occur inside of a **switch** statement or one of the loop statements **while**, **do ... while**, or **for**. Otherwise there is nothing to break out of.

No active switches or loops to delete

This is an internal error in the compiler. It is trying to finish up a **switch** statement or a loop, and the switch/loop table is empty. If you have made changes to the part of the compiler which handles switches or loops, check your logic.

No arrays of functions

Arrays of functions are not allowed. Perhaps you meant to declare an array of pointers to functions.

No entry in case table to delete

This is an internal compiler error. The compiler is finishing a **switch** statement and it finds fewer entries than it expects in the switch case table. If you have changed this part of the compiler, check your logic.

No long integers

This release of Q/C does not support long integers. The variables declared as **long** will be compiled as **int**.

No template

A structure is being declared but you haven't said what it looks like. You must either specify the tag of a previously declared structure or a template defining this type of structure.

Not a function

It looks like you are trying to make a function call, but the type of the expression preceding the left parenthesis is not "function".

Not a pointer

The indirection operator (*) can only be applied to an expression with type pointer. For example, *0x80 is not a legal way to refer to the contents of location 80H because the type of the expression 0x80 is **int**.

Not a structure or union member

The name following the operator -> or . has not been declared as a member of a structure or union.

Not enough table space

There is not enough free memory space available for the compiler to allocate its tables. Decrease the size of the tables using QRESET described in Section 1.5 and try again.

Only aggregates can be initialized this way

The use of nested braces ({}) in an initializer is allowed only when you initialize arrays or structures.

Only one #else allowed

A second **#else** was found with no **#if**, **#ifdef**, or **#ifndef** preceding it.

Out of memory

The compiler found that the stack has overrun the memory allocated for compiler tables. Since memory has already been corrupted in unpredictable ways, you may get other errors. You should reboot CP/M in case it was changed, and then use QRESET described in Section 1.5 to decrease the size of the compiler tables before you try compiling again.

Size unknown

The compiler needs to know how big one of these things is, but it can't tell. One way to get this message is attempting to add an offset to a function. Since a reference to a function is treated as a pointer-to-function, the offset must be scaled by the size of the thing pointed to. The size is assumed to be zero.

String is bigger than array

The string specified to initialize the array is longer that the declared size of the array. You may have forgotten to count the null character ('\0') which terminates the string.

String space full

The space allocated for holding strings (called the literal pool) is full. The literal pool can be enlarged by running QRESET which is described in section 1.5. Q/C dumps the literal pool to the assembler output file each time it finds a new function definition. This means that you can also cure this problem by splitting a function with many string constants into several functions.

Symbol table full

There is no room in the symbol table to add the variable just declared. The easiest way to fix this is to increase the size of the symbol table using the QRESET program described in section 1.5. If you don't have enough memory left to do this, you can split your program up into more source files.

This type is too ornate

The type you are declaring is too complex for Q/C. For example, declaring a pointer with more than 30 levels of indirection will cause this error.

Too many active switches or loops

The compiler holds the label information necessary for doing **break** and **continue** statements in a table. You have nested your **switch** and/or loop statements so deeply that the table has overflowed. Increase the size of the switch/loop table using QRESET described in section 1.5.

Too many command line args

This message will only be given when you run a program. The library function which parses the CP/M command line found more arguments than it could hold in **argv**. If you need this many arguments, increase the dimension of **argv** in **_shell** and **_rshell**, and recompile the library routines.

Too many different types in use

You have exceeded the capacity of the type table. Use QRESET described in Section 1.5 to increase the size of this table.

Too many initializers

You have given more initializers for this variable than its declared size.

Too many switch cases

You have exceeded the number of **case** statements which the compiler can hold in the switch case table. Each **case** statement value counts as one entry even if several appear on a single C statement. When **switch** statements are nested, the switch case table holds an entry for every active **case**. QRESET, described in section 1.5, can be used to increase the size of the switch case table.

Unbalanced braces

The end of the program was reached and the number of opening and closing braces ({}) was not equal.

Unclosed comment

The end of the program was reached without finding an "*/" to close the last comment.

Undeclared tag: name

The tag shown has never been defined. You must specify what the structure or union looks like before you can define one using this tag because Q/C needs to know how big it is. You can define a pointer using a tag before the tag is defined, but the definition of the tag has to appear before the end of the source file if it is a global tag or before the end of the function if it is a local tag.

Undefined label: name

The name shown was used in a **goto** statement in this function, but it never appeared as the label on a statement.

Undefined variable: name

The name shown has never been declared. This may be a spelling error. Q/C will declare **name** to attempt to eliminate further error messages. If **name** is followed by a [, it will be declared as pointer to **int**. Otherwise it will be declared plain **int**.

Unknown #preprocessor command

The line started with a #, but it is not a preprocessor command that Q/C recognizes. You may have spelled it wrong.

Usage:cc infile ... -acdilmortv -sxxxx outfile

You typed the command line wrong. If you can't see what you have done wrong by looking at the general form shown above, review Chapter 2.

Appendix C

Sample Compiler Output

This appendix shows you what the compiler output looks like when you use the -C option to get a fully commented listing. The C program being compiled is:

```
/* sample - demonstrate the -C compiler option */
int externi;
sample()
        {
        static int stati;
        register int regi;
        auto int autoi;

        externi = 0;
        stati = 0;
        regi = 0;
        autoi = 0;
        }
```

The compiler output for this program with the -C option is shown on the next page. The comments on the right side of the page (which are preceded by the arrow <——) are not part of the compiler output. They are added to point out various features.

```
;Compiled by Q/C V3.x
;/* sample - demonstrate the -C compiler option */
;int externi;                    <--- Your C program is shown as comments
        DSEG
        PUBLIC  externi?        <--- Your external name is used with the ?
externi?:       DW      0            added to avoid conflicts with assembler
;sample()                            reserved words
        CSEG
        PUBLIC  sample?
sample?:
;       {
;       static int stati;
        DSEG
?2      DW      0               <--- stati is given the internal name ?2
;       register int regi;
;       auto int autoi;
;       externi = 0;
        CSEG
        CALL    ?ensr           <--- The library routine reserves 2 bytes
        DW      -2                   on the stack for autoi and saves
        LXI     H,0                  the registers
        SHLD    externi?        <--- Your external name is used
;       stati = 0;
        LXI     H,0
        SHLD    ?2      ;stati  <--- Q/C stores your local variable stati
;       regi = 0;
        LXI     H,0
        SHLD    r?1?    ;regi   <--- Q/C stores your register variable regi
;       autoi = 0;
        MOV     L,C
        MOV     H,B     ;autoi  <--- Q/C loads the address of your local
        MVI     M,0                  variable autoi
        INX     H
        MVI     M,0
;       }
        CALL    ?exrs           <--- The library routine frees the 2 bytes
        DW      2                    on the stack and restores registers
        EXTRN   r?1?            <--- Q/C informs the assembler that the
        EXTRN   r?2?                 registers and library routines will
        EXTRN   r?3?                 be found elsewhere
        EXTRN   r?4?
        EXTRN   r?5?
        EXTRN   ?gc,?sxt,?gcs,?g,?gs,?p,?o,?x,?a,?e,?ne,?gt
        EXTRN   ?lt,?le,?ge,?ugt,?ult,?ule,?uge,?asr,?asrl,?asl
        EXTRN   ?asll,?lsr,?lsrl,?s,?neg,?com,?n,?mult,?div
        EXTRN   ?udiv,?sw,?enr,?en,?ensr,?ens,?exr,?exrs,?exs
        END
```

Appendix D

Compiling the Compiler

For all you brave people (masochists?) who want to change the compiler and recompile it, here goes. The first thing you must do is use QRESET (described in Section 1.5) to change the compiler table sizes to the following values:

Symbol table: 250 entries
Literal pool: 300 characters
Macro pool: 1300 characters

Running the EXPAND Program

The source code for the compiler is in the files **CSTDDEF.HX**, **CGLBDEF.CX**, **CGLBDECL.CX**, and the nine files **CC1.CX** through **CC9.CX**. The "X" in the type means that these files are in compressed form and must be eXpanded before they can be used. The expansion program is run like this:

A>EXPAND CC1.CX CC1.C

The first file name **CC1.CX** is expanded and written to the second file name **CC1.C**. Of course, disk drive specifiers are legal in these names.

Several simplifications are possible. If you do not specify an output file, EXPAND will supply a name based on the input file name like this:

*.CX becomes *.C
*.HX becomes *.H
*.MX becomes *.MAC

For example

A>EXPAND CSTDDEF.HX

will create the expanded file CSTDDEF.H. If you specify only an output drive, the created filename will be on that drive. This means that saying

A>EXPAND CGLBDEF.CX B:

will create B:CGLBDEF.C. From this point on, I will only mention the names of the expanded files.

Compiling the Compiler

The first time you will have to compile and assemble all nine pieces. If you save all of the .REL files produced by the assembler, however, you will only need to compile and assemble those parts of the program which actually change in the future. The old and new .REL files are then linked to make a new .COM file.

NOTE: The version of Q/C that comes on your distribution disk was linked with PLINK-II described below. If you link with Digital Research's LINK or Microsoft's L80, CC.COM will be the same size but it will use about 1K more memory at execution time. For the explanation, see the section "Using PLINK-II."

After you've got a new compiler there are several stages of testing you will want to do. First try it on a simple program just to see that it still can compile at all. Then write some programs to test the new features that you have added to the compiler.

When you are satisfied that your changes are doing what you intended, you will want to see that the new compiler is "fertile" -- that it can reproduce itself. Call the original version of the compiler the parent and the new version of the compiler that you just created the child. Now use the child to to compile the compiler once again and create a new version called the grandchild. If everything is still working you can compile the parts of the compiler (CC1.C through CC9.C) one at a time with the child and the grandchild. Use the program COMPARE included on your Q/C disk to compare the two assembler files generated by the child and grandchild and be sure that they are identical. If they are not, start checking to see why the compiler is not generating the same assembler code each time and correct it. When the child and the grandchild both produce identical code, you can safely erase the parent (or perhaps more safely, put it away for a while and then erase it). Notice that you haven't proved that the compiler is 100% correct -- just that it can compile itself correctly.

Now that you have a new fertile compiler, you can change the compiler to actually use the new features. Then you can repeat the whole process and finally end up with a new improved compiler which will do more or better or both. By using as many of the features as possible in the compiler, you also make it a good test program for itself.

No doubt you will come up with your own procedure for compiling the compiler, but the following discussion should help get you started.

If you are using 8" single density disks or 5" disks, you will need two drives. The disk in drive A should have the new compiler source files CC1.C through CC9.C, the header files CSTDIO.H and CSTDDEF.H, the global declaration files CGLBDEF.C and CGLBDECL.C, the old version of CC.COM, your assembler and linker, and the run-time library CRUNLIB.REL. The disk in drive B should be pretty much empty. Further juggling of files will probably be necessary if you are using 5" disks.

Using RMAC

If you use RMAC, start by changing the default assembler to RMAC. This is done by changing the definition of DEFASM in the header file CSTDDEF.H to

```
#define DEFASM 'a'
```

Next compile and assemble each part of the compiler like this

```
A>CC CC1 -AO B:
(compiler messages)
A>RMAC B:CC1 $PZ-S
(assembler messages)
A>ERA B:CC1.MAC
```

This will produce the file CC1.REL on drive B. When you have compiled all the parts you need, you can link the pieces together by saying

```
A>LINK B:CC=B:CC1,B:CC2,B:CC3,B:CC4,B:CC5,B:CC6,B:CC7,B:CC8,B:CC9,&
LINK 1.3
*CRUNLIB[S,A,$IB,$SZ]
(linker messages)
```

giving you a new version of the compiler CC.COM on drive B. The original compiler will still be on drive A. The Additional Memory (A) switch must be used in a 56K CP/M system or LINK runs out of memory space. The $I switch is then used to tell LINK to store its buffers in temporary files on drive B.

Using M80

If you use M80 you cannot link Q/C on a 56K CP/M system. I estimate that it would take at least a 60K system. One way around this is to use PLINK-II which is described in the next section.

Compile and assemble each part of the compiler like this

```
A>CC CC1 -O B:
(compiler messages)
A>M80 =B:CC1
(assembler messages)
A>ERA B:CC1.MAC
```

This will produce the file CC1.REL on drive B.

When all parts are compiled, you link them by saying

```
A>B:
B>A:L80 CC1,CC2,CC3,CC4,CC5,CC6,CC7,CC8,CC9,A:CRUNLIB/S,CC/N/E
(linker messages)
```

giving you a new version of the compiler CC.COM on drive B. The original compiler will still be on drive A.

Using PLINK-II

The PLINK-II linker from Phoenix Software Associates Ltd. has (at least) two advantages over the linkers previously described. It can link any size .COM file up to the limit of 64K, and it allows you to reuse one-time initialization code as part of the free space. The functions in CC9.C are used only when the compiler starts. Using the FREEMEMORY statement places this module after the address which the linker says is the start of free memory space. Then as long as no allocated memory is stored into before the functions in CC9.C finish, the space they occupy can be reclaimed.

I use a file called CC.LNK defined as follows:

```
FILE B:CC1,B:CC2,B:CC3,B:CC4,B:CC5,B:CC6,B:CC7,B:CC8
LIB CRUNLIB
FREEMEMORY
FILE B:CC9
OUT B:CC;
```

Then when all the .REL files are ready, I give the command

```
A>PLINK-II @CC
```

and a new version of Q/C is produced on drive B.

PLINK-II is available from:

Lifeboat Associates
1651 Third Ave.
New York, NY 10028
(212) 860-0300

Effect on QRESET

All of the locations changed by QRESET are global variables defined in CGLBDEF.C. These variables are forced into the code segment by a CSEG directive so they will all be located at the beginning of CC.COM. QRESET finds them by locating the compiler sign-on message which is just ahead of them. As long as you don't change the order or location of these global definitions QRESET should continue to work correctly.

Appendix E

Maintaining the Function Library

This appendix describes the design details of the Q/C function library and then tells you how to build a new library if you want to make changes or additions.

Design of the Q/C Function Library

The design goal for the Q/C function library is to be as close as possible to the standard C library. The entire function library other than the CP/M and MP/M system calls, the non-local jump routines **setjmp** and **longjmp**, and the 8080/Z80 port I/O routines **in** and **out** are written in C. Whenever the compiler is changed to generate better code, the I/O library can be recompiled to take advantage of the improvements.

Some of the library routines are also provided in assembly language. In these cases, the portable C version can be selected by defining the symbolic constant PORTABLE as discussed in the comments at the beginning of the library routines. If you are using The Code Works CWA assembler, you must compile the C versions because CWA can only assemble Zilog mnemonics.

The following discussion starts with the low-level I/O routines which interact directly with CP/M, and continues with the character (buffered) routines. These routines deal only with the low-level library functions, not with CP/M itself. The discussion ends with the routines which use the standard input/output files and how these files are redirected.

Low-level (System) I/O

At this level, everything is done with CP/M system calls using the function **bdosl** in the compiler support library. For each file that is open, a file control block (fcb) is maintained. It contains a 36 byte CP/M fcb which allows use of the CP/M 2.2 random access functions, a one-byte status flag and a two-byte unsigned integer to hold the current last record number.

Q/C uses individual bits in the status flag to indicate the following conditions about the file:

```
#define READ    01      /* open for input */
#define WRITE   02      /* open for output */
#define APPEND  04      /* open for output and append new data */
#define BINARY  010     /* don't treat data as text */
#define BUF     020     /* a buffer has been allocated */
#define USERBUF 040     /* a user-supplied buffer is being used */
#define FEOF    0100    /* an input file has reached end-of-file */
#define FERR    0200    /* an error has occurred on this file */
```

Recording the last record number allows the **read** function to know where the end-of-file is even before you close a file opened for reading and writing.

Space for the Q/C fcb's is obtained by calling the library function **sbrk**. The pointer returned from **sbrk** is stored in an array of pointers to fcb structures like this

```
#define NFILES  10      /* you can change this number */
struct fcb {
        char    flag;
        unsigned file_size;
        struct cpmfcb {
                char    drive;
                char    filename[8];
                char    filetype[3];
                char    extent;
                int     reserved;
                char    rec_cnt;
                char    dir[16];
                char    curr_rec;
                unsigned rand_rec;
                char    overflow;
                } cpm_fcb;
        } *_fcb[NFILES];
```

The name **_fcb** is one that C programs do not normally need to be aware of, so it starts with "_" to avoid colliding with your external names.

Just as in standard C, when you **open** or **creat** a file you get back a file descriptor (fd) which is a small positive integer. These numbers range from 6 to NFILES + 5. This is done to avoid the UNIX fd's 0 (**stdin**), 1 (**stdout**), and 2 (**stderr**), and also fd 5 which is used for the CP/M LST: device (the printer). So, if the library is compiled with NFILES = 10 allowing ten files to be open at once, the fd's will range from 6 to 15. The fcb associated with each file will begin at the address in **_fcb[fd - 6]**. Your program does not need to be aware of anything but the file descriptor, of course, since it is what you use to tell **read**, **write**, **seekr**, and **close** which file to work with.

Character (Buffered) I/O

At this level the I/O is done with individual characters or strings of characters terminated as usual with a null ('\0') character. These routines

all use the low-level functions **open**, **creat**, **read**, **write**, and **close** to do their work, so they are nearly independent of CP/M.

The main dependencies are that they recognize the CP/M end-of-file character 0x1A (^Z), and they change CR/LF character combinations indicating CP/M end of line to the C convention of a single newline character '\n' (0x0A) at the end of each line. This allows you to write your Q/C programs using the standard C conventions and still be compatible with CP/M text file conventions so you can edit files produced by a Q/C program, for example. This tampering can be eliminated by opening the files for binary I/O.

Other dependencies are that the CP/M CONBUF function #10 is used by **gets** and **fgets** when you read from **stdin**. Also, you can access the CP/M LST: device (the printer) by opening "lst:" with **fopen**.

To allow your program to read and write any number of characters and still do the actual I/O in 128 byte records, the character I/O routines hold the data in buffers. The size of these buffers is controlled by the symbolic parameter NSECTS in the disk I/O library. It is just the number of CP/M 128 byte records (disk sectors) that the character I/O routines will buffer and will read or write at one time. In fact, to increase the efficiency of disk I/O, you will typically want to set NSECTS equal to 4 or 8. This is particularly helpful if your input and output files are on the same drive since this will minimize the number of times the read/write head must move back and forth.

To allow the character I/O functions to manage these buffers, the library has an I/O block (iob) for each buffered file. The definitions for using buffered files appear in the standard header file QSTDIO.H:

```
#define FILE    struct _iob

struct _iob {
        char    _flag;      /* status flag for this file */
        char    *_pch;      /* pointer to next character in buffer */
        int     _cnt;       /* number of bytes left in buffer */
        char    *_buf;      /* pointer to buffer for this file */
        int     _bufsize;   /* size of buffer for this file */
        char    _fd;        /* the file descriptor for this file */
        };
```

Q/C holds the I/O blocks in an array of structures defined as

```
FILE _iob[NFILES];
```

When you open a file for buffered I/O by calling **fopen** you get back a file pointer (fp) which is actually a pointer to (the address of) the iob being used for this file. **fopen** calls **open** or **creat** to open the file for input or output and saves the fd in the iob for the other character I/O functions. The other fields in the iob are set to their initial values. This can all be seen in the listing of **fopen**.

The buffer is not allocated until the first time the file is read or written. Until this time, you can supply your own buffer by using the library functions **setbuf** and **setbsize**. If you do not supply a buffer, the storage allocator **sbrk** is called to obtain space for the buffer.

When you call routines like **getc** and **fgets** you are actually being handed characters from the buffer most of the time. These routines change the pointer to the next character in the buffer and decrease the count of characters remaining each time. When the buffer is empty, it gets filled by calling the low-level routine **read** which reads the required number (NSECTS) of CP/M 128 byte records directly into the buffer.

The output routines **putc** and **fputs** work in a similar fashion. When the buffer for an output file is filled, they call **write** to write the buffer to disk. Look at the library programs to see the details of how all this is accomplished.

The one big difference in handling input and output files comes at close time. In both cases, **fclose** zeros all the bits in the status flag except "buffer allocated" to indicate that this iob is no longer in use. This frees the iob so that another file can be opened. For an input file, this is all that is done.

If the file is open for output, however, the last buffer must be written, and a CP/M close must be done to record the fcb information permanently on the disk. If the file is open for normal (text) output, **fclose** first adds a CP/M EOF character 0x1A (^Z) to the end of the data in the buffer. For text or binary files, it then calls **fflush**. **fflush** figures out how many CP/M records need to be written to flush out the partially filled buffer, and then calls **write** to write them. **fclose** then calls **close** to do the CP/M close.

Standard I/O Files and Redirection

Under UNIX, the shell handles the redirection of files, but no comparable capability exists in CP/M. In Q/C there is a library routine called **_rshell** which is called before your **main** function. It parses the CP/M command line to build **argc** and **argv**, and it also looks for instructions to redirect **stdin** and **stdout**. If you request redirection, **_rshell** will open the files you specified and set **stdin** and **stdout** to the file pointers returned by **fopen**.

The files **stdin**, **stdout**, and **stderr** are normally associated with the CP/M CON: device which is your terminal. You can call any of the buffered I/O routines with the names **stdin**, **stdout**, or **stderr** and the appropriate file will be used. In fact, the functions which do I/O using the standard I/O files are defined in terms of the buffered I/O functions. For example, **putchar** looks like this:

```
putchar(c)
        {
        return putc(c, stdout);
        }
```

Q/C adds the ability to write to the CP/M LST: device which is normally your printer. You can use the buffered I/O output routines by opening **lst:** like this

```
FILE *fplst, *fopen();
...
fplst = fopen("lst:", "w");
```

and using **fplst** as the file pointer to the other functions.

Also, command line redirection recognizes **lst:** as a request to direct the standard output file **stdout** to the printer. For example, if **copy** copies **stdin** to **stdout**, you can print the disk file USERMAN.TXT by typing the underlined command

```
A>COPY <USERMAN.TXT >LST:
```

Rebuilding the Function Library

If you make changes to the function library you will have to recompile and assemble it to get **CRUNLIB.REL**. The source for the function library is in the four files **CDISKLIB.CX**, **CUTILIB.CX**, **CASMLIB.CX**, and **CRUNTIME.MX**. These files are all compressed and must be expanded as described in the section "Running the EXPAND Program" in Appendix D. **CDISKLIB.C** contains all of the I/O and memory allocation functions. **CUTILIB.C** contains the string and character handling functions. **CASMLIB.C** contains the functions which are provided only in assembly language and **CRUNTIME.MAC** contains the compiler support routines. Because of the large number of files involved, you will probably want to have a separate disk with little or nothing else on it to hold all the intermediate assembler and .REL files.

CDISKLIB.C, **CUTILIB.C** and **CASMLIB.C** must be compiled with the Q/C library generation option (-L). This option compiles all global definitions at the beginning of the C source program into the normal output file. Each C function is compiled into a separate assembler file named function.ext where **function** is the C function name and .ext is the file extension appropriate for your assembler. There is one peculiarity in this process. CP/M does not accept the underscore (_) character in its file names, and M80 will not accept file names with any special characters in them. Because of this, Q/C translates the underscore character to the digit 1 when it creates the assembler file. As an example, the library file **_exit** will be placed in the file called **1EXIT.MAC** for M80 and CWA or **1EXIT.ASM** for RMAC. These files must all be assembled separately.

Two important considerations when building the library are:

(1) putting the global variables in a place where they will always be loaded, and

(2) ordering the functions in the library so all references are forward references.

The first item is handled as follows. All global variables used by the library functions are defined in **CDISKLIB.C**. Assuming your source files are in drive A and your work disk is in drive B, compile **CDISKLIB.C** like this

```
A>CC CDISKLIB -LO B:STDIN
```

This causes the global variables to be placed in **B:STDIN.MAC** or **B:STDIN.ASM**. You assemble this with the command

```
A>M80 =B:STDIN
or
A>RMAC B:STDIN $PZ-S
```

giving **B:STDIN.REL**. Since this module is included in **CRUNLIB.REL** as **STDIN**, any program which references **stdin** will cause this module to be loaded. Including **QSTDIO.H** in all your programs thus insures that the library global variables will be loaded.

The compiler support routines must then be assembled. If you are using M80, simply say:

```
A>M80 =CRUNTIME
```

If you are using RMAC, you should first change the EQUate at the beginning of **CRUNTIME.MAC** to read:

```
RMAC    EQU     TRUE
```

Then assemble this file by saying:

```
A>RMAC CRUNTIME.MAC $PZ-S
```

In either case you will end up with the file **CRUNTIME.REL**.

IMPORTANT NOTE FOR RMAC USERS:

Once you rebuild the library using RMAC, it is NOT usable with L80 or PLINK-II. RMAC cannot assemble the M80 end-of-memory symbol $MEMRY, so the Q/C memory allocator will no longer work.

To make all references in the library forward references, the individual modules must be ordered as follows:

1. First, modules from **CDISKLIB.C:** lRSHELL, lSHELL, CANTOPEN, EXIT, GETS, PUTS, SCANF, FSCANF, lSCAN, QPRINTF, PRINTF, FPRINTF, GETCHAR, PUTCHAR, FOPEN, FCLOSE, FGETS, FPUTS, FREAD, FWRITE, GETW, PUTW, GETC, lFILL, UNGETC, PUTC, FFLUSH, lCHKBUF, SETBUF, SETBSIZE, FEOF, FERROR, CLEARERR, FILENO, OPEN, CREAT, READ, WRITE, CLOSE, SEEKR, TELLR, UNLINK, lGFD, lGFCB, lEXIT, MAKFCB, CALLOC, MALLOC, FREE, SBRK, MAXSBRK, MOAT, GETKEY, STDIN.

2. Second, modules from **CUTILIB.C:** SSCANF, SPRINTF, lCAT, lFMT, ITOB, ATOI, lATOI, STRCAT, STRCPY, STRNCAT, STRNCPY, STRMOV, STRLEN, STRCMP, STRNCMP, CHUPPER, CHLOWER, ISPUNCT, ISCNTRL, ISALNUM, ISALPHA, ISUPPER, ISLOWER, ISDIGIT, ISSPACE, ISASCII, ISPRINT, INDEX, RINDEX, TOUPPER, TOLOWER, IMIN, IMAX, PEEK, POKE, WPEEK, WPOKE.

3. Third, modules from **CASMLIB:** IN, OUT, SETJMP, LONGJMP, BDOS, BDOS1, MPM.

4. Last, from **CRUNTIME.MAC:** everything is pulled in at once as one module called "CRUNTIME".

This ordering ensures that the linkers will load all functions needed with a single pass through the function library.

Once you have all the individual .REL files, the library is built using the Microsoft or Digital Research LIB program or the The Code Works CWLIB program. All of the modules are simply included in the new library in the order they appear in the three lists above.

Since the library is already set up, any maintenance you do will typically involve only one or a few functions. In this case you can build the new library from the old one and just substitute the new modules in the correct location. As an example of how this is done, suppose you are changing the functions **fprintf** and **putc** which are in **CDISKLIB.C.** It is easiest to compile the entire library and then assemble only the functions being replaced. Assuming that the source files are on drive A and the work disk is on drive B, give the commands

```
A>CC CDISKLIB -LO B:STDIN
  (compiler messages)
A>M80                          or      A>RMAC B:PRINTF $PZ-S
*=B:FPRINTF                              (assembler messages)
No Fatal error(s)                      A>RMAC B:PUTC $PZ-S
*=B:PUTC                                 (assembler messages)
No Fatal error(s)                      A>
*^C
A>
```

Now you can build your new library for M80 on drive B, and leave the

original library unchanged on drive A by saying

```
A>REN CRUNOLD.REL=CRUNLIB.REL
A>LIB
*B:CRUNLIB=CRUNOLD<1RSHEL..PRINTF>,B:FPRINTF
*CRUNOLD<GETCHA..UNGETC>,B:PUTC,CRUNOLD<FFLUSH..CRUNTI>/E
A>
```

or for RMAC

```
A>LIB B:CRUNLIB=CRUNLIB(1RSHEL-PRINTF),B:FPRINTF
A>LIB B:CRUNLIB=B:CRUNLIB,CRUNLIB(GETCHA-UNGETC),B:PUTC
A>LIB B:CRUNLIB=B:CRUNLIB,CRUNLIB(FFLUSH-CRUNTI)
```

You have included all the modules from the original library except **FPRINTF** and **PUTC.** These were loaded from the .REL files you just created on drive B. Notice that only the first six characters of a module name are retained, so you must specify only the first six characters of long module names. In the example the module names **1RSHELL, GETCHAR,** and **CRUNTIME** were shortened to **1RSHEL, GETCHA,** AND **CRUNTI.** On the other hand **B:FPRINTF.REL** is a CP/M file name so it is given in full.

For more information on doing library maintenance, see the manual for the assembler and library manager you are using.

Appendix F

Q/C on CP/M-Compatible Systems

As I stated in Chapter 1, Q/C should run on CP/M-compatible systems, but this is not guaranteed. If you have problems, I will give you what assistance I can. Since the vast majority of users have CP/M, however, this is where the main support will be.

When you run Q/C on a CP/M-compatible system, the problems stem from the way values are returned from operating system calls. When a BDOS service is needed, the function number is loaded in in the C register and any additional parameter is loaded in DE. Then a CALL is done to location 5H.

Unfortunately, the CP/M Interface Guide does not clearly specify how values are returned. For CP/M 2.2 it says that single byte values are returned in A and double byte values are returned in HL. For compatibility with earlier versions, CP/M 2.2 returns L=A and H=B. It does not state what is in B when the value returned is a single byte, however. On the CP/M systems I have seen, B=H=0 for single byte returns. On CP/M-compatible systems this is not always true.

Q/C puts its function return values in HL. To satisfy everybody it provides two functions to do CP/M BDOS calls. **bdosl** takes the single byte returned by CP/M in the A register and moves it to L. It then loads zero in the H register. **bdos** simply returns the two byte value which CP/M puts in the HL register pair. All BDOS calls in the Q/C library expect single byte return values, so they use the **bdosl** function.

Although this technique makes Q/C and QRESET work on other systems, you may still have a problem with the programs you write. If you make any direct BDOS calls using the Q/C library routines, be sure to call **bdosl** when you expect a single byte return value and **bdos** when you expect a double byte.