Supersoft C Compiler

Table Of Contents

CHAPTER 1

Introduction

The SuperSoft C Compiler is a self-compiling, optimizing, compiler generating a final output file in assembly source code. It accepts essentially the full C language, with the few exceptions detailed in Appendix A of this manual. Under CP/M, the compiler requires two passes; under CP/M-86 and MS-DOS, the optimization and code generation are split into individual phases, effectively making it a three-pass compiler. SuperSoft C is adaptable for a variety of operating systems and central processing units (CPUs). See Appendix B for a list of the currently available machine and operating system configurations. Due to the inherent portability of the C language and its particular implementation in our compiler, configurations for other operating systems and machines can be easily and rapidly developed.

A wealth of user callable functions are supplied with SuperSoft C. These include many UNIX compatible functions, allowing the porting of source between UNIX C and SuperSoft C with few if any source changes. A full standard I/O package (STDIO) is provided. This package allows file I/O that is independent of the natural record size of the target system. For instance, under CP/M, a SuperSoft C program may open a file, seek to any byte in the program, and then close the file. The appropriate byte and only the appropriate byte will be changed by this sequence of operations. Beyond that, porting services are available from SuperSoft.

Writing our compiler in the language it implements not only has facilitated its development and maintenance but also has provided the most direct means of testing it. It has undergone many other extensive tests including the fact that it is used in most of our programming projects. As a result, SuperSoft C has been tested on tens of thousands of unique lines of C source code.

As a result of the optimizations performed, the code generated by the compiler has spatial and temporal

efficiency adequate for system- as well as application-level programming. One measure of its efficiency is that for at least one operating system, CP/M, with the exception of a ten line procedure in assembly code required as a link to the operating system, the entire I/O interface of the compiler is written in SuperSoft C.

Design decisions made with regard to code generation allow the SuperSoft C Compiler to generate rather good code for subroutine entry and exit. These decisions allow us to have true register variables even on machines with few registers. (Specifically, on the 8088 series SuperSoft C uses the BC register pair as a true register variable.)

MODULARITY

The fact that each pass of the SuperSoft C Compiler is modular confers certain important advantages. Dividing the compiler into separate self-contained programs allows it to run with a relatively small amount of available memory and still support most of the C language. This modular structure also leads to a clean interface between the first pass, or Parser (CC), and the optional optimization pass (COD2COD) and between the optimization pass and the final pass, or Code Generator (C2 or C2I86). The output file of one pass simply becomes the input file of the next. Modularity also facilitates adapting the compiler to other machines and operating systems, since the first module is machine- and system-independent and only portions of the code generator need be changed.

INTERMEDIATE OUTPUT: U-CODE

The compiler's first pass, CC, accepts as input a SuperSoft C source code file and parses it. As output it generates a file in an intermediate code known as Universal code or U-code (implying code not specific to any one system or machine). Since one of the design specifications of our compiler was that the output of all of its passes be intelligible to a human being, U-code may be viewed and modified using an ordinary text editor.

A machine-independent, U-code to U-code optimizer is supplied with your compiler. Under CP/M-86 and MS-DOS, the optimizing process takes place in a separate pass (COD2COD); under CP/M, this process occurs during the code generation pass (C2). The optimizer accepts as input the U-code file generated by CC. The input file undergoes a complex

optimization process involving global code rearrangement within functions as well as reiterative local code transformations within a "peephole" or window of a certain number of lines of code. The code generation process (C2 or C2I86) produces a final output file in assembly language.

FINAL OUTPUT: ASSEMBLY CODE

Several benefits result from the choice of assembly code as the final output of the compiler, some of particular value to the system level programmer. Since the code generated is intentionally not specific to any one of the assemblers in use on a given machine, practically any hardware-compatible assembler (including absolute as well as relocating assemblers) may be used. Thus the output of this compiler can be integrated into any software system already written in, or compatible with, assembly source code. The programmer can also insert lines of assembly code directly into a C source file by bracketing them with the compiler directives #ASM and #ENDASM. Lines so inserted are not optimized or altered in any way by the compiler.

Use of assembly source code also satisfies our design specification requiring that the output of each pass be intelligible to a human being. The resulting readability of the compiler's output facilitates debugging and allows any user to see the kind of code generated by the compiler. Thus the programmer need not take for granted that the code generated is what was desired and may either alter the source code or "hand polish" the generated code to make the result more suitable for a particularly demanding application.

EXECUTION TIME VS. COMPILATION TIME

A major trade-off in the design of any compiler is in the time required to compile a program versus the time required to execute it. Since one of our primary design goals was to generate code efficient enough for system level programming, we have emphasized speed of execution at the expense of speed of compilation. (Certain optimizations performed by the code optimizer require time proportional to the square of the size--number of U-code instructions--of the largest C function to be optimized.) Optimization can be turned off for faster compilation. This emphasis, while increasing the turn-around time during program development, does make our compiler useful for a far broader range of programming tasks. The SuperSoft C Compiler is unique in that it allows you to do efficient system level programming with structure and clarity on a relatively small hardware system.

RECOMMENDED READING

The standard reference for the C language is:

> Brian W. Kernighan, and Dennis M. Ritchie, The C Programming Language (Englewood Cliffs, NJ: Prentice-Hall Inc.), 1978.

The programming manual that Dennis Ritchie, the chief designer of C, and Brian Kernighan have written is a well-conceived and readable introduction to C and includes the C Reference Manual as an appendix. It is indispensable to any would-be C programmer. An article which touches on the evolution and philosophy of the C language, is:

> D. M. Ritchie, et al., "The C Programming Language," The Bell System Technical Journal, 57(6) (July-August 1978), 1991-2019.

A tutorial on C is:

> Thomas Plum, Learning to Program in C (Cardiff, NJ: Plum Hall, 1983).

Also useful are:

> Alan R. Seuer, The C Puzzle Book (Englewood Cliffs, NJ: Prentice-Hall Inc., 1982).
>
> Jean Yates and Rebecca Thomas, A User Guide to the UNIX System (Berkeley, CA: OSBORNE/McGraw-Hill, 1982).
>
> Ann and Nico Lomuto, A Unix Primer (Englewood Cliffs, NJ: Prentice-Hall Inc., 1983).

ORGANIZATION OF THIS MANUAL

The chapters that follow provide the information you will need to use the SuperSoft C Compiler. Chapter 2 presents instructions for invoking the compiler and the preprocessor directives. Descriptions of the standard library functions supplied with the compiler are given in Chapter 3. Chapter 4 describes how to insert code into the code generator's run-time library. The differences between SuperSoft C and Standard Version 7 UNIX C are listed in Appendix A. The currently available machine and operating system configurations of the SuperSoft C Compiler are listed in Appendix B. Appendix C describes some common problems and their solutions. Appendix D consists of a list of supplied functions organized according to the file in which they reside. Appendix E discusses the use of overlayed programs. The program's error messages are listed in Appendix F. Appendix G describes the Double Floating Point Functions; Appendix H, the Long Integer Functions.

We hope that the SuperSoft C Compiler becomes one of your most useful programming tools. We welcome any comments you may have about the compiler or its documentation.

CHAPTER 2

Using the SuperSoft C Compiler

TOPICS DISCUSSED

This chapter presents the information you will need to make the best use of the SuperSoft C Compiler. The topics discussed are: the files supplied on the compiler disk, which of those files you will need to compile your programs, other software you will need, the syntax of the compiler's command lines, the use of our compiler in a specific operating system environment, the command line options available for each pass, and the compiler preprocessor directives supported in SuperSoft C.

FOR YOUR OWN PROTECTION, make a copy of your SuperSoft C Compiler disk when you first receive it. Store the original in a safe place as a master disk and for all subsequent work use only the copy.

FILES SUPPLIED...

...UNDER THE CP/M-80 OPERATING SYSTEM

If you have the CP/M-80 version of the compiler, you should find the following files on your working copy:

```
CC.COM      :  first pass of the compiler: the parser
C2.COM      :  second pass: Code Optimizer/Generator
C2.RH       :  run-time library header file used for ASM
C2.RT       :  run-time trailer file used for ASM
C2.RTM      :  run-time trailer for RMAC and M80
C2PRE.ASM   :  run time header source for L80
C2POST.ASM  :  run time trailer source for L80
C2PRE.REL   :  run time header for L80
C2POST.REL  :  run time trailer for L80
*.ASM       :  various other run-time sources
MDEP.C      :  #ASM/#ENDASM version of run-time
C.SUB       :  CP/M SUBMIT file for compilation
CBRACK.H    :  upper case defines for keyboards w/o lower case
CUSTOMIZ.H  :  C library compile time parameters
STDIO.H     :  standard I/O functions header
STDIO.C     :  standard I/O functions
ALLOC.C     :  dynamic memory allocation functions
CRUNT2.C    :  C language parts of the run-time
FUNC.C      :  auxiliary functions
FORMATIO.C  :  printf, scanf, et al. functions
LONG.C      :  long integer functions
DOUBLE.C    :  double floating point functions
BCD80.C     :  assembly language support for DOUBLE.C
SAMP?.C     :  sample programs which test features of C
LIBC.REL    :  C library in relocatable format
LIBC.SUB    :  SUBMIT file to compile the library functions
*.SUB       :  various SUBMIT files
SH.COM      :  enhanced SUBMIT facility
```

CC.COM and C2.COM are the compiler executable files under CP/M-80.

...UNDER THE CP/M-86 OPERATING SYSTEM

If you have the CP/M-86 version of the compiler, you should find the following files on your working copy:

```
CC.CMD      :  first pass of the compiler: the parser
COD2COD.CMD:  second pass: Code Optimizer
C2I86.CMD   :  third pass: Code Generator
C2I86.RH    :  run-time library header file used for ASM86
C2I86.RT    :  run-time trailer file used for ASM86
C2PRE.ASM   :  run time header
C2POST.ASM  :  run time trailer
*.ASM       :  various other run-time sources
MDEP.C      :  #ASM/#ENDASM version of run-time
C.SUB       :  CP/M SUBMIT file for compilation
CBRACK.H    :  upper case defines for keyboards w/o lower case
CUSTOMIZ.H  :  C library compile time parameters
STDIO.H     :  standard I/O functions header
STDIO.C     :  standard I/O functions
ALLOC.C     :  dynamic memory allocation functions
CRUNT2.C    :  C language parts of the run-time
FUNC.C      :  auxiliary functions
FORMATIO.C  :  printf, scanf functions and so on
LONG.C      :  long integer functions
DOUBLE.C    :  double floating point functions
SAMP?.C     :  sample programs which test features of C
```

CC.CMD, COD2COD.CMD, and C2I86.CMD are the compiler executable files under CP/M-86. The second pass is split into COD2COD (the optimizer) and C2I86 (the code generator), which we call a third pass.

...UNDER THE MS-DOS AND PC-DOS OPERATING SYSTEMS

If you have the MS-DOS or PC-DOS versions of the compiler, you will find these files on your working copy:

```
CC.EXE      :  first pass of the compiler: the parser
COD2COD.EXE:  second pass: Code Optimizer
C2I86.EXE   :  third pass: Code Generator
C2PRE.ASM   :  run-time header for LINK
C2POST.ASM  :  run-time trailer for LINK
C2I86.RTB   :  run-time library header file for MASM
C2I86.RTM   :  run-time library trailer file for MASM
MDEP.C      :  #ASM/#ENDASM version of run-time
C.BAT       :  MS-DOS BATCH file for compilation
CBRACK.H    :  upper case defines for keyboards w/o lower case
CUSTOMIZ.H  :  C library compile time parameters
STDIO.H     :  standard I/O functions header
STDIO.C     :  standard I/O functions
ALLOC.C     :  dynamic memory allocation functions
CRUNT2.C    :  C language parts of the run-time
FUNC.C      :  auxiliary functions
FORMATIO.C  :  printf, scanf functions and so on
LONG.C      :  long integer functions
DOUBLE.C    :  double floating point functions
SAMP?.C     :  sample programs which test features of C
LIBC.BAT    :  BATCH file to compile the library functions
LIBC.LIB    :  C library in relocatable format
```

CC.EXE, COD2COD.EXE, and C2I86.EXE are the compiler executable files under MS-DOS and PC-DOS. The second pass is split into COD2COD (the optimizer) and C2I86 (the code generator), which we call a third pass.

For the CP/M, CP/M-86, and MS-DOS versions of the program, the following--C2PRE, C2POST, STDIO.H, STDIO.C, FORMATIO.C, ALLOC.C, FUNC.C, CRUNT2.C, LONG.C, and DOUBLE.C--are the built-in function files. (The other files on the disk with the extension ".C" are programs in SuperSoft C source code provided as examples.)

...UNDER THE ZMOS OPERATING SYSTEM

If you have the ZMOS version of the compiler, you should find the following files on your working copy:

```
CC.IMAG       :  first pass of the compiler, the parser
C2Z8001.IMAG  :  second pass, the Code Optimizer/Generator
WHEADER.REG   :  run-time library header file output by C2Z8001
WTRAILER.REG  :  run-time trailer file output by C2Z8001
ZSTDIO.REG    :  standard I/O functions
ALLOC.REG     :  dynamic memory allocation functions
CRUNT2.REG    :  C language parts of the run-time
FUNC.REG      :  auxiliary functions
FORMATIO.REG  :  printf, scanf et al.
LONG.REG      :  long integer functions
DOUBLE.REG    :  double floating point functions
SAMP?.REG     :  sample programs which test features of C
```

CC.IMAG and C2Z8001.IMAG are the compiler executable files under ZMOS. Files with the extension of ".REG" contain header files, library functions, or, in the case of SAMP?.REG, the sample programs.

FILES NEEDED FOR COMPILATION OF YOUR PROGRAMS

To compile a program you will need the executable files appropriate to your particular operating system. In addition, you will need the appropriate header and trailer files and the files containing any library functions desired. The code generation pass of the compiler automatically incorporates all the code segments and functions contained in its run-time library and the header and trailer files into your program during compilation. If using a relocating assembler, you will need the appropriate trailer file (C2.RTM for CP/M-80) and LIBC. Using a relocating assembler, you will have to link in the run-time library functions. A ".H" or ".C" file will be needed only if your program calls one or more of its functions. Since the functions contained in these files are such basic building blocks, most programs will call a substantial number of them. Therefore, if your program does require any of the functions defined in these ".C" or ".H" files, you must see that it incorporates all necessary function and data definitions from those files and, in turn, that it incorporates the same information from the functions that these functions call for. Five methods of accomplishing this are described in the section of Chapter 4 called "Incorporating Standard Library Functions". Also see Appendix D for a summary of which supplied C functions are in which files.

Since the final output of this compiler is in your machine's assembly source code, you will also need an assembler compatible with your operating system and hardware, plus the software required to load and run programs on your system.

COMMAND FORMATS AND SEQUENCES

To compile a SuperSoft C source program, each of the passes must be invoked with a sequence something like the following:

```
passname filename.ext ... options
```

Filename refers to the prefix portion of the file specification required by your operating system. Passname refers to the name of a pass of the C compiler.

Command line options for each pass should be separated by spaces. No options need be listed, since the default conditions specify the normally desired mode of operation for the compiler. However, if you are running the compiler on a CPU other than an 8080, 8085, 8086, or Z80 (the 8080 series), or are targetting for a different machine than the host machine, you may have to set the machine-dependent options, +J, -I, and -P, described on page 31, to their proper values for your processor.

When invoking CC, you may list as many filenames in the command line as you desire, separated by spaces. The files specified need not contain complete functions since they will all be parsed, in the order listed, as if they were one file. The U-code output file will be assigned the first filename given. You may specify only one filename when executing the other passes. Each pass emits a file with the same name you specified but with a different extension.

By convention, C source code files have the extension or suffix ".C", although any extension is legal. CC automatically provides the extension ".COD" for its U-code output files. The extension ".U" is used for optimized U-code. The extension ".ASM" is used for assembly language output. Other passes have similar specifications. These are:

OPERATING SYSTEM	PASS NAME	INPUT FILE		OUTPUT FILE
CP/M-80, CP/M-86, and MS-DOS	CC	f.C	-->	f.COD
CP/M-80	C2	f.COD	-->	f.ASM
CP/M-86, MS-DOS	COD2COD	f.COD	-->	f.U
ZMOS	C2Z8001	f.COD	-->	CASM.REG
UNIX	C2Z8002	f.COD	-->	f.ASM
UNIX	C2Z8002	f.U	-->	f.ASM
CP/M-86	C2I86	f.COD	-->	f.A86
CP/M-86	C2I86	f.U	-->	f.A86
MS-DOS	C2I86	f.COD	-->	f.ASM
MS-DOS	C2I86	f.U	-->	f.ASM

After the SuperSoft C Compiler has done its work, it leaves a file that should be run through an assembler. The choice of assemblers is wide, depending on the operating system and your preference.

OPERATING SYSTEM	PASS NAME	INPUT FILE		OUTPUT FILE
CP/M-80	ASM	f.ASM	-->	f.HEX
CP/M-80	MAC	f.ASM	-->	f.HEX
CP/M-80	M80	f.ASM	-->	f.REL
CP/M-80	RMAC	f.ASM	-->	f.REL
CP/M-86	ASM86	f.A86	-->	f.H86
MS-DOS	MASM	f.ASM	-->	f.OBJ
UNIX	AS	f.ASM	-->	f.O
ZMOS	RASM	f.REG	-->	f.IMAG

After the compilation pass, the output file must be transformed into an executable file. This completes the compilation process. Here are some examples of this completion, organized by operating system and file transformation.

OPERATING SYSTEM	PASS NAME	INPUT FILE		OUTPUT FILE
CP/M-80	LOAD	f.HEX	-->	f.COM
CP/M-80	L80	f.REL	-->	f.COM
CP/M-80	ELINK	f.REL	-->	f.COM
CP/M-80	LINK80	f.REL	-->	f.COM
CP/M-86	GENCMD	f.H86	-->	f.CMD
MS-DOS	LINK	f.OBJ	-->	f.EXE
UNIX	LD	f.O	-->	f

As an example of the kind of procedure that must be followed to compile and run a SuperSoft C program in a specific machine and operating system environment, we will describe the procedures required under the various operating systems. We will assume that you feel ready to compile and run your first C program, SAMPl.C. After first checking that all the necessary files are available on your disk and that your program includes all data and function definitions required, you are ready to begin.

COMPILATION USING ASM, THE STANDARD CP/M-80 ASSEMBLER

To transform SAMP1.C into an executable command file so that it can be run under CP/M-80, you should type each of the commands listed below in sequence. This procedure is similar to the procedure for MAC described later. Make sure that the files C2.RH and C2.RT are on the default disk (usually the same disk as C2.COM). Execution of each of these CP/M commands begins as soon as you hit RETURN at the end of the line. Type:

```
CC SAMP1.C
C2 SAMP1.COD +ASM
ASM SAMP1
LOAD SAMP1
SAMP1
```

Each of the above commands causes a new file to be created on the disk which contains SAMP1.C. The names of these files are, respectively:

```
SAMP1.COD     ; output from CC
SAMP1.ASM     ; output from C2
SAMP1.HEX     ; output from ASM
SAMP1.COM     ; output from LOAD
```

The last file created is the executable command file for your program. This is the only form in which your program can be run under CP/M. (The others may be deleted.) Once this file is created, you need only type:

```
SAMP1
```

Then hit RETURN, and execution of your program will begin.

CC, C2, and ASM may also generate error messages indicating defects in your program. Although these messages are largely self-explanatory, a complete explanation of them can be found in Appendix F of this manual. The CP/M documentation provides information on the error messages generated by ASM. Two types of messages from ASM can have roots in poorly formed C programs: (1) Phase errors and multiply defined labels are usually caused by the redefinition of a C external (including C functions). Note that externals have only so many significant leading characters. This number depends on the assembler (and

loader) used. (2) Undefined labels usually indicate undefined C externals (including C functions).

In the course of developing programs, you are likely to repeat the procedure just described--with no change except in the filename specified--an enormous number of times. It would save considerable time if you could cause this entire procedure to be carried out by means of a single command. Under CP/M, the SUBMIT command and SUBMIT files provide the means of accomplishing this.

An appropriate SUBMIT file, with filename C.SUB, would contain the following command lines

(ASM.SUB)

```
CC $1.C
C2 $1.COD +ASM
ASM $1
LOAD $1
$1
```

where "$1" is a symbolic parameter to be later replaced by the first actual parameter of a SUBMIT command.

After you have created such a file on your disk, type:

```
SUBMIT C SAMP1
```

Then press RETURN. If CP/M finds C.SUB on the current disk, the following five commands will be executed (SAMP1 has been substituted for each occurrence of "$1"):

```
CC SAMP1.C
C2 SAMP1.COD +ASM
ASM SAMP1
LOAD SAMP1
SAMP1
```

For further information about SUBMIT files, consult the CP/M documentation or one of the several CP/M manuals available.

COMPILATION USING MAC, THE DIGITAL RESEARCH MACRO ASSEMBLER

To transform SAMP1.C into an executable command file using the Digital Research Assembler MAC and actually execute that file under CP/M-80, you should type each of the commands listed below in sequence. This procedure is similar to the procedure for ASM. Make sure that the files C2.RH and C2.RT are on the default disk (usually the same disk as C2.COM). Execution of each of these CP/M commands begins as soon as you hit RETURN at the end of the line. Type:

```
CC SAMP1.C
C2 SAMP1.COD +ASM
MAC SAMP1
LOAD SAMP1
SAMP1
```

Each of the above commands causes a new file to be created on the disk which contains SAMP1.C. The names of these files are, respectively:

```
SAMP1.COD     ; output from CC
SAMP1.ASM     ; output from C2
SAMP1.HEX     ; output from MAC
SAMP1.COM     ; output from LOAD
```

The last file created is the executable command file for your program. This is the only form in which your program can be run under CP/M. (The others may be deleted.) Once this file is created, you need only type:

```
SAMP1
```

Then hit RETURN, and execution of your program will begin.

CC, C2, and MAC may also generate error messages indicating defects in your program. Although these messages are largely self-explanatory, a complete explanation of them can be found in Appendix F of this manual. The CP/M documentation provides information on the error messages generated by MAC. Two types of messages from MAC can have roots in poorly formed C programs: (1) Phase errors and multiply defined labels are usually caused by the redefinition of a C external (including C functions). Note that externals have only so many significant leading characters. This number depends on the assembler (and

loader) used. (2) Undefined labels usually indicate undefined C externals (including C functions).

In the course of developing programs, you are likely to repeat the procedure just described--with no change except in the filename specified--an enormous number of times. It would save considerable time if you could cause this entire procedure to be carried out by means of a single command. Under CP/M, the SUBMIT command and SUBMIT files provide the means of accomplishing this.

An appropriate SUBMIT file, with filename C.SUB, would contain the following command lines

```
CC $1.C
C2 $1.COD +ASM
MAC $1
LOAD $1
$1
```

where "$1" is a symbolic parameter to be later replaced by the first actual parameter of a SUBMIT command.

After you have created such a file on your disk, type:

```
SUBMIT C SAMP1
```

Then press RETURN. If CP/M finds C.SUB on the current disk, the following five commands will be executed (SAMP1 has been substituted for each occurrence of "$1"):

```
CC SAMP1.C
C2 SAMP1.COD +ASM
MAC SAMP1
LOAD SAMP1
SAMP1
```

For further information about SUBMIT files, consult the CP/M documentation or one of the several CP/M manuals available.

COMPILATION USING ASM86, THE STANDARD CP/M-86 ASSEMBLER

To transform SAMP1.C into an executable command file so that it can be run under CP/M-86, you should type each of the commands listed below in sequence. Type:

```
CC SAMP1.C
COD2COD SAMP1.COD
C2I86 SAMP1.U
ASM86 SAMP1
GENCMD SAMP1 DATA[X1000]
```

Each of the above commands causes a new file to be created on the disk which contains SAMP1.C. The names of these files are, respectively:

```
SAMP1.COD     ; output from CC
SAMP1.U       ; output from COD2COD
SAMP1.A86     ; output from C2I86
SAMP1.H86     ; output from ASM86
SAMP1.CMD     ; output from GENCMD
```

The last file created is the executable command file for your program. This is the only form in which your program can be run under CP/M-86. (The others may be deleted.) Once this file is created, you need only type:

```
SAMP1
```

Then hit RETURN and execution of your program will begin.

CC, COD2COD, C2I86, and ASM86 may also generate error messages indicating defects in your program. Although these messages are largely self-explanatory, a complete explanation of them can be found in Appendix F of this manual. In addition, the CP/M-86 documentation provides information on the error messages generated by ASM86. Two types of messages from ASM86 can have roots in poorly formed C programs: (1) Phase errors and multiply defined labels are usually caused by the redefinition of a C external (including C functions). Note that externals have only so many significant leading characters. This number depends on the assembler (and loader) used. (2) Undefined labels usually indicate undefined C externals (including C functions). In the course of developing programs, you are likely to repeat the procedure just described--with no change except in the filename specified--an enormous number

of times. It would save considerable time if you could cause this entire procedure to be carried out by means of a single command. Under CP/M-86, the SUBMIT command and SUBMIT files provide the means of accomplishing this.

An appropriate SUBMIT file, with filename C.SUB, would contain the following command lines

```
CC $1.C
COD2COD $1.COD
C2I86 $1.U
ASM86 $1
GENCMD $1  data[x1000]
$1
```

where "$1" is a symbolic parameter to be later replaced by the first actual parameter of a SUBMIT command.

After you have created such a file on your disk, type:

```
SUBMIT C SAMP1
```

Then press RETURN. If CP/M finds C.SUB on the current disk, the following five commands will be executed (SAMP1 has been substituted for each occurrence of "$1"):

```
CC SAMP1.C
COD2COD SAMP1.COD
C2I86 SAMP1.U
ASM86 SAMP1
GENCMD SAMP1  DATA[x1000]
SAMP1
```

For further information about SUBMIT files, consult the CP/M-86 documentation or one of the several CP/M-86 manuals available.

COMPILATION USING MASM, AN MS-DOS/PC-DOS ASSEMBLER

To transform SAMP1.C into an executable command file so that it can be run under MS-DOS, you should type each of the commands listed below in sequence. Type:

```
CC SAMP1.C
COD2COD SAMP1.COD
C2I86 SAMP1.U -ASM +MSDOS
MASM SAMP1.ASM;
LINK C2PRE+SAMP1+C2POST,SAMP1,NUL,LIBC
SAMP1
```

Each of the above commands causes a new file to be created on the disk which contains SAMP1.C. The names of these files are, respectively:

```
SAMP1.COD     ; output from CC
SAMP1.U       ; output from COD2COD
SAMP1.ASM     ; output from C2I86
SAMP1.OBJ     ; output from MASM
SAMP1.EXE     ; output from LINK
```

The last file created is the executable command file for your program. This is the only form in which your program can be run under MS-DOS. (The others may be deleted.) Once this file is created, you need only type:

```
SAMP1
```

Then hit RETURN, and execution of your program will begin.

CC, COD2COD, C2I86, and MASM may also generate error messages indicating defects in your program. Although these messages are largely self-explanatory, a complete explanation of them can be found in Appendix F of this manual. In addition, the MS-DOS documentation provides information on the error messages generated by MASM. Two types of messages from MASM can have roots in poorly formed C programs: (1) Phase errors and multiply defined labels are usually caused by the redefinition of a C external (including C functions). Note that externals have only so many significant leading characters. This number depends on the assembler (and loader) used. (2) Undefined labels usually indicate undefined C externals (including C functions).

In the course of developing programs, you are likely to repeat the procedure just described--with no change except in the filename specified--an enormous number of times. It would save considerable time if you could cause this entire procedure to be carried out by means of a single command. Under MS-DOS, the BATCH command and BATCH files provide the means of accomplishing this.

An appropriate BATCH file, with filename C.BAT, would contain the following command lines

```
CC %1.C
COD2COD %1.COD
C2I86 %1.U -ASM +MSDOS
MASM %1.ASM;
LINK C2PRE+%1+C2POST,%1,NUL,LIBC
%1
```

where '%1' is a symbolic parameter to be replaced later by the first actual parameter of a BATCH command.

After you have created such a file on your disk, type:

```
C SAMP1
```

Then hit RETURN.

For further information about BATCH files consult the MS-DOS documentation or one of the several MS-DOS manuals available.

COMPILATION USING M80, A RELOCATING ASSEMBLER

To transform SAMP1.C into an executable command file so that it can be used under CP/M-80, you should type each of the commands listed below in sequence. This procedure is the same as those described above with the exception that relocatable modules are produced and modules are linked together. Type:

```
CC SAMP1.C
C2 SAMP1.COD
M80 =SAMP1.ASM
L80 C2PRE,SAMP1,LIBC/S,C2POST,SAMP1/N/E/Y
SAMP1
```

C2POST must be specified as shown (last module).

Alternatively, the L80 line can be replaced by using SuperSoft's ELINK:

```
ELINK BC(SAMP1);C2PRE,SAMP1;SR(LIBC);IN(C2POST);EN;
```

Each of the above commands causes new files to be created on the current disk. These are:

```
SAMP1.COD   ; output from CC
SAMP1.ASM   ; output from C2
SAMP1.REL   ; output from M80
SAMP1.COM   ; output from L80 or ELINK
```

Using the above procedure will require the availability of the following files in relocatable format:

```
C2PRE.REL    ; run-time header from C2PRE.ASM
C2POST.REL   ; run-time trailer from C2POST.ASM
LIBC.REL     ; the C library
```

LIBC.REL is a library created from the recompilation of the following sources: ALLOC.C, STDIO.C, CRUNT2.C, FUNC.C, and FORMATIO.C. You will also need the file C2.RTM present on disk.

An appropriate SUBMIT file for relocating assemblers using L80 is:

```
CC $1.C
C2 $1.COD
M80 =$1.ASM
L80 C2PRE,$1,LIBC/S,C2POST,$1/N/E/Y
$1
```

Or if ELINK is used:

```
CC $1.C
C2 $1.COD
M80 =$1.ASM
ELINK BC($1);C2PRE,$1;SR(LIBC);IN(C2POST);EN;
$1
```

COMPILATION USING RMAC, A RELOCATING ASSEMBLER

To transform SAMP1.C into an executable command file so that it can be used under CP/M-80, you should type each of the commands listed below in sequence. This procedure also produces REL modules and links them together as in the M80 and L80 example above. Below is the procedure:

```
CC SAMP1.C
C2 SAMP1.COD -OFILESAMP1.MAC
RMAC SAMP1
LINK SAMP1,C2PRE,LIBC[S],C2POST
SAMP1
```

C2POST must be specified as shown (last module).

Alternatively, the LINK80 line can be replaced by using SuperSoft's ELINK:

```
ELINK BC(SAMP1);C2PRE,SAMP1;SR(LIBC);IN(C2POST);EN;
```

C2POST must be specified last. Each of the above commands causes a new file to be created on the currently logged in disk. These are:

```
SAMP1.COD   ; output from CC
SAMP1.MAC   ; output from C2
SAMP1.REL   ; output from RMAC
SAMP1.COM   ; output from LINK80 or ELINK
```

Using the above procedure will require the availability of the following files in RMAC REL format:

```
C2PRE.REL    ; from run-time library C2PRE.ASM
C2POST.REL   ; run-time trailer from  C2POST.ASM
LIBC.REL     ; the C library
```

LIBC.REL is a library created from the recompilation of the following sources: ALLOC.C, STDIO.C, CRUNT2.C, FUNC.C, and FORMATIO.C.

COMPILATION USING AS, A RELOCATING ASSEMBLER

To transform SAMP1.C into an executable command file so that it can be used under UNIX, you should type each of the commands listed below in sequence. This procedure also produces relocatable modules and links them together as in the M80 and L80 example above. Below is the procedure:

```
CC SAMP1.C
COD2COD SAMP1.COD
C2Z8002 SAMP1.U
AS SAMP1.ASM -O SAMP1.O
LD SAMP1.O C2PRE.O LIBC.A C2POST.O
SAMP1
```

C2POST.O must be specified last. Each of the above commands causes a new file to be created on the currently logged in disk. These are:

```
SAMP1.COD    ; output from CC
SAMP1.U      ; output from COD2COD
SAMP1.ASM    ; output from C2Z8002
SAMP1.O      ; output from AS
SAMP1        ; output from LD
```

Using the above procedure will require the availability of the following files in UNIX.O format:

```
C2PRE.O      ; from run-time library C2PRE.ASM
C2POST.O     ; run-time trailer from C2POST.ASM
LIBC.A       ; the C library
```

LIBC.A is a library created from the recompilation of the following sources: ALLOC.C, STDIO.C, CRUNT2.C, FUNC.C, and FORMATIO.C.

COMPILATION USING RASM

To transform SAMP1.C into an executable command file so that it can be used under ZMOS, you should type each of the commands listed below in sequence. This procedure also produces IMAG modules and links them together as in the ASM example. Below is the procedure:

```
CC SAMP1.C
C2Z8001 SAMP1.COD
RASM SAMP1
SAMP1
```

Each of the above commands causes a new file to be created on the currently logged in disk. These are:

```
SAMP1.COD   ; output from CC
SAMP1.REG   ; output from C2Z8001
SAMP1.IMAG  ; output from RASM
```

You will also need the files WHEADER.REG and WTRAILER.REG present on disk.

COMMAND LINE OPTIONS

CC MACHINE-INDEPENDENT COMMAND LINE OPTIONS

+A places the address of the last memory location used by CC in front of each C source code line inserted as a comment into the output file. This option has no effect unless the option +L has also been selected. This memory allocation information is helpful in determining if and when the compiler is out of memory. Default is -A.

-BUFSIZ is followed immediately by a decimal integer. This integer becomes the buffer size for I/O operations. The larger this value, the faster the compilation. Smaller sizes use less memory. A number of 128 is about the minimum for most systems. This size is in effect for both input and output buffers (see -WBUFSIZ below.) If this option is used, the selected value will be printed on the console. Default is -BUFSIZ1024.

+CO forces output to console. Default is -CO (output to filename.COD).

+CR appends a Carriage Return to each U-code output line generated. Certain text editors (e.g., CP/M's ED) expect this format. Default is -CR.

-D followed immediately (no white space) by an identifier, defines that identifier just as if a #DEFINE was encountered for that identifier. Useful for #IFDEF and #ENDIF preprocessor directives. Default is to predefine no identifier.

+DDT incorporates additional debugging information into the output file. Default is -DDT.

+F flushes (writes to file) all U-code lines generated immediately after each line of C source code is processed. If options +L and +F are both selected, then each source code comment line placed in the output file is followed immediately by all the U-code lines generated from it. Selecting this option slows execution of both CC and C2 (or C2I86) and inhibits certain optimizations normally performed by C2 (or COD2COD). Default is -F.

-G turns off U-code generation. This option is useful for checking the syntax of a source code file. Selecting

this option and option +L below will also produce a numbered listing of the C source code with macro defines expanded. Default is to generate U-code. Default is +G.

+L lists each line of C source code as a comment in the U-code output file. The source lines are grouped together by function and are thus only loosely associated with their corresponding U-code lines. Macros are expanded and line numbers appear before each source code line. Default is -L.

-LISTLNO suppresses listing of line numbers in the output file. Active only if +L is also specified. Default is +LISTLNO.

+LNO places source code line number in front of first corresponding line of U-code in the output file. Output also indicates line number at which each file incorporated by use of a #INCLUDE directive begins. Nested #INCLUDEs are clearly indicated as well. Default is -LNO.

-N is followed immediately by an integer specifying the initial value for the consecutively numbered labels generated by CC. Default is an initial value of two.

-OFILE redirects the output file to any specified drive under any specified filename. It is followed immediately by a filename which will be used for the output file. There must be no whitespace between -OFILE and the filename.

+PRE is equivalent to specifying +L -LISTNO -G. Creates a file that has been preprocessed only. The output can be rerun through CC. Default is -PRE.

+S prints symbol table information as each symbol is entered into the symbol table. Mostly of use to the compiler writer. Default is -S.

+SILENT suppresses some output header information. Helps lower the clutter on the console. Default is -SILENT.

+SUBM conditions the output to allow separate assembly of each C function. If this switch is specified as a CC parameter then it also should be specified as a C2 (or C2I86) parameter. Default is -SUBM.

+CASE accepts keywords without regard to (upper/lower) case. This applies to preprocessor keywords as well as C keywords. Useful with terminals that accept upper case only. Note the include file CBRACK.H has upper case keywords for common C tokens such as { and } . This is useful for keyboards without lower case. Default is -CASE.

-WBUFSIZ is followed immediately by a decimal integer. This integer becomes the buffer size for output operations. The larger this value, the faster the compilation. Smaller sizes use less memory. A number of 128 is about the minimum for most systems. If this option is used, the selected value will be printed on the console. Default is -WBUFSIZ1024.

CC MACHINE-DEPENDENT COMMAND LINE OPTIONS

The three options below may be used to tailor CC for a particular target processor (CPU). They are the only machine-dependent options. If the value set for any of these options is not appropriate for your processor, the compiler will not function properly. (In what follows, jflag, int_size, and pad_size all refer to the names of variables within the compiler.)

+J sets jflag equal to one (1). Whenever a single byte representing a char is passed as a local parameter to a function, it is widened to an int. A setting of one (1) for jflag causes such a byte to be stored in the rightmost byte of its assigned storage location. This is the proper setting for the Z8000. The default condition is to set jflag equal to zero (0) which causes the same byte representing a char to be stored in the leftmost byte of its assigned storage location. This is the proper setting for the 8080 series CPUs.

-I is followed immediately by an integer specifying the size of an int (int_size) in bytes. Default is two bytes.

-P is followed immediately by an integer specifying the appropriate modulus (pad_size) for all integer addresses. Since each integer address will be a multiple of the value of pad_size, each integer address modulo the value of pad_size will be equal to zero (0). Default is a modulus of one (1).

The values for the three above options required to tailor this compiler for a particular target processor may be set in one operation through use of one of the composite flags defined below.

+i8080 sets: jflag = 0, int_size = 2, pad_size = 1. Target processors are the Intel 8080, 8085, 8086, 8088, 186, 188, and 286.

+Z80 sets the same values as above. Target processor is the Zilog Z80. Note: This is a CC option and not a C2 option, so 8080 mnemonics are still output by C2.

+Z8000 sets: jflag = 1, int_size = 4, pad_size = 2. Target processor is the Zilog Z8001.

+Z8001 sets: jflag = 1, int_size = 4, pad_size = 2. Target processor is the Zilog Z8001.

+Z8002 sets: jflag = 1, int_size = 2, pad_size = 2. Target processor is the Zilog Z8002.

Default condition is identical to the result of +i8080 or +Z80.

C2, COD2COD, and C2I86 COMMAND LINE OPTIONS

+ASM is used to indicate that an absolute assembler is being used. It includes the header file (default: C2.RH or C2I86.RH) before any generated code and the trailer file (default: C2.RT or C2I86.RT) after any generated code. These files should be on the default disk. +ASM is not used under MS-DOS or PC-DOS.

-ASM is the default and it indicates that a relocating assembler is being used. Under CP/M, C2.RTM, the default trailer file for relocating assemblers which is appended to the end of the file, is used instead of C2.RT. (There is no header file.) Under MS-DOS or PC-DOS, -ASM uses C2I86.RTB and C2I86.RTM as the header and trailer files, respectively, to output code definitions.

-BUFSIZ is followed immediately by a decimal integer. This integer becomes the buffer size for I/O operations. The larger this value, the faster the compilation. Smaller sizes use less memory. A number of 128 is about the minimum for most systems. This size is in effect for both input and output buffers (see -WBUFSIZ below.) If this option is used, the selected value will be printed on the console. Default is -BUFSIZ1024.

+CO forces output to console where output may be viewed. Default is to output to filename.ASM.

-ENT<keyword> defines the keyword used to declare the entry point of a label for relocating assemblers. (Note: no space between -ENT and the keyword.) Default is -ENTENTRY.

-EXT<keyword> defines the keyword used to declare an external label for relocating assemblers. (Note: no space between -EXT and the keyword.) Default is -EXTPUBLIC.

-G turns off assembly code generation. Default is to generate assembly output code.

+L places each line of U-code in the assembly code output file as a comment. If optimization is performed, U-code lines are grouped together by function; otherwise, they occur directly before the code that is generated for the line. Default is -L.

+MSDOS instructs C2I86 to output code compatible with MS-DOS/PC-DOS assembler.

-O turns off optimization. U-code to assembly code translation only is done. Default is to perform optimization.

+OC traces optimization process by writing information to the output file indicating each code change as it is made. This option will slow execution of C2 or COD2COD and vastly increase the amount of output. Of particular use only to the optimizer author. Default is -OC.

-OFILE redirects the output file to any specified drive under any specified filename. It is followed immediately by a filename which will be used for the output file. There must be no whitespace between -OFILE and the filename.

-ORG followed immediately by a decimal integer forces the output code to be ORGed at the value specified. Default is 256 if +ASM is specified; no ORG at all otherwise.

+PRGL causes the name of the function currently being optimized to be displayed on the console as an indication of where C2 or COD2COD is in its execution. Default is -PRGL.

-Q is followed immediately by an integer specifying the optimization level. Default is full optimization. (Not currently implemented.)

-RH<filename> causes the optimizer to use the specified file as the run-time header file. (Note: there is no space between -RH and the filename.) Default is C2.RH (C2I86.RH under CP/M-86) if +ASM is specified. For -ASM, no file is prepended under CP/M; C2I86.RTS is used under MS-DOS.

-RT<filename> causes the optimizer to use the specified file as the run-time trailer file. (Note: there is no space between -RT and the filename.) Default is C2.RT (C2I86.RT under CP/M-86) if +ASM is specified; for -ASM, C2.RTM (C2I86.RTM under MS-DOS).

+T generates and inserts code in output file that, at run-time, causes the name of each function to be displayed each time it is entered. This option, useful in debugging, allows the programmer to trace the control flow within a program. (Not currently implemented.) Default is -T.

-WBUFSIZ is followed immediately by a decimal integer. This integer becomes the buffer size for output operations. The larger this value, the faster the compilation. Smaller sizes use less memory. A number of 128 is about the minimum for most systems. If this option is

used, the selected value will be printed on the console. Default is -WBUFSIZ1024.

-X causes generation of slightly smaller code at the expense of its speed of execution. Optimization performed trades speed efficiency for space efficiency. Default is to trade space efficiency for speed efficiency. The difference is relatively slight because most optimizations have a positive effect on both speed and space efficiency. (Not currently implemented.)

+Z is followed immediately by a character string which C2 (C2I86 under CP/M-86 and MS-DOS) uses as a prefix for all the labels it generates. A prefix 'C' is the default (+ZC).

THE COMPILER PREPROCESSOR DIRECTIVES

All files input to the SuperSoft C Compiler pass through a preprocessor which scans them for lines beginning with '#'. Such lines, which are not themselves compiled but instead direct the compilation process, are known as compiler preprocessor directives. These lines are terminated by newline characters rather than semicolons. The preprocessor directives supported in SuperSoft C are:

```
#define
#endif
#else
#include
#if
#ifdef
#ifndef
#line
#undef
#asm
#endasm
```

Each of these directives is syntactically independent of the rest of the C syntax and may appear anywhere in a program. The effects of all the directives extend to the end of a compilation. Their effects and uses are described in the paragraphs below.

THE #DEFINE DIRECTIVE

If a line of the form

```
#define IDENTIFIER TOKENSTRING
```

is found in source input, it causes the preprocessor to replace all subsequent occurrences of the identifier specified with the TOKENSTRING specified. (A token is a lexical unit in a programming language, such as an identifier, an operator, a keyword, etc.) This is the simplest form of macro substitution. If a #DEFINE'd identifier occurs within double or single quotes, it will not be replaced. Each replacement string will be rescanned for other #DEFINE'd identifiers, thus allowing nesting of #DEFINE directives. Redefining an identifier results in a compilation error. A parameterized #DEFINE is not supported.

The most common use of this directive is in defining a set of symbolic constants in a program's header for use in the functions that follow (a far better practice than inserting literal constants in program statements). An abbreviated example of this kind of use is:

```
#define BSIZ 0x100
char Buf1[BSIZ], Buf2[BSIZ];

main()
{
     register int i;

     for(i=0;i++<BSIZ;){
               Buf1[i] = Buf2[i] = 0;
}
```

Macro substitution through the #define directive has many other uses. For further examples and information refer to pages 12; 86, and 207 in Kernighan and Ritchie.

THE #LINE DIRECTIVE

The line directive sets the source line number as used by the compiler. This will affect the line number as printed by the compiler when errors occur. The form is:

```
#line   LINENUMBER
```

where LINENUMBER is a constant.

E #INCLUDE DIRECTIVE

If a line of the form

```
#include "drivename:filename"
```

or a line of the form:

```
#include <drivename:filename>
```

is input to the compiler, it causes the preprocessor to replace that line with the entire contents of the file so named on the drive specified. If the preprocessor cannot find a file, it will look for it on the current drive. In the future (that is, not in the current version) the preprocessor will look for the named file on all drives that it knows are available (all drives mentioned up to this point) if the #INCLUDE <filename> directive is used. Any #INCLUDE directives found in a #INCLUDE'd file will be processed in the same way. Hence, nesting of #INCLUDE directives is possible. For examples and information refer to pages 86, 143, and 207 in Kernighan and Ritchie.

THE #IF, #IFDEF, #IFNDEF, #ELSE, AND #ENDIF DIRECTIVES

These directives perform conditional compilation. The general form is:

```
#CONDITIONAL
        .
        .
   TRUECODE
        .
        .
#else
        .
        .
   FALSECODE
        .
        .
#endif
```

where #CONDITIONAL is one of #IF, #IFDEF, or #IFNDEF. The #ELSE is optional. That is, the form can be:

```
#CONDITIONAL
        .
        .
   TRUECODE
        .
        .
#endif
```

If the #CONDITIONAL is considered to be true, then lines shown as TRUECODE are compiled and the lines labeled FALSECODE are ignored. If the #CONDITIONAL is considered to be false, then the lines labeled FALSECODE are compiled and the lines shown as TRUECODE are ignored.

The #IF directive is of the form:

```
#if     CONSTANT
```

The CONSTANT must be a constant. If it is, then the #IF is considered to be true.

The #IFDEF directive is of the form:

#ifdef IDENTIFIER

The IDENTIFIER must be a #DEFINE-style identifier. If IDENTIFIER has been previously defined via a #DEFINE, then the #IFDEF is considered to be true.

#ifndef IDENTIFIER

The IDENTIFIER must be a #DEFINE-style identifier. If IDENTIFIER has been previously defined by means of a #DEFINE, then the #IFNDEF is considered to be false.

HE #ASM AND #ENDASM DIRECTIVES

This special feature of SuperSoft C allows you to insert lines of assembly code directly into your C source code file. This would appear as follows:

```
.
.
.
putchar('y');
#asm

     mvi a,88
     call output

#endasm
crlf();
.
.
.
```

All assembly language lines between #ASM and #ENDASM go through the passes of the compiler unchanged and are incorporated at the corresponding locations in the final output file. Inside of a C function, the lines will be executed as encountered; outside of a function the lines must contain a label and must be explicitly called in order to be executed. The type of assembler language lines that are put outside of a C function are directives and equates. No included assembly language lines are optimized or checked for errors. Labels beginning with 'C' in the inserted code

should be avoided, since labels with this prefix are generated by the compiler and may result in a duplicate label when the program is assembled.

Included assembly language lines may be used anywhere in a C source file. If they occur outside of a C function or if the -O option (don't optimize) option is used on the C2 or COD2COD pass, then the amount of included assembly language is unlimited. Otherwise, the amount of assembly language will be limited by the per-procedure optimizations of C2 or COD2COD.

The following options are assumed by C2 under CP/M-80 and will affect how the included assembly language lines are used:

.RADIX 10 (decimal radix),
.I8080 (8080 mnemonics),
CSEG (code segment).

C globals may be referenced in included assembly language (and also in linked assembly language) and assembly symbols may be referenced by C. Currently, C globals will match assembler symbols by the same name. This will not always be the case. In a subsequent version of the compiler, an extra postfix character will be added to a C symbol for it to match an assembler global. Also STATICs will not have a readily available assembler symbol.

C makes use of these registers:

primary register (the current sub-expression),
secondary register (the previous sub-expression),
stack pointer,
register variable,
base pointer (only in some implementations).

C expects the register variable to be preserved at all times; the primary register to be set to the return values on function returns; and the primary and secondary registers to be preserved within the computation of any sub-expression. The stack pointer should also be preserved across function calls and included assembly language. The secondary register is more transitory: it is not expected to be preserved across function calls. However, it is expected to be preserved within most sub-expressions and during the vectoring into a switch case. It is not normally desirable or useful to impose included assembly language inside of a C sub-expression. An example of including assembly language in a C sub-expression which would cause problems:

```
        a =
#asm
        MVI     H,9
#endasm
                b;
```

When calling C code from assembly language or from any other language, it is best not to assume that any registers are preserved by the C code, although some registers are preserved by some SuperSoft C code generators. When calling assembly language from C, or when "falling through" including assembly language lines, be sure to preserve the register variable and the stack pointer. Other (primarily state) registers may have to be preserved on various machines. For instance the DS, CS, and SS registers must be preserved on the 8086. The following table summarizes the assignment of C registers.

Register	8080/Z80	8086/286	Z8002	Z8001
primary	HL	BX	R4	RR4
secondary	DE	SI	R5	RR6
stack pointer	SP	SP	R15	RR14
reg. variable	BC	DI	R8	RR8

4E #UNDEF DIRECTIVE

This directive is of the form:-

```
#undef IDENTIFIER
```

It forces the preprocessor to forget the named IDENTIFIER.

USING LIST FUNCTIONS

SuperSoft C is able to handle functions without a predetermined number of arguments. This is done with an additional function attribute called list. The SuperSoft C standard library functions include the following predefined list functions: printf, scanf, fprintf, fscanf, sprintf, and sscanf. The operation of these list functions is transparent to the user: you need not do anything special to take advantage of these built-in functions.

However, if you wish to use your own list functions, you will have to know how the list attribute is declared and how parameters are passed. For the list functions of your own creation to execute properly, they must be declared before they are first used by your program--as follows:

```
TYPE listf(.);
```

where listf is an arbitrary list function, and TYPE is the return value declaration for listf, which should, of course, match all other declarations for listf. This declaration causes, for each invocation of that function, the last argument on the parameter stack to be the argument count. If this declaration appears at the top of your program as a global external, it need appear only once in your program.

The body of a list function must collect its arguments in a special manner in order to take into account the altered parameter stack. You may want to look at the body of one of the above functions for the special coding necessary to utilize the additional information available via the parameter stack. The list attribute allows C functions which expect a certain number of arguments to protect themselves from being passed an incorrect number of arguments, permitting the coding of very reliable programs. For instance, the following poorly formed function call can be detected:

```
fprintf("%s");
```

This detection ability is a feature that few C compilers have. With most C compilers the above code would cause undefined (and potentially disastrous) I/O operations to occur.

The body of a list function should look something like this:

```
pf(nargs)
 int nargs[];
{
        unsigned cnt;
        int * p;
        cnt = nargs;
        if(cnt<MINIMUM || cnt>MAXIMUM)
                return;

        p = Xrev(&nargs); /* lib arg reversal fn */

/*
 * at this point:
 *
 *      cnt is the count of passed parameters
 *      p[0] is the leftmost parameter
 *      p[1] is the 2nd leftmost parameter
 *      p[2] is the 3rd leftmost parameter
 *      ... and so on up to p[cnt-1]
 */

}
```

CHAPTER 3

The SuperSoft C Standard Library Functions

The standard library functions supplied with the SuperSoft C Compiler are defined in the following files:

...UNDER CP/M-80

```
C2.RH      :  run-time library header
C2.RT      :  run-time library trailer
C2.RTM     :  run-time trailer for RMAC and M80
C2PRE.ASM  :  run-time header for L80
C2POST.ASM:   run-time trailer for L80
MDEP.C     :  #ASM/#ENDASM version of run-time
CUSTOMIZ.H:   header for the C library
CBRACK.H   :  header for keyboards w/o lowercase
STDIO.H    :  header for STDIO.C
STDIO.C    :  standard UNIX-style I/O functions
ALLOC.C    :  dynamic memory allocation functions
CRUNT2.C   :  auxiliary, common run-time functions
LONG.C     :  long integer functions
DOUBLE.C   :  double floating point functions
BCD80.C    :  assembly language support for DOUBLE.C
FUNC.C     :  auxiliary functions
FORMATIO.C:   printf, scanf, et al.
```

...UNDER CP/M-86

```
C2I86.RH   :  run-time library header
C2I86.RT   :  run-time library trailer
MDEP.C     :  #ASM/#ENDASM version of run-time
CUSTOMIZ.H:   header for the C library
CBRACK.H   :  header for keyboards w/o lowercase
STDIO.H    :  header for STDIO.C
STDIO.C    :  standard UNIX-style I/O functions
ALLOC.C    :  dynamic memory allocation functions
CRUNT2.C   :  auxiliary, common run-time functions
LONG.C     :  long integer functions
DOUBLE.C   :  double floating point functions
FUNC.C     :  auxiliary functions
FORMATIO.C:   printf, scanf, et al.
```

...UNDER MS-DOS AND PC-DOS

C2PRE.ASM :	run-time header for LINK
C2POST.ASM:	run-time trailer for LINK
C2I86.RTB :	run-time header file for MASM
C2I86.RTM :	run-time trailer file for MASM
MDEP.C :	#ASM/#ENDASM version of run-time
CUSTOMIZ.H:	header for the C library
CBRACK.H :	header for keyboards w/o lowercase
STDIO.H :	header for STDIO.C
STDIO.C :	standard UNIX-style I/O functions
ALLOC.C :	dynamic memory allocation functions
CRUNT2.C :	auxiliary, common run-time functions
LONG.C :	long integer functions
DOUBLE.C :	double floating point functions
FUNC.C :	auxiliary functions
FORMATIO.C:	printf, scanf, et al.

During its second pass (or third pass, in the case of CP/M-86 and MS-DOS), the compiler automatically incorporates certain precompiled, preoptimized assembly code into your program. This code is contained in the compiler's run-time library files. The other files contain the definitions of the remaining standard library functions in SuperSoft C source code. There are various methods for combining your programs with the library functions. These are described in Chapter 4.

DESCRIPTIONS OF THE SUPERSOFT C STANDARD LIBRARY FUNCTIONS

The rest of this chapter will consist of descriptions of each of the SuperSoft C standard library functions in alphabetical order. The pages in The C Programming Language by Kernighan and Ritchie referred to under most of the functions do not provide details on our particular implementation of that function but are, instead, given as a guide to using such a function in general. Since the majority of the standard library functions are provided in SuperSoft C source code and the rest in assembly code, you can discover the details of our implementation by examining that code.

In presenting each of the functions below, we have adopted a kind of shorthand for indicating the data types returned by and passed as arguments to each function. The first few lines appearing before the description of each function are the first lines of its definition in SuperSoft C, namely the declarations of the function itself and of its arguments. The type of a function is the type of the value it returns. If the type of a function is not indicated, then it does not return any defined value. The type of each of its arguments will always be indicated.

The file STDIO.H contains #DEFINE statements establishing the following predefined symbolic return values (the actual numeric values returned are shown in parentheses):

```
TRUE    (1)
FALSE   (0)
SUCCESS (0)
ERROR   (-1)
NULL    (0)
```

The file STDIO.H also contains #DEFINE statements establishing the various data types that are used by library functions. These include:

DATA TYPE	DEFINITION	EXPLANATION
BOOL	char	TRUE or FALSE
RESULT	int	SUCCESS or FAILURE
FILE	struct filedesc	file descriptor
FILESPEC	char *	filename
STREAM	FILE *	buffered file descriptor
TYPE		an arbitrary or varying type

STDIO.C contains a #INCLUDE of STDIO.H and many of the functions defined in STDIO.C will return these values.

The SuperSoft C Built-in Functions:

Grouped According to Use

SYSTEM ACCESS

bdos inp
bios outp
exit inp16
ccexit outp16
ccall cpmver
ccalla xmain
comlen lock
comline nice

CONTROL FLOW

exec setjmp
execl setexit
longjmp sleep
reset wait
pause

CONSOLE I/O

getchar putchar
gets putdec
kbhit puts
scanf printf
ugetchar

BUFFERED FILE I/O

fclose fread
fflush fscanf
fgetc fwrite
fgets getc
fopen getw
fprintf putc
fputc putw
fputs ungetc
fdopen freopen
clearerr pgetc
pputc ferror
get2b put2b
fileno

DIRECT FILE I/O

close rtell
creat seek
open tell
otell write
read access
chmod

GENERAL FILE

fabort rename
unlink link
perror errno
isfd isatty
mktemp

MEMORY INITIALIZATION

initb peek
initw poke
movmem getval
setmem

NUMERIC

abs absval
max min
rand srand
double long
assert

STRING PROCESSING

atoi xrprintf
index xrscanf
isprint rindex
isspace sprintf
ispunct sscanf
iswhite strcat
isalnum strcmp
isalpha strcpy
isascii streq
iscntrl strlen
isdigit strncat
isupper strncmp
islower strncpy
isnumeric substr
qsort tolower
swab toupper

DYNAMIC MEMORY ALLOCATION

alloc malloc
calloc realloc
brk sbrk
ubrk wrdbrk
evnbrk free
xrev isheap
topofmem

ABS

```
int abs(i)

    int i;
```

abs returns the absolute value of i (-i if i is less than zero, i otherwise). This function is not available under MS-DOS or CP/M-86 because of an assembler keyword conflict. See absval below.

BSVAL

```
int absval(i)

    int i;
```

absval returns the absolute value of i (-i if i is less than zero, i otherwise). See abs above.

CCESS

```
BOOL access(filename,mode)

    FILESPEC filename;
    unsigned mode;
```

access returns TRUE (1) if the file is accessible to the given mode; FALSE (0) otherwise.

ALLOC

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

alloc allocates a contiguous memory region of length n. Every block it allocates starts on an even address. See malloc, its identical twin.

SSERT

```
BOOL ASSERT(b)
    BOOL b;
```

assert prints "Assertion failed\n" on the console and exits if b is FALSE; otherwise, assert merely returns.

.TOI

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

atoi returns the decimal integer value corresponding to the null-terminated ASCII string pointed to by s. If this string contains characters other than leading tabs, blanks, and a minus sign (all optional) followed by consecutive decimal digits, atoi returns 0 (see K&R, pp. 39, 58).

BDOS

```
int bdos(fn, parm)

    int fn, parm
```

bdos enables users to incorporate direct BDOS function calls into programs written in SuperSoft C. Programs utilizing bdos will not always be portable. For example, BDOS function call 7 under CP/M-80 and CP/M-86 represents "Get I/O Byte", but under MS-DOS the same function number represents "Direct Console Input". On the other hand, BDOS function call 9, "Print String", represents the same function under all three systems. Users should consult their system interface guide for further information.

Under CP/M-80, bdos loads machine register C with fn and machine register pair de with parm, then calls location 5. It returns (as an integer) a single byte identical to the contents of the A register, except under BDOS function calls 12, 24, 27, 29, and 31, when it returns a word identical to the contents of the HL register pair.

Under CP/M-86, bdos loads machine register CX with fn and machine register DX with parm, then performs an INT 224 instruction. It returns (as an integer) a single byte identical to the contents of the AL register, except under BDOS function calls 12, 24, 27, 29, and 31 when it returns a word identical to the contents of the BX register. In addition, calls 27 and 31 set the global variable ESS to the value returned in register ES; call 52 returns a word identical to the contents of the ES register and also sets the global variable BXX to the value returned in BX; and call 59 returns a word identical to the contents of the AX register. For information about call 50, see BIOS, which is described later.

Under MS-DOS, bdos loads machine register AH with fn and machine register DX with parm, then performs an INT 0x21 instruction. It returns (as an integer) a single byte identical to the contents of the AL register, except under BDOS function call 6 when it returns AL, plus 0x400 if the zero flag was set. Other exceptions under MS-DOS: function calls 12 and 31 through 40 will return invalid results; calls 24 and 28 through 32 are not defined under this system.

A summary of BDOS exceptions under CP/M, CP/M-86, and MS-DOS follows:

CP/M-80 All return A except:

BDOS Function Call	Returns
12 (Return Version Number)	HL
24 (Return Login Vector)	HL
27 (Get Addr (Alloc))	HL
29 (Get R/O Vector)	HL
31 (Get Addr (Disk Parms))	HL

CP/M-86 All return AL except:

BDOS Function Call	Returns	Sets
12 (Return Version Number)	BX	
24 (Return Login Vector)	BX	
27 (Get Addr (Alloc))	BX	ESS
29 (Get Addr (R/O Vector))	BX	
31 (Get Addr (Disk Parms))	BX	ESS
50 (Direct BIOS Call)	(See bios)	
52 (Get DMA Segment Base)	ES	BXX
59 (Program Load)	AX	

MS-DOS All return AL except:

BDOS Function Call	Returns
6 (Direct console I/O)	AL and 0x400 if zero flag was set
12 (Character input with buffer flush)	Invalid Function
24, 28-32	These functions not defined under MS-DOS.
33-40	Invalid Functions

NOTE: The value that bdos returns is not sign-extended.

BIOS

CP/M-80	CP/M-86
int bios(jmpnum, bc, de)	int bios(jmpnum, cx, dx)
int jmpnum;	int jmpnum;
int c, de;	int cx, dx;

bios enables CP/M users to incorporate direct BIOS calls into programs written in SuperSoft C. Programs that call bios will not be portable beyond CP/M systems. bios sets machine register pair BC (CX under CP/M-86) to the value given in bc (cx under CP/M-86) and register pair DE (DX under CP/M-86) to the value given in de (dx under CP/M-86) and initiates the appropriate BIOS call by transferring control to the BIOS jump vector entry point specified in jmpnum. This entry point may be specified numerically or symbolically.

The appropriate mnemonics or symbolic names for each entry point (as given in the CP/M Alteration Guide and CP/M-86 System Reference Guide supplied by Digital Research, Inc.) and the numeric value for each entry point are given below:

Entry Points Under CP/M-80

BOOT -----	0	SELDSK ---	9
WBOOT ----	1	SETTRK ---	10
CONST ----	2	SETSEC ---	11
CONIN ----	3	SETDMA ---	12
CONOUT ---	4	READ -----	13
LIST -----	5	WRITE ----	14
PUNCH ----	6	LISTST ---	15
READER ---	7	SECTRAN --	16
HOME -----	8		

Under CP/M, if jmpnum is either SELDSK (9) or SECTRAN (16), bios returns the value remaining in register HL after execution of the BIOS call; otherwise, it returns the value remaining in register A without sign-extension (that is, as a value between 0 and 255).

Entry Points Under CP/M-86

INIT -----	0	SETSEC ---	11
WBOOT ----	1	SETDMA ---	12
CONST ----	2	READ -----	13
CONIN ----	3	WRITE ----	14
CONOUT ---	4	LISTST ---	15
LIST -----	5	SECTRAN --	16
PUNCH ----	6	SETDMAB --	17
READER ---	7	GETSEGB --	18
HOME -----	8	GETIOB ---	19
SELDSK ---	9	SETIOB ---	20
SETTRK ---	10		

Under CP/M-86, if jmpnum is either SELDSK (9), SECTRAN (16), or GETSEGB (18), bios returns the value remaining in register BX after execution of the BIOS call; otherwise, it returns the value remaining in register AL. Additionally, SELDSK and GETSEGB set the global variable ESS to the value returned in register ES.

BRK

```
RESULT brk(p)

    char *p;
```

brk sets the external variable name CCEDATA to the memory byte pointed to by p and returns CCEDATA, which is equivalent to the pointer value it was passed. CCEDATA is initially set to the byte immediately following the last byte in your program's external data area. Since CCEDATA is used as a base value by the other dynamic memory allocation functions, this properly initializes those functions for your program. You will almost never need to call brk in your own programs. brk returns ERROR if it could not perform the break.

ALLOC

```
char *calloc(nthings, sizeofthings)

    unsigned nthings, sizeofthings;
```

Allocates a contiguous memory region whose length equals the product of nthings and sizeofthings. Every block allocated starts on an even address. The bytes of the block contain a zero (?) value. See malloc.

CCALL

```
int ccall(addr, hl, a, bc, de)

    char *addr, a;
    int hl, bc, de;
```

ccall sets machine registers hl, a, bc, and de to the values given in hl, a, bc, and de respectively and calls the assembly language subroutine beginning at addr. ccall returns the value present in the hl register after execution of the subroutine. Programs calling ccall will not be portable beyond the 8080 series CPUs.

CCALLA

```
int ccalla(addr, hl, a, bc, de)

    char *addr, a;
    int hl, bc, de;
```

ccalla is identical to ccall except that it returns the value present in the A register after execution of the subroutine. Programs calling ccalla will also not be portable beyond the 8080 series CPUs.

As an example of the use of this function, consider the following:

```
int bdos(c, de)
 char c;
 int de;
{
        return ccalla(5, 0, 0, c, de);
}
```

The C function listed above is clearly an implementation of the function BDOS in terms of ccalla. (This is not the way BDOS is implemented.) Both ccall and ccalla may also be used to invoke assembly language subroutines of your own creation.

CCEXIT

```
ccexit(i)
    int i;
```

Returns to the operating system. i is the return code. i should be 0 if the program is completed successfully. i should be between 1 and 255, otherwise. (These values can be arbitrarily assigned to error codes.) ccexit does not close any open files.

CHMOD

```
SUCCESS chmod(mode)
    unsigned mode;
```

Sets the filemode. If the 0x80 bit is set, sets R/O mode under CP/M and CP/M-86.

0/24

CLEARERR

```
clearerr(fd)
    STREAM fd;
```

Sets the file error state to be clear. Currently there is just one global error state variable: errno (described later).

CLOSE

```
RESULT close(fd)
    FILE *fd;
```

Closes the file specified by the pointer fd to its file descriptor. close returns SUCCESS (0) if the file specified was successfully closed. close returns ERROR (-1) and does not close the file if: (1) fd does not point to a valid file descriptor, or (2) the file could not be closed due to an error at the operating system level.

close does not place an end-of-file (EOF) character in the buffer (if there is one associated with the file) and does not call fflush. If you call close for a file opened for buffered output via fopen, without first calling fflush, you will lose any data remaining in that file's I/O buffer. (See K&R, p. 163).

CODEND

```
char *codend()
```

codend returns a pointer to the byte immediately following the end of the code for the root segment of your program. Unless you have re-origined your program's external data area, the value returned by codend will point to the beginning of that area. (Not currently implemented.)

COMLEN

int comlen()

Returns the length of the command line. Available only on systems that have the command line available as an array of characters.

MLINE

char *comline()

Returns the address of the command line. Available only on systems that have the command line available as an array of characters.

CPMVER

```
int cpmver()
```

Returns the value present in register HL after execution of a call to BDOS function number 12 (See the CP/M Interface Guide supplied by Digital Research Inc.). This return value, which is not sign-extended, is 0 if the calling program is running under CP/M versions released prior to Version 2.0, 0x0020 if it is running under CP/M Version 2.0, in the range 0x0021 to 0x002F under versions subsequent to 2.0, and 0x0100 under MP/M. This function is useful in writing C programs to run under CP/M or MP/M--independent of version number. Programs calling cpmver will not be portable beyond CP/M, CP/M-86, and MP/M.

REAT

```
FILE *creat(fspec,mode)
    FILESPEC fspec;
    unsigned mode;
```

Creates a file on disk with the file specification given in fspec and opens it for output. If the value of mode has the 0x80 bit set, then a readable-writeable file is created; otherwise, a read-only file is created. Any existing file with the same specification will be deleted.

creat, if successful, returns a pointer to a valid file descriptor for the file specified. You should store this pointer for subsequent use in your program. creat returns ERROR (-1) and does not create or open any file if: (1) not enough user memory is available to store a new file descriptor, (2) the file specification given is invalid, or (3) the file could not be created and opened due to an error at the operating system level (See K&R, p. 162).

ENDEXT

```
char *endext()
```

Returns a pointer to the byte immediately following the last byte in your program's external data area. This value should be identical to the initial value of CCEDATA. (Not currently implemented.)

ERRNO

```
int errno
```

Not a function, but an external variable that will be set to an error code whenever an I/O error occurs. Note that it is not automatically cleared if no error occurs. There is no zero error message, so clearing errno is the accepted way to 'preset' it for picking up error values. perror, described later, prints an error message on the console, given a non-zero error.

DOUBLE arithmetic

The file DOUBLE.C contains a double precision floating point aritmetic package. These functions perform arithmetic on BCD numbers which are 8 bytes long. All variables used by this package should be declared DOUBLE or LONG FLOAT. The functions are passed the address of each argument. Usually the functions return the destination address so that calls can be nested in the following manner:

```
Badd (&d, Bmul(&a, &b, &c), &e)
```

In this example d=b*c+e.

See Appendix G for descriptions of these functions.

EVNBRK

```
char *evnbrk(n)

    unsigned n;
```

Performs identically to sbrk with the exception that it always returns an even value (a pointer with its low order bit zero). It accomplishes this by skipping one byte if sbrk returns an odd value for the given n argument. If successful, evnbrk returns a pointer to the first memory location in the block added; otherwise, it returns a value of ERROR (-1). Failure can be caused by: (1) overlapping a stored program, (2) overlapping the run-time stack, or (3) exceeding the available memory. Note that n is unsigned.

Also see sbrk, ubrk, wrdbrk, and brk.

EXEC

```
RESULT exec(fspec)

    FILESPEC fspec;
```

Performs an interprogram jump to the file specified in the null-terminated string pointed to by fspec. exec is a call to execl with two arguments: execl(fspec,0). It is included for compatibility with BDS C. See execl below.

EXECL

```
RESULT execl(fspec, arg0, arg1, arg2,...argn)

    FILESPEC fspec;
    char *arg0, *arg1, *arg2, ..., *argn;
```

Performs an interprogram jump to the file specified in the null-terminated string pointed to by fspec. That is, it loads and executes the code that file fspec is assumed to contain. Interprogram jumping is sometimes referred to as program chaining. execl enables you to successively execute a series of programs, with each program in the series overlaying the memory image of the preceding program.

Command line parameters may be passed to the invoked program in a series of null-terminated strings pointed to by arg1, arg2, ..., argn. The last argument (argn) must be zero. This is for compatibility with UNIX. arg0 is ignored under CP/M due to an unfortunate feature of that operating system. execl constructs a command line from the strings pointed to by its arguments (under CP/M and MS-DOS this means interleaving spaces between the arguments).

Data may also be passed from the invoking program to the invoked program within files or through the external data area. To pass data within a file, the invoking program should close the file and the invoked program should reopen it. To pass data (including open file descriptors) through the external data area, the origin of that area must be the same for both programs. On CP/M-86 and MS-DOS, this could be accomplished by having both programs define the exact same set of globals.

execl returns ERROR (-1), since any return from execl is an error. This will happen if the file fspec does not exist or could not be read for any reason, preventing overlaying of the memory image of the invoking program and the execution of the invoked program. The invoked program can jump back to the invoking program via another call to execl.

execl is a list function. (See Chapter 2, USING LIST FUNCTIONS.)

EXIT

```
exit(i)
    int i;
```

Transfers control from the program back to the operating system (causes an exit to system level). exit never returns. It calls fclose to close stdout and stderr (the redirected standard output file and standard error file. See xmain). No other files are flushed (written to disk) nor are any other open files closed. (See K&R, p. 154.) The parameter i is currently ignored under CP/M and MS-DOS. If an exit to the operating system is desired without any I/O occurring, call ccexit instead of exit.

:TERNS

```
char *externs()
```

Returns a pointer to the first byte in the external data area of your program. Unless you have re-origined this area this value will be the same as that returned by codend. (Not currently implemented.)

FABORT

```
RESULT fabort(fd)

    FILE *fd;
```

Frees the file descriptor allocated to an open file without closing that file. The file is specified by the pointer fd to its file descriptor. Calling fabort will have no effect on the contents of a file opened for input only, but calling it for a file opened for output may cause the loss of some or all of the data written to that file. (We have included this function for the sake of compatability with BDS C--we do not recommend that you use it.)

fabort, if successful, returns SUCCESS (0). fabort returns ERROR (-1) if fd does not point to a valid file descriptor.

CLOSE

```
RESULT fclose(fd)

    STREAM fd;
```

Closes a file opened for buffered output via fopen. The file is specified by the pointer fd to its file descriptor. fclose places an end-of-file (EOF) character at the current position in the file's I/O buffer and calls fflush before closing the file.

fclose returns SUCCESS (0) if the file specified was successfully closed. fclose returns ERROR (-1) and does not close the file if: (1) the file specified was not opened for buffered output via fopen, (2) fd does not point to a valid file descriptor, or (3) the file could not be closed due to an error at the operating system level (see K&R, p. 153).

FDOPEN

```
STREAM fdopen(fd, mode, buffer_size)

    FILE *fd;
    char *mode;
    unsigned buffer_size;
```

Converts from an unbuffered file descriptor (FILE *) to a STREAM (buffered file descriptor). mode must be compatible with the read/write attributes of fd. buffer_size is the size of the buffer to be used by the STREAM descriptor. Note that it can be used to change the size of a STREAM descriptor if used in conjunction with fileno, as in:

```
strm = fdopen(fileno(strm),mode,buffer_size);
```

Returns NULL if unsuccessful.

RROR

```
int ferror(fd)

    STREAM fd;
```

Returns the error value for the given stream. The error value, which is read from errno, is set on error. To clear it, use clearerr (described earlier). Currently there is only one error value.

FFLUSH

```
RESULT fflush(fd)

    STREAM fd;
```

Flushes or writes to file the current contents of the I/O buffer associated with a file opened for buffered output via fopen. The file is specified by the pointer fd to its file descriptor. The size of the I/O buffer is set when the file is opened. After a call to fflush, the file I/O pointer will point to just past the last byte accessed, as expected.

fflush returns SUCCESS (0) if the buffer was successfully written to file. (Calling fflush when the buffer is empty has no effect other than to return SUCCESS.) fflush returns ERROR (-1) and does not flush the buffer if: (1) the file was not opened for buffered output via fopen, (2) fd does not point to a valid file descriptor, or (3) the entire contents of the buffer could not be written due to an error at the operating system level (see K&R, p. 166).

FTC

```
int fgetc(fd)

    STREAM fd;
```

Identical to getc (described later). Guaranteed to be a function rather than a preprocessor macro.

FGETS

```
char *fgets(s, n, fd)

    char *s;
    unsigned n;
    STREAM fd;
```

Reads a maximum of n-1 characters (bytes) from a file opened for buffered input via fopen into the string beginning at s. The file is specified by the pointer fd to its file descriptor. fgets will not read past a newline or more than n-1 characters, whichever comes first. fgets then appends a null character to the characters read to create a null-terminated string.

fgets, if successful, returns a pointer to this string (identical to the value passed in s). fgets returns a null pointer value (0) if: (1) fd does not point to a valid file descriptor, (2) the file could not be read due to an error at the operating system level, or (3) the end of the file has been reached (see K&R, p. 155).

FILENO

```
FILE *fileno(sfd)

    STREAM sfd;
```

Returns the file descriptor associated with the STREAM sfd.

FOPEN

```
STREAM fopen(fspec, mode, buffer_size)

    FILESPEC fspec;
    char *mode;
    int buffer_size;
```

Creates and/or opens a file for buffered I/O with the file specification given in fspec. It uses open (see the open function later in this chapter) to actually open the file. Thus full file specifications may be used. If fopen is successful, it returns a pointer to a valid file descriptor for the file specified. You should store this pointer for subsequent use in your program. fopen returns NULL (0) and does not create or open any file if: (1) the file mode is unrecognized, (2) not enough user memory is available for a new file descriptor, (3) the file specification given is invalid, or (4) the file cannot be opened or created due to an error at the operating system level.

For mode you must specify one of the following: "w", "a", or "r". Whichever you specify determines the file's I/O mode, as indicated in the following table:

"w"	write-only mode
"a"	write-only mode, append output to end of file
"r"	read-only mode

If a file's I/O mode is either "w" or "a", it is said to be open for buffered output. If a file's I/O mode is "r", it is said to be open for buffered input.

The value you specify for buffer_size determines the size of the I/O buffer associated with the file. This value is best set as a positive integral multiple of the system record size. (A system record is the minimum unit of data transferred during file I/O operations. Under CP/M, the system record size is 128 bytes. Under UNIX it is 512 bytes. Consult your operating system's documentation for further information.) A pointer to the first byte in this I/O buffer is stored in the file descriptor. Thus, only a pointer to this file descriptor need be passed to any of the other buffered file I/O functions (see K&R, pp. 151, 167).

FPRINTF

```
RESULT fprintf(fd, format, arg1, arg2,...)

    FILE *fd;
    char *format;
    TYPE arg1;
    TYPE arg2;
      •
      •
      •
```

Identical to printf except that instead of writing to the standard output it writes its formatted output string to the I/O buffer associated with a file opened for buffered output via fopen, beginning at the current location in that buffer. Whenever that buffer becomes full, it is automatically flushed (i.e., its entire contents are written to the file). The file is specified by the pointer fd to its file descriptor.

fprintf is a list function. See p. 43.

fprintf returns SUCCESS (0) if its entire output string was successfully written to the buffer. However, due to the buffering of file I/O operations, such a return value cannot guarantee that this same string will be successfully written to the file, since errors resulting from and affecting the outcome of a particular call to fprintf may not become apparent until some later function call causes that file's I/O buffer to be flushed. fprintf returns ERROR (-1) if: (1) fd does not point to a valid file descriptor, (2) the file was not opened for buffered output via fopen, (3) the file could not be written due to an error at the operating system level, or (4) the entire string could not be written to the file due to a lack of disk space (see K&R, p. 152).

FPUTC

```
RESULT fputc(c, fd)

    char c;
    STREAM fd;
```

Writes the character c to the STREAM fd. Identical with putc (described later), except that it is guaranteed not to be a preprocessor macro.

PUTS

```
RESULT fputs(s, fd)

    char *s;
    STREAM fd;
```

Writes the null-terminated string pointed to by s to the I/O buffer associated with a file opened for buffered output via fopen, beginning at the current location in that buffer. Whenever that buffer becomes full, it is automatically flushed (i.e., its entire contents are written to the file). The file is specified by the pointer fd to its file descriptor. For each newline character ('\n') appearing in the string, a carriage return and a newline ("\r\n") are written to the buffer. The terminal null character is not written.

fputs returns ERROR (-1) and does not write the string if: (1) fd does not point to a valid file descriptor, (2) the file was not opened for buffered output via fopen, or (3) the file could not be written due to an error at the operating system level. Otherwise, fputs returns the number of bytes actually written to the buffer minus the number of carriage return characters inserted. However, due to the buffering of file I/O operations, such a return value does not guarantee that those same bytes will be successfully written to the file, since errors resulting from and affecting the outcome of a particular call to fputs may not become apparent until some later function call causes that file's I/O buffer to be flushed (see K&R, p. 155).

FREAD

```
int fread(buf, fsizeofitem, nitems, fd)

    TYPE *buf;
    unsigned sizeofitems;
    unsigned nitems;
    STREAM fd;
```

Reads a number of items (nitems) from STREAM fd. The size of an individual item is sizeofitems. buf is the address to read into. fread returns the number of items read or ERROR(-1) on error. freads can be interspersed with getc, fgetc, get2b, and getw (all described in this chapter). Alignment is of no concern. Also see read described later.

REE

```
free(p)

    char *p;
```

Frees a block in memory previously allocated by a call to alloc. The argument p, which should be identical to the value returned by that call to alloc, is a pointer to the first memory location in the block. Allocated blocks may be freed in any order. To call free with an argument not previously obtained by a call to alloc is a serious error (see K&R, pp. 97, 177).

FREOPEN

```
STREAM freopen(fspec, mode, buffer_size, strm)

    FILE *fspec;
    STREAM strm;
    char *mode;
    unsigned buffer_size;
```

Redirects strm as if strm=fopen(fspec,mode,buffer_size) had been called. Returns NULL if unsuccessful.

SCANF

```
RESULT fscanf(fd, format, arg1, arg2,...)

    FILE *fd;
    char *format, *arg1, *arg2, ...;
    TYPE *arg1;
    TYPE *arg2;
      .
      .
      .
```

Identical to scanf except that the input string is read from the I/O buffer associated with a file opened for buffered input via fopen rather than from the standard input. The file is specified by the pointer fd to its file descriptor. fscanf begins reading at the current position in the I/O buffer. It stops reading when it has successfully assigned values to the bytes corresponding to each item listed in the format string or when it has reached the end of the file--whichever comes first.

fscanf is a list function. See USING LIST FUNCTIONS at the end of Chapter 2.

If no errors occur, fscanf returns the number of values successfully assigned. fscanf returns ERROR (-1) and performs no input if: (1) fd does not point to a valid file descriptor, (2) the file was not opened for buffered input via fopen, or (3) the file could not be read due to an error at the operating system level (see K&R, p. 152).

FWRITE

```
int fwrite(buf, sizeofitem, nitems, fd)

    TYPE *buf;
    unsigned sizeofitem;
    unsigned nitems;
    STREAM fd;
```

Writes a number of items (nitems) into STREAM fd. The size of an individual item is sizeofitems. buf is the address to write from. fwrite returns the number of items written or ERROR(-1) on error. fwrites can be interspersed with putc, fputc, put2b, and putw (all described in this chapter). Alignment is of no concern. Also see write described later.

ETC

```
int getc(fd)

    STREAM fd;
```

Returns one character (byte) as an integer (between 0 and 255 inclusively) in sequence from a file opened for buffered input via fopen. The file is specified by the pointer fd to its file descriptor. Carriage returns and linefeeds are returned explicitly. getc returns ERROR (-1) if: (1) the file was not opened for buffered input, or (2) the end of the file has been reached (see K&R, pp. 152, 166).

GETCHAR

```
char getchar()
```

Returns the next character from standard input (the CP/M or MS-DOS CON: device--usually the console keyboard). If getchar encounters a Control-Z (CP/M's EOF marker), it returns an error(-1). (See K&R, pp. 13, 40, 144, 152, 161, 162.)

GETS

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

Reads the next line from standard input (the CP/M or MS-DOS CON: device--usually the console keyboard) into the string beginning at s. gets replaces the newline character ('\n') or the carriage return/newline combination ("\r\n") that terminates the input line with a null character to create a null-terminated string. Since gets does not test whether the string beginning at s is long enough to contain the input line, you should define this string such that it can contain the longest input line you could reasonably expect.

GETW

```
int getw(fd)

    STREAM fd;
```

Returns one integer in sequence from a file opened for buffered input via fopen. The file is specified by the pointer fd to its file descriptor. Carriage returns and linefeeds are returned explicitly. getw returns ERROR (-1) if: (1) the file was not opened for buffered input, or (2) the end of the file has been reached, or (3) if the integer equal to ERROR (-1) appears in the input file. Thus errno should be checked for true error conditions. Calls to getc may be interspersed with calls to getw. Information read by getc and getw may be written by putc and putw. There need not be any particular alignment of information in the input file.

:T2B

```
int get2b(fd)

    STREAM fd;
```

Returns a two byte quantity from a buffered input stream. get2b is similar to getw and fread, except that it is invariant with respect to byte ordering.

INDEX

```
char *index(s, c)

    char *s, c;
```

Returns a pointer to the first occurrence of the character c in the string beginning at s. index returns a null pointer value (0) if c does not occur in the string. See rindex described later in this chapter.

NITB

```
initb(array, s)

    char *array, *s;
```

Permits relatively convenient initialization of character arrays. It should be passed two parameters: the first, array, should be a pointer to an array of characters; the second, s, should be a pointer to a null-terminated string of ASCII characters representing decimal integer values separated by commas. When called, initb converts each decimal integer value in the string beginning at s, in sequence, to a binary integer value and assigns the least significant 8 bits of that value to the corresponding element in the character-array pointed to by array.

If there are n integer values in the string and greater than n elements in the array, only the first n elements of the array will be assigned values and the contents of the remaining elements will be unaltered. If there are n integer values in the string and less than n elements in the array, bytes beyond the end of the array will be assigned values as if they were elements of the array and data may be overwritten in error. It is the programmer's responsibility to prevent or provide for these situations.

INITW

```
initw(array, s)

    int *array;
    char *s;
```

Permits relatively convenient initialization of integer arrays. It should be passed two parameters: the first, array, should be a pointer to an array of integers; the second, s, should be a pointer to a null-terminated string of ASCII characters representing decimal integer values separated by commas. When called, initb converts each decimal integer value in the string beginning at s, in sequence, to a binary integer value and assigns that value to the corresponding element in the integer array pointed to by array.

If there are n integer values in the string and greater than n elements in the array, only the first n elements of the array will be assigned values and the contents of the remaining elements will be unaltered. If there are n integer values in the string and less than n elements in the array, bytes beyond the end of the array will be assigned values as if they were elements of the array and data may be overwritten in error. It is the programmer's responsibility to prevent or provide for these situations.

NP

```
char inp(port)

    int port;
```

Returns the byte value present at the specified input port after execution of a byte IN machine instruction for that port. (This function is available only on machines for which the byte IN instruction or equivalent makes sense.)

ISASCII

```
BOOL isascii(c)

     char c;
```

Returns TRUE (1) if c is an ASCII character; otherwise, it returns FALSE (0).

ISATTY

```
BOOL isatty(fd)

     FILE *fd;
```

Returns TRUE(1) if this file descriptor is a terminal; otherwise, FALSE(0). Returns TRUE if the file descriptor refers to the console ("CON:").

ISCNTRL

```
BOOL iscntrl(c)

     char c;
```

Returns TRUE (1) if c is an ASCII control character; otherwise, it returns FALSE (0).

INP16

```
int inp16(port)
    int port;
```

inp16, not available under CP/M-83, returns the value present at the specified input port after execution of a 16-bit IN machine instruction for that port. (This function is available only on machines for which the 16-bit IN instruction or equivalent makes sense.)

ISALNUM

```
BOOL isalnum(c)
    char c;
```

Returns TRUE (1) if c is an ASCII alphanumeric character; otherwise, it returns FALSE (0).

0,24

ISALPHA

```
BOOL isalpha(c)
    char c;
```

Returns TRUE (1) if c is an ASCII alphabetical character; otherwise, it returns FALSE (0) (see K&R, pp. 127, 156).

ISDIGIT

```
BOOL isdigit(c)
    char c;
```

Returns TRUE (1) if c is an ASCII character representing one of the decimal digits 0-9; otherwise, it returns FALSE (0) (see K&R, pp. 127, 156).

ISFD

```
BOOL isfd(fd)
    FILE *fd;
```

Returns TRUE if fd is a valid file descriptor; FALSE otherwise.

ISHEAP

```
BOOL isheap(ptr)
    char *ptr;
```

Returns TRUE iff ptr points to a data area returned from malloc. Must be called only after the first call to malloc.

CHAPTER 4

Combining Source Code and Altering Assembly Code

METHOD 1: THE RELOCATING ASSEMBLERS, MODULES, AND LIBRARIES

The C language is constructed so that variables and functions can be compiled separately into relocatable modules and then linked together to form working programs using features that are available with most linkers. (See Chapter 2, "Using the SuperSoft C Compiler".) SuperSoft C allows the C EXTERN storage class for external (globally accessible or global) variables and functions. In the special case of functions, those not previously declared are automatically declared as external. SuperSoft C translates the EXTERN storage class into the proper assembler directive to allow the references to be resolved at link time.

There must be exactly one definition of each globally accessible variable or function. Every external with the same name will be bound to the same definition in the program. The various declarations should agree with the definitions. Although the C compiler will check all declarations in a given source file for consistency, the linker may not be able to check fully for consistency. A variable may be declared as many times as desired with the EXTERN storage class explicitly stated, but must be declared exactly once without the EXTERN storage class and outside of a function body. A function must appear exactly once with a function body and without the EXTERN keyword.

Of course, variables must differ within the first few characters to appear unique to an assembler or linker. In the case of M80, names are truncated at six or seven characters depending upon the version. In CP/M-80 .REL format, names are limited to seven characters. Also, many assemblers (including M80) translate lower case characters in a name into upper case, effectively lowering the number of unique names. For portability reasons at least, case should not be used to distinguish variable or function

names. For example, a program should not contain 'THelp' and 'TheLP' as unique variable names.

IT IS HIGHLY RECOMMENDED THAT SUPERSOFT C BE USED WITH A RELOCATING ASSEMBLER, rather than an absolute assembler such as ASM or ASM86. Relocation, available to users with a relocating assembler package such as Digital Research's RMAC package, is by far the easiest and most usable method of incorporating both user and library functions into a final executable (COM, CMD, or EXE) file.

Linking is usually faster than compilation. Once linked, a file may be used over and over again. Linking also allows groups of people to work on a program, giving any one of those people the ability to combine and execute the program, but disallowing each access to the other's source. This last advantage is extremely important in a large project, or in a small project completed over time, especially on systems that do not have adequate security between users.

The use of libraries is the easiest method of combining variables and functions into working programs. It requires a librarian as well as a linker. In the case of CP/M-80, the linker might be SuperSoft's ELINK and the librarian might be LIB80. You must compile each relocatable module, follow the librarian instructions to create a library that includes all modules, and then specify that library to link a program.

Most linkers do not make much of a distinction between a relocatable module and a library. Either can be specified to build an executable file. The difference is mainly in the way that they are combined. Relocatable modules are loaded into the executable file in their entirety. Libraries are built from multiple relocatable modules. They are searched by the linker in order to include only those modules which satisfy external references not previously defined.

The SuperSoft C compiler under CP/M-80 comes with a library of standard functions (LIBC.REL) built from the files FORMATIO.C, STDIO.C, STDIO.H, FUNC.C, CRUNT2.C, MDEP.C, LONG.C, DOUBLE.C, and a number of .ASM files. The +SUBM option of the C compiler (described below) and the SUBMIT file LIBC.SUB was used to build this library. Version 3.44 of M80 and LIB80 from Microsoft was used to assemble and collect the library. This library contains all of the functions that are described in Chapter 3. You could add your own functions, or alter the supplied functions by adding relocatable module names to the LIBC.SUB file and compiling the appropriate source files.

If you are using another linking system, you may or may not be able to make use of the LIBC.REL library. ELINK is compatible with LIBC.REL; the LINK80 may not be. If you have any problems, you should rebuild the library, as described below. This is not a task to take lightly. It requires a considerable amount of disk space (mostly directory space for a lot of small files) and time. It may take hours.

The method described below may also be used to build your own unique libraries. In building a library you will probably have to sort the modules into a dependency order: if module A refers to module B, then module A will have to be before module B in the library. If this is too much trouble, you can try searching the library more than once in order to pick up all references. Most linkers allow the searching of more than one library, so adding another library to a linker command line is the easiest and sanest method for adding functions to your C library. Note, however, it is not necessary to compile source files into libraries. For many uses, leaving them as relocatable modules is adequate. The libraries are mainly useful for the selection of variables or functions rather than the inclusion of whole relocatable modules into a program.

The SuperSoft C compiler contains a method for creating libraries in which each function can be separated and loaded only if needed. In order to do this, specify the +SUBM flag on the command line of both the first (CC) and second (C2 or CO2COD) passes of the compiler. The output of the second pass is altered: no longer is the output a single file with the extension .ASM. Instead, it is a file per function, each file consisting of the name of the function and the extension .MAC (under CP/M, MS-DOS, and PC-DOS these names have '_' removed and are truncated to eight characters in order to be compatible with the native assembler/linker systems). Also a file is created which contains directives to form the relocatable modules (named .REL under CP/M-80 and .OBJ under MS-DOS). This file is named $.SUB. Under CP/M this file is directly executable via SUBMIT and it will assemble the appropriate relocatable modules. After executing the $.SUB file, the librarian must be used to collect the functions into a library.

The file LIBC.SUB contains a SUBMIT file that will create the standard SuperSoft C library. This is not an ordinary SUBMIT file, but requires recursive capability. The SuperSoft program SH.COM (supplied with the C compiler under CP/M-80) will allow recursion and must be used with LIBC.SUB.

SH is similar to, but more powerful than, SUBMIT. SUBMIT allows nine arguments: $1 through $9. SH allows an indefinite number of arguments: $0 contains the name of the SUBMIT file, and $1, $2, $3, ..., $9, $10, $11, ... contain

the arguments. As a result, $11 would be interpreted differently by SH and SUBMIT. SH interprets it as the eleventh argument and SUBMIT interprets it as the first argument followed by a 1.

An SH file may contain other lines invoking SH. Each time SH is invoked it stacks the old $$$.SUB file by renaming it as $nn.SUB, where nn is replaced by a unique decimal number. Then it places at the end of the SUBMIT file a line of the form 'SH -E $nn.SUB', which unstacks the old $$$.SUB file by erasing $$$.SUB and renaming $nn.SUB to be $$$.SUB.

Additional features of SH: (1) the SHIFT operator shifts the argument list "down" by one ($1 becomes $0, $2 becomes $1, and so on). (2) SH strips comments (lines preceded by ';') from the $$$.SUB file to allow larger $$$.SUB files to work. '$*' in a .SUB file expands to be the argument list, each argument separated by a space and '$:,' expands to be the argument list, each argument separated by a ','. (3) Blank lines, partial last lines, and large SUBMIT files cause SUBMIT to execute in an uneexpected manner; there is no such problem with SH. (4) SH can be used from any drive; SUBMIT can be used only from drive A.

METHOD 2: THE #INCLUDE DIRECTIVE

The #INCLUDE preprocessor directive is able to incorporate the entire file containing the desired variables or functions into your program. A good practice is to place any #INCLUDE directives for the standard library function files in your program's preamble, immediately after any external data definitions.

The sample programs given out with SuperSoft C make use of this method of building programs from functions. It is assumed that you are using an absolute assembler such as CP/M-80's ASM. The use of the #INCLUDE directive is the least common denominator: the sample programs as supplied will work with either an absolute assembler or a relocating assembler. At the top of the sample programs are #INCLUDE statements for the appropriate library modules. For example SAMP1.C includes CRUNT2.C and FUNC.C.

A disadvantage to #INCLUDEing all necessary library files into your program is that the time to compile your program will include the time to compile all of the included code. It may also make your program unnecessarily large by incorporating into it functions that are never called. If you are using a relocating assembler, you will probably want to compile the sample programs in a modified manner. Remove the #INCLUDE statements from the sample programs. Then compile as specified in Chapter 2.

The result will be about 25 percent of the size of the executable file that you would get if you used an absolute assembler. Also, compilation time will be a small percent of the orginal compilation time.

METHOD 3: THE CC COMMAND LINE FILENAME LIST

If you list a number of filenames on the command line for CC, the first pass of the compiler, all the files named will be parsed, in the order listed, as if they were one file. The output will still be placed in a single output file. Thus you may incorporate any C source code file into your program simply by listing its filename in the command line for CC. The CC command line that would incorporate all the SuperSoft C standard library functions into the program Y.C is:

```
CC Y.C CRUNT2.C STDIO.C ALLOC.C FUNC.C FORMATIO.C
```

This method has all the advantages and disadvantages of the previous one (using #INCLUDES), but is more flexible. To change the files incorporated into your program, rather than changing your program, you would just type a different set of filenames in the CC command line.

METHOD 4: PRECOMPILATION AND INSERTION IN A HEADER FILE

Any of the library files which are in SuperSoft C source code may be compiled and the resulting file inserted, with minor editing, into the compiler's run-time library. This increases the speed with which you can compile programs that incorporate these functions. However, this method shares the same disadvantage of possibly excessive program size with the two other all-inclusive methods described above.

Of course, you are not limited to including the libraries as a whole. If you have an idea of the functions you will commonly use in your programs, you may create your own version of the run-time library by precompiling and inserting these functions into your header file. Nor is it necessary that these functions be the functions supplied with the compiler: they may be of your own creation. This method can save you considerable time in compiling your programs and considerable program space by eliminating unnecessary functions.

SuperSoft's C allows the user to add code to the code generator's header file, the file that is automatically placed at the beginning of C programs. (The header provides necessary machine dependent functions.) Any function already in assembly source and included in the header file need not be recompiled and re-optimized each time the rest of the program is compiled. This is particularly useful when using an absolute assembler like CP/M-80's ASM or CP/M-86's ASM86.

To use this method, the user compiles the necessary routine(s), producing an assembly language file. This assembly source file is then added into the header file.

Below is a detailed, step by step, description of how the CRUNT2.C file of subroutines can be moved into the runtime routines under CP/M-80, thus eliminating the need to use a #INCLUDE "CRUNT2.C" directive (or a link of a CRUNT2 object file).

1) We are ready to compile the CRUNT2.C. Issue the following commands:

```
A>CC CRUNT2.C
.
.
.
A>C2 CRUNT2.COD +ASM +ZA -RH -RT
```

The '+ZA' options tells the code generator to start all labels with the letter 'A'. This avoids any conflict with the code produced by normal compilations which start each label with a 'C'. The -RT and -RH options prevent the inclusion of the header and trailer files.

2) Next, REName the file to CRUNT2.LIB, in preparation for merging into the C2.RH file:

```
REN CRUNT2.LIB=CRUNT2.ASM
```

3) A minor amount of editing must be done before the CRUNT2.LIB file can be merged. Type the following:

```
A>ED CRUNT2.LIB
*#a
*3T
```

you will then see

```
;C Optimizer V1.2
        ORG     256
;C Compiler V1.2
```

When you list the top of the file on your screen, your file should be the same (except for version numbers) as that listed here. You must now delete the following line:

```
ORG     256
```

4) After deleting this line, exit the edit with an 'E' (thus saving your changes).

METHOD 5: CUT AND PASTE

This method is simple in concept but laborious in execution. The idea is to create, from the standard library function files, a file or files containing only those data and function definitions required by your program (cut) and then merge the files thus created into your program before compiling it (paste).

The advantage of this approach is that both the source and object code for your program will be as small as your program's implementation will allow. Of course, you must be careful that all data and function definitions required for execution of your program are present and intact. Clearly, this can be time-consuming, and some troublesome bugs can creep in during the process.

However, it may be essential in some applications that your programs make the most efficient possible use of the available memory. The size of some programs alone may approach or exceed the memory capacity of your system, while others may require correspondingly large amounts of data storage during execution.

You may also find yourself repeatedly using a particular subset of the standard library functions, mingled with some functions of your own. In this situation, you could use this method as a first step toward creating your own customized set of SuperSoft C standard library functions.

You should not attempt to use this method unless you are quite familiar with C and with your own operating system. What follows is only a very general outline of the steps necessary to the cut and paste method.

1) Make a copy on disk of each file containing data or function definitions that you wish to incorporate into your program.

2) Using your editor, delete from each file those data and function definitions you do not wish to incorporate into your program. Be careful not to delete the definitions of those functions called by the functions your program calls or any of the data definitions required by any function that you intend to incorporate.

3) If you now have more than one file containing the definitions of interest, you may leave them as separate files or concatenate them as you choose.

5) Now you must edit the C2.RH file. Your screen should show:

```
A>ED C2.RH
*#A
*F** insert any user^Z0lt
 ; ***** insert any user code here *****
*rcrunt2.lib
*e
```

This will cause the editor to read in the CRUNT2.LIB file.

After exiting the editor with the 'E' command, all of the functions contained in CRUNT2.C will have been moved into the system runtime file C2.RH. This means that you will no longer have to use the #INCLUDE "CRUNT2.C" directive. However, since you will always be loading these functions even if they are not needed, there may be a need for a different C2.RH for different programs.

4) You are again faced with the choice of how to incorporate the files you have created into your program. You may use any one of the methods previously described for incorporating the SuperSoft C standard library function files. Your programs will now contain only the data and function definitions they require. Another option is to use your editor to insert the definitions you have created into your program at appropriate places.

HOW TO REORIGIN THE CODE GENERATED BY THE COMPILER

A relocating assembler frees programmers from concerning themselves with the actual locations at which their programs execute or fetch data. It does this by generating object code in which memory addresses appear as displacements from some relative point. These relative points can be either externally defined references or relative program origins. An external reference uses a symbolic label referring to a location in another module. For example, an external reference in module A is a way to reference a location in module B without knowing the absolute address of that location. The relative program origin is the beginning of a particular module, relative to which all of its internal addresses are calculated. Relative addresses are fixed when the program is linked.

Assemblers will support some means of specifying the absolute (or relative) origin for your code. In the case of CP/M's absolute assembler, ASM, the ORG instruction accomplishes this function. The argument of an ORG statement is used by ASM as the origin for the code that it precedes--up to the next ORG statement or the end of the file. The ORG instruction can also be used to establish different relative and absolute origins for each program segment or data area. This capability is particularly important when you wish to load your program into ROM but your data into RAM. If a program you wish to load into ROM requires writeable data areas, at least two ORG statements must appear in your file: one at the beginning of your program specifying its absolute origin in ROM and another at the beginning of the writeable data area specifying its absolute origin in RAM.

The final assembly code output generated by this compiler contains two ORG statements, appearing (at the beginning of the program and data areas respectively) as follows:

```
ORG *   ; REORIGIN PROGRAM HERE
.
.
.
ORG *   ; REORIGIN DATA HERE
.
.
.
```

The argument '*' signifies the current location, but the ORG statement is not activated until you insert the values for these arguments appropriate to your situation.

A NOTE ON THE GENERATED CODE: The code produced by the compiler requires that the heap and stack grow toward each other. This usually means that the stack will be at larger numbered locations than the heap. This relationship should be maintained even if the code produced by the compiler is going to be ROMmed.

ISLOWER

```
BOOL islower(c)

     char c;
```

Returns TRUE (1) if c is an ASCII lower case alphabetical character; otherwise, it returns FALSE (0). (see K&R, p. 156).

ISNUMERIC

```
BOOL isnumeric(c, radix)

     char c;
     int radix;
```

Returns TRUE (1) if c is an ASCII character representing a valid digit in the number system with the base specified in radix; otherwise, it returns FALSE (0). For example,

```
isnumeric('A',15)
```

returns TRUE. isnumeric is defined only if 1 < radix < 36.

ISPRINT

```
BOOL isprint(c)

     char c;
```

Returns TRUE (1) if c is a printable ASCII character; otherwise, it returns FALSE (0).

ISPUNCT

```
BOOL ispunct(c)

    char c;
```

Returns TRUE (1) if c is an ASCII character representing a punctuation mark; otherwise, it returns FALSE (0).

ISSPACE

```
BOOL isspace(c)

    char c;
```

Returns TRUE (1) if c is an ASCII character representing a space, a tab, or a newline; otherwise, it returns FALSE (0). (This function is included for compatability with BDS C; the standard UNIX C function is iswhite. However, see K&R, p. 156.)

ISUPPER

```
BOOL isupper(c)

    char c;
```

Returns TRUE (1) if c is an ASCII upper case alphabetical character; otherwise, it returns FALSE (0) (see K&R, pp. 145, 156).

ISWHITE

```
BOOL iswhite(c)

    char c;
```

Returns TRUE (1) if c is an ASCII character representing a space, a tab, a newline, or a formfeed; otherwise, it returns FALSE (Ø).

BHIT

```
BOOL kbhit()
```

Tests whether a character has been typed on the console keyboard--returning true if it has, FALSE if it has not. More precisely, kbhit returns true (non-zero) if a character is present at the standard input (the CP/M or MS-DOS CON: device--usually the console keyboard); otherwise, it returns FALSE (Ø). This function is not available on systems that do not have such a function (such as UNIX).

LINK

```
link(ofspec, nfspec)

    FILESPEC ofspec;
    FILESPEC nfspec;
```

Makes nfspec into a synonym for the filename nsfspec. Not possible under CP/M or MS-DOS. Under these systems, link operates just like rename (described later).

LOCK

lock()

Locks a process in fast memory. A no-op under CP/M and MS-DOS.

)NG arithmetic

The file LONG.C contains a machine independent LONG integer arithmetic package. These functions perform arithmetic on 32-bit two's complement integers. All variables used by this package should be declared LONG or UNSIGNED LONG. The functions are passed the address of each argument. Usually the functions return no value.

See Appendix H for descriptions of the long integer functions.

LONGJMP

```
longjmp(savearea, i)

    int savearea[];
    int i;
```

Restores the program state from the savearea. (savearea should be 6 bytes long on the 8080 and 8086 series.) The program state includes all register variables, the return program counter, and the stack pointer. savearea should have been previously used as an argument to setjmp (described later). Upon a call to longjmp with the same savearea, the state is restored, effectively appearing as if a return from setjmp has occurred, with the return value being i. longjmp is a generalized version of reset.

MALLOC

```
char *malloc(n)

    unsigned n;
```

Allocates a contiguous memory region of length n. Every block it allocates starts on an even address.

malloc, if successful, returns a pointer to the first memory location in the block. You should store this pointer for subsequent use in your program. malloc returns a NULL pointer value (0) and does not allocate any memory if allocating a contiguous block of the size requested would overlap: (1) a stored program, (2) the run-time stack, or (3) a previously allocated block in memory. (See K&R's description of alloc, pp. 97, 175).

MAX

```
int max(a, b)
     int a, b;
```

Returns the greater of a or b.

N

```
min(a, b)
     int a, b;
```

Returns the lesser of a or b.

:TEMP

```
mktemp(f)
     FILESPEC f;
```

Alters the FILESPEC string to a unique filename: the string that is returned can be used in an fopen, open, or creat without conflicting with an existing file. If mktemp cannot form a unique filename, then it returns NULL; if it can, it returns f, its argument. mktemp looks for a capital X and replaces it and the following character with a two digit decimal number.

MOVMEM

```
movmem(source, dest, n)

    char *source, *dest;
    int n;
```

Copies the contents of the n contiguous bytes in memory beginning at source into the n contiguous bytes beginning at dest. There is no restriction on the overlapping of these two regions. The bytes in the region pointed to by source are unaltered unless they are overwritten as a result of overlapping between the regions.

ICE

```
nice(n)
    int n;
```

Sets the priority of the current process. A null procedure under CP/M and MS-DOS.

OPEN

```
FILE *open(fspec, mode)

    FILESPEC fspec;
    unsigned mode;
```

Opens the file specified in fspec for direct I/O. This file must already exist. (See creat.) open, if successful, returns a pointer to a valid file descriptor for the file specified. You should store this pointer for subsequent use in your program. open returns ERROR (-1) and does not open the file if: (1) not enough user memory is available for a new file descriptor, (2) the file specification given is invalid, (3) the file specified either does not exist or was not created via creat, or (4) the file could not be opened due to an error at the operating system level.

For-mode you must specify one of the following: 0, 1, or 2. Which one you do specify determines the file's I/O mode, as indicated in the following table:

0	read-only mode
1	write-only mode
2	read-write mode

(See K&R, p. 162.)

OTELL

```
unsigned int otell(fd)

    FILE *fd;
```

Returns the byte offset from the beginning of the currently accessed 512-byte block of a file, at which the next file I/O operation on that file will begin. The file is specified by the pointer fd to its file descriptor. otell does not indicate within which 512-byte block the I/O operation will begin. See rtell and tell.

OUTP

```
outp(prt, b)

    int prt;
    int b;
```

Places b at the output port designated in prt and executes a byte OUT machine instruction for that port. (This function is available only on machines for which the byte OUT instruction makes sense.)

OUTP16

```
outp16(prt, w)

    int prt, w;
```

Places the word w at the output port designated in prt and executes a word OUT machine instruction for that port. (This function is available only on machines for which the word OUT instruction makes sense.)

PERROR

```
perror(s)
    char *s;
```

Prints the string s then a colon and then prints a string interpreting the value of errno, the I/O error value. If errno is zero, no interpretation is printed.

;ETC

```
int pgetc(fd)
    STREAM fd;
```

Identical to getc except that it replaces the system end of line indicator with a '\n'. Under CP/M and MS-DOS, this means that whenever it encounters a carriage return character ('\r') followed immediately by a newline character ('\n'), it returns only the newline. pgetc thus converts lines within files from CP/M to UNIX (ANSI) format.

POKE

```
poke(addr, b)

    char *addr, b;
```

Writes the byte b into the memory byte at addr. b must be an lvalue expression (see K&R, p. 183). The function poke, which has been included for compatibility with BDS C, is redundant in C, since indirection is a feature of the language.

PUTC

```
int pputc(c, fd)

    char c;
    STREAM fd;
```

Identical to putc except that it replaces the '\n' character with the system end of line character. Under CP/M and MS-DOS, this means that whenever it passes a newline character ('\n'), it first writes a carriage return character ('\r') to the file's I/O buffer and then writes the newline character that was passed. pputc thus converts lines written to files from UNIX (ANSI) to CP/M format.

PRINTF

```
printf(format,arg1,arg2,...)

    char *format;
    TYPE *arg1;
    TYPE *arg2;
      .
      .
      .
```

Writes a formatted output string to the standard output (the CP/M or MS-DOS CON: device--usually your console screen). printf must be passed the pointer, format, to a null-terminated string. (A string constant is also valid for format, since it evaluates to a null-terminated string.) This string controls the generation of the output string. printf may be passed a series of other arguments: arg1, arg2, ... The individual arguments in this series may be characters, integers, unsigned integers, or string pointers. Only the first argument, format, is required; all others are optional.

printf is a list function. See USING LIST FUNCTIONS at the end of Chapter 2.

The string pointed to by format may contain either ordinary characters or special substrings, beginning with the character %, that are called conversion specifications. Each ordinary character, when encountered by printf as it scans the string from right to left, is simply written to the standard output. Each conversion specification, when encountered, causes the value of the next argument in the series arg1, arg2, ... to be converted and formatted as specified and written to the standard output.

Following the character % in each conversion specification there may appear:

1) an optional minus sign, '-', which, if present, causes the converted value to be left-adjusted in its field. Right-adjustment is the default.

2) an optional string of decimal digits specifying the minimum number of characters in the field in which the value is to be written. The converted value will never be truncated. However, if it has fewer characters than are here specified, it will be padded on the left (or right, if left-adjustment has been specified) with spaces to the width specified. If this digit string begins with a zero, the converted value will be padded with zeros instead of spaces.

3) another optional string of decimal digits, which must be preceded by a period, '.', specifying the maximum number of characters to be copied from a null-terminated string.

4) a character, called the conversion character, indicating the type of conversion to be performed.

Of the above, only the conversion character must be present in a conversion specification. All the others, if present, must be in the order they are listed.

The (valid) conversion characters and the types of conversions they specify are:

c --the least significant byte of the argument is interpreted as a character. That character is written only if it is printable.

d --the argument, which should be an integer, is converted to decimal notation.

o --the argument, which should be an integer, is converted to octal notation.

x --the argument, which should be an integer, is converted to hexadecimal notation.

u --the argument, which should be an unsigned integer, is converted to decimal notation.

s --the argument is interpreted as a string pointer. Characters from the string pointed to are read and written until either a null character is read or an optionally specified maximum number of characters has been written. See item-3 above.

% --the character % is written. This is an escape sequence. No argument is involved.

(See K&R, pp. 7, 11, 145.)

PUTC

```
RESULT putc(c, fd)

    char c;
    STREAM fd;
```

Writes the character c to the I/O buffer associated with a file opened for buffered output via fopen, beginning at the current location in that buffer. Whenever that buffer becomes full, it is automatically flushed (that is, its entire contents are written to the file). The file is specified by the pointer fd to its file descriptor.

putc returns ERROR (-1) and does not write the character if: (1) fd does not point to a valid file descriptor, (2) the file was not opened for buffered output via fopen, or (3) the buffer could not be written due to an error at the operating system level. Otherwise, putc writes the character to the file's I/O buffer and returns SUCCESS (0). However, due to the buffering of file I/O operations, such a return value does not guarantee that that same character will be successfully written to the file, since errors resulting from and affecting the outcome of a particular call to putc may not become apparent until some later function call causes that file's I/O buffer to be flushed (see K&R, pp. 152, 166).

'UTCHAR

```
putchar(c)

    char c;
```

Writes the character c to the standard output (the CP/M or MS-DOS CON: device--usually your console screen). (See K&R, pp. 13, 144, 152).

PUTDEC

```
putdec(nn)
    int nn;
```

Prints the decimal number nn on the console. See printf for a more elaborate function.

TS

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

Writes the string beginning at s to the standard output (the CP/M or MS-DOS CON: device--usually your console screen). All carriage commands must appear explicitly in this string.

0,21

PUTW

```
RESULT putw(i, fd)

    int i;
    STREAM fd;
```

Writes the integer i to the I/O buffer associated with a file opened for buffered output via fopen, beginning at the current location in that buffer. Whenever that buffer becomes full, it is automatically flushed (that is, its entire contents are written to the file). The file is specified by the pointer fd to its file descriptor.

putw returns ERROR (-1) and does not write the integer if: (1) fd does not point to a valid file descriptor, (2) the file was not opened for buffered output via fopen, or (3) the buffer could not be written due to an error at the operating system level. Otherwise, putw writes the integer to the file's I/O buffer and returns SUCCESS (0). However, due to the buffering of file I/O operations, such a return value does not guarantee that that same integer will be successfully written to the file, since errors resulting from and affecting the outcome of a particular call to putw may not become apparent until some later function call causes that file's I/O buffer to be flushed. Calls to putc and putw may be interspersed. Files written with putc and putw may be read using getc and getw.

PUT2B

```
int put2b(i, fd)

    int i;
    STREAM fd;
```

Outputs a two byte quantity to a buffered output stream. put2b is similar to putw and fwrite, except that it is invariant with respect to byte ordering.

QSORT

```
int qsort(tbl, nrecs, reclen, cmp)

    char tbl[][];
    unsigned nregs, reclen;
    int (*cmp)();
```

qsort performs an ascending order sort on the double dimensioned array

```
    tbl[nrecs][reclen]
```

That is, tbl points to the base of the array; reclen is the length of the record that is to be sorted; nrecs is the number of records; cmp the function to be used to perform the comparison. It will be called with two pointers to records. Effectively, its declaration is:

```
    int
    cmp(a,b)
     char a[reclen];
     char b[reclen];
    {
```

It must return a value less than zero if the comparison is "less than"; a value of zero if the comparison is "same"; and it must return a value greater than zero otherwise.

```
int rand()
```

Returns the next value in a pseudo-random number sequence initialized by a prior call to srand. Values in the sequence will range from 0 to 65,535.

The C expression

```
    rand() % n
```

will evaluate to an integer greater than or equal to 0 but less than n.

READ

```
int read(fd, bufr, n)

    FILE *fd;
    TYPE *bufr;
    unsigned n;
```

Reads a maximum of n bytes from a file opened for either direct or buffered input, beginning at the current location of the file I/O pointer, into the memory buffer pointed to by bufr. The file is specified by the pointer fd to its file descriptor.

You should define the buffer pointed to by bufr such that it can contain at least n bytes.

The file I/O pointer will always point to the beginning of a system record. After a call to read, the file I/O pointer will point to the beginning of the system record following the last one read.

If no errors occur, read returns the actual number of bytes read. If those bytes are being read from a file, read returns either a multiple of the system record size or zero (Ø). Zero will be returned only if the end of the file has been reached. If bytes are being read from a serial device (such as the CP/M or MS-DOS CON: or RDR: devices) opened as a file, read returns one (1), since only one byte per call to read can be read from a serial device. read returns ERROR (-1) and does not attempt to read the file if: (1) the file was not opened for input, (2) n is less than the system record size, or (3) the file could not be read due to an error at the operating system level (see K&R, p. 16Ø).

REALLOC

```
TYPE *realloc(p, nbytes)

    TYPE *p;
    unsigned nbytes;
```

Changes the size of the allocated region pointed to by p (p must have been previously set by a call to malloc). realloc preserves the content of the region, as best as can be done, since the region may have a new size. realloc returns a pointer to the new size region.

RENAME

```
RESULT rename(fname, fspec)

    FILESPEC fname;
    FILESPEC fspec;
```

Renames the file specified in fspec, giving it the name contained in the null-terminated string pointed to by fname. (A string constant, such as "newname", is also valid for fname, since it evaluates to a pointer to a null-terminated string.) The drive name and the number, if any, are unchanged.

RESET

```
reset(n)

    int n;
```

Causes program execution to return to the point set by a prior call to setexit. This transfer has the appearance of a return from setexit. The parameter n passed to reset appears as the value returned by setexit.

reset and setexit together allow simpler and cleaner coding of repeated exits to a common point--particularly when such transfers require unraveling a number of levels of function calls. For example: in writing an interactive editor you could call setexit at the top of the command loop and test whether or not its apparent return value was equal to zero (0). Each non-zero value could be used to indicate a different error condition. The error number could be printed and command loop execution could continue. Calls to reset would be sprinkled in appropriate places throughout the loop. In each instance the parameter passed to reset would indicate the presence (non-zero) or absence (zero) of a particular error condition.

reset and setexit, while they resemble functions in usage and syntax, are implemented as compiler preprocesser directives rather than as functions. Thus, you will not find them in any of the standard library function files.

RINDEX

```
char *rindex(s, c)

    char *s, c;
```

Returns a pointer to the last occurrence of the character c in the string beginning at s. rindex returns a null pointer value (0) if c does not occur in the string. See index described earlier in this chapter.

RTELL

```
unsigned int rtell(fd)

    FILE *fd;
```

Returns the offset, in 512-byte blocks, from the beginning of a file of the 512-byte file block within which the next file I/O operation on that file will begin. rtell does not indicate the offset into that block at which the I/O operation will begin. The file is specified by the pointer fd to its file descriptor. See otell and tell.

BRK

```
char *sbrk(n)

    int n;
```

Adds n bytes to user memory (increments CCEDATA by n). sbrk, if successful, returns a pointer to the first byte in the block added. sbrk returns a value of ERROR (-1) and adds no bytes to user memory if a block of the size specified would: (1) overlap a stored program, (2) overlap the run-time stack, or (3) exceed the available memory.

IMPORTANT: Do not call sbrk with a negative argument between calls to alloc (see K&R, p.175).

SCANF

```
RESULT scanf(format, arg1, arg2,...)

    char *format;
    TYPE *arg1;
    TYPE *arg2;
      .
      .
      .
```

Reads a formatted input string from the standard input (the CP/M or MS-DOS CON: device--usually the console keyboard). Under control of the format string pointed to by its first argument, format, scanf extracts a series of substrings, known as input fields, from its input string, converts the values represented in each of these fields, known as input values, and assigns these converted values, in sequence, to the objects pointed to by its remaining arguments arg1, arg2, ...

scanf is a list function. See USING LIST FUNCTIONS at the end of Chapter 2.

As its first argument, scanf must be passed a pointer, format, to an appropriate null-terminated format string. (A string constant is also valid for format, since it evaluates to a pointer to a null-terminated string.) A series of other arguments, arg1, arg2, ..., may be passed to scanf, all of which must be pointers. The individual objects pointed to by arg1, arg2, ... may be either characters, character arrays, or integers.

The format string may contain either "whitespace" characters (that is, spaces, tabs, and newlines), ordinary characters, or special substrings, beginning with the character %, known as conversion specifications. The first conversion specification in the format string corresponds to and determines the boundaries of the first input field in the input string. It also determines the type of conversion to be performed on the input value represented in that field. Each successive pair of conversion specifications and input fields bears this same relationship.

Following the character % in each conversion specification there may appear:

1) an optional assignment suppression character, '*', which, if present, causes the corresponding input field to be skipped.

2) an optional string of decimal digits specifying the maximum number of characters in the corresponding input field.

3) a character, called the conversion character, indicating the type of conversion to be performed on the corresponding input value.

Of the above, only the conversion character must be present in a conversion specification. All the others, if present, must be in the order they are listed above.

The valid conversion characters and the types of conversions they specify are:

% --a single % character is expected in the input string at this point. This is an escape sequence--no assignment is performed.

c --the input value is interpreted as a character. The corresponding argument should be a character pointer. The normal skip over space characters is suppressed. To read the next non-space character, use %1s. If a field width is also specified, the corresponding argument should be a pointer to an array of characters, and the specified number of characters will be read.

s --the input value is interpreted as a character string. The corresponding argument should be a pointer to an array of characters large enough to hold the string in addition to a terminal null-character added by scanf. The input field is terminated either by a space or a newline or when the maximum number of characters has been read, whichever comes first.

[--the input value is interpreted as a character string. The corresponding argument should be a pointer to a character array large enough to hold the string plus a terminal null-character added by scanf. Where the input field is terminated is determined as follows. The left bracket above is followed by a set of characters and a right bracket. If the first character in that set is not a circumflex, '^', the input field is terminated by the first character not in the set within the brackets. If the first character is a circumflex, the input field is terminated by the first character in the set within the brackets (the '^' excluded).

d --the input value is interpreted as a decimal integer and is converted to a binary integer. The corresponding argument should be an integer pointer.

o --the input value is interpreted as an octal integer and converted to a binary integer. The corresponding argument should be an integer pointer.

x --the input value is interpreted as a hexadecimal integer and is converted to a binary integer. The corresponding argument should be an integer pointer.

The central task of scanf is to determine the boundaries of the input fields in its input string, which contain the input values to be converted and assigned. To find these substrings, scanf scans the characters in its input string, comparing each of them with the corresponding characters in the string pointed to by format. If a character in the input string matches the corresponding character in the format string, it is discarded and the next character in the input string is read. If the corresponding characters do not match, scanf returns immediately. Note that any amount of whitespace in the input string matches any amount of whitespace in the format string. Whitespace in the format string is optional (it is ignored), while, in the input string it can delimit input fields. Thus, corresponding characters are not simply those characters that are the same number of bytes from the beginning of their respective strings. Whenever the character %, which introduces a conversion specification, is encountered in the format string, the corresponding character in the input string is assumed to be the first byte of an input field. An input field extends either until a space character is encountered in the input string or the number of bytes specified for the field width has been read, whichever comes first. The conversion characters c and [above are the only exceptions to this otherwise general rule. Any inappropriate character in an input field causes scanf to return immediately.

scanf returns either the number of converted input values it assigned or, if no input is present at the standard input, the constant EOF (see K&R, p. 147).

SEEK

```
int seek(fd, offset, origin)

    FILE *fd;
    int offset;
    int origin;
```

Sets the value of the file I/O pointer associated with an open file. A file I/O pointer must be between 0 and 8 megabytes. The file is specified by the pointer fd to its file descriptor and may have been opened for either direct or buffered I/O. seek is primarily used in conjunction with tell and the direct file I/O functions read and write. Seek must be used with more care in conjunction with the buffered file I/O functions in order to prevent data loss.

The value assigned to offset has a different interpretation depending on the value assigned to origin:

If origin is 0, then the file I/O pointer will point to the beginning of the file plus offset bytes.

If origin is 1, then the file I/O pointer will point to its current position in the file plus offset bytes.

If origin is 2, then the file I/O pointer will point to the end of the file plus offset bytes.

If origin is 3, then the file I/O pointer will point to the beginning of the file plus offset times 512 bytes.

If origin is 4, then the file I/O pointer will point to its current position in the file plus offset times 512 bytes.

If origin is 5, then the file I/O pointer will point to the end of the file plus offset times 512 bytes.

(See K&R, p. 164.)

SETEXIT

```
int setexit()
```

Calling setexit sets its location as the "reset" point--the point to which subsequent calls to reset transfer program execution; 0 is returned. Each call to reset that follows causes an apparent return from the function setexit. setexit appears to return the value of the parameter, n, that was passed to reset. See reset and setjmp.

TJMP

```
setjmp(savearea)
    int savearea[savesize];
```

Calling setjmp stores the program state in the savearea and returns 0. (savearea should be 6 bytes long on both the 8080 and 8086 series.) The program state includes all register variables, the return program counter, and the stack pointer. Upon a call to longjmp (see earlier description) with the same savearea, the state is restored, effectively appearing as if a return from setjmp has occurred, with the return value being supplied by longjmp. setjmp is a generalized version of setexit.

SETMEM

```
setmem(p, n, b)

    TYPE *p;
    unsigned n;
    char b;
```

Sets the n contiguous bytes beginning at p to the value specified in b. You can use setmem to initialize a variety of buffers and arrays.

SLEEP

```
sleep(n)

    unsigned n;
```

Suspends execution for n tenths of a second on a Z80 CPU running at 4 Mhz. You can tailor this function to a different CPU and/or clock rate by changing the value of one or two constants located in the function's code.

SPRINTF

```
sprintf(s, format, arg1, arg2,...)

     char *s, *format;
     TYPE arg1;
     TYPE arg2;
       .
       .
       .
```

Identical to printf except that it writes its formatted output into the string beginning at s. Contrast this with printf, which writes its output to the standard output, and fprintf, which writes its output to a file. sprintf appends a null character to the formatted output string (see K&R, p. 150). sprintf is a list function.

AND

```
srand(seed)

     int seed;
```

Initializes the return value of rand to the value passed in seed.

SSCANF

```
sscanf(s, format, arg1, arg2,...)

     char *s, *format;
     TYPE *arg1;
     TYPE *arg2;
       .
       .
       .
```

Identical to scanf (and fscanf) except that its formatted input string is read from the null-terminated string beginning at s rather than from the standard input. sscanf does not read the terminal null character (see K&R, p. 150). sscanf is a list function.

STRCAT

```
char *strcat(s1, s2)

     char *s1, *s2;
```

Appends a copy of the string beginning at s2 to the end of the string beginning at s1, creating a single null-terminated string. Note that the resulting string begins at s1 and contains a single, terminal null character.

strcat returns a pointer to the resulting string identical to the parameter, s1, that it was passed (see K&R, p. 44).

STRCMP

```
int strcmp(s1, s2)

    char *s1, *s2;
```

Compares the string beginning at s1 with the string beginning at s2. This comparison is similar to an alphabetical comparison except that it is based on the numerical values of corresponding characters in the two strings. This comparison ends when the first null character in either string is encountered.

strcmp returns a positive integer, zero (0), or a negative integer depending on whether the string beginning at s1 is, respectively, greater than, equal to, or less than the string beginning at s2 (see K&R, p. 101).

STRCPY

```
strcpy(s1, s2)

    char *s1, *s2;
```

Copies the string beginning at s2 into the string beginning at s1, stopping after a null character has been copied. If the length of the string beginning at s2 is greater than the length of the string beginning at s1, data in the bytes following the latter may be overwritten in error (see K&R, p. 100).

STREQ

```
int *streq(s1, s2)

     char *s1, *s2;
```

Compares the characters in the strings beginning at s1 and s2, where n is the number of characters (excluding the terminal null) in the string beginning at s2. streq returns n if the corresponding characters in the two strings are identical; otherwise, it returns zero (0). Like substr (described later), except that it returns end of string instead of beginning.

TRLEN

```
int strlen(s)

     char *s;
```

Returns the number of characters (excluding the terminal null) in the string beginning at s (see K&R, pp. 36, 95, 98).

0,24

STRNCAT

```
char *strcat(s1, s2, n)

     char *s1, *s2;
     int n;
```

Identical to strcat except that strncat appends at most n characters from the string beginning at s2 (truncating from the right) to the end of the string beginning at s1.

STRNCMP

```
int strncmp(s, t, n)

     char *s;
     char *t;
     unsigned n;
```

Compares the strings pointed to by s and t. The comparison stops at first '\0' (like strcmp) or after n characters are scanned, whichever comes first.

STRNCPY

```
char *strncpy(s1, s2, n)

     char *s1, *s2;
     int n;
```

Identical to strcpy except that strncpy copies exactly n characters into the string beginning at s1, truncating or null-padding the string beginning at s2 if necessary. The resulting string may not be null-terminated if the string beginning at s2 contains n or more characters.

SUBSTR

```
char *substr(pa, pb)

     char *pa;
     char *pb;
```

Locates the beginning of the first occurrence of the substring (pointed to by pa) in the string pointed to by pb. Returns NULL if pa is not found in pb. See strcmp, streq, and index, also described in this chapter, for similar string functions.

SWAB

```
swab(s1, s2, n)

     char *s1, *s2;
     int n;
```

Copies n bytes from s1 to s2, swapping every pair of bytes.

LL

```
unsigned int tell(fd)

     FILE *fd;
```

Returns the byte offset from the beginning of a file at which the next I/O operation on that file will begin. The file is specified by the pointer fd to its file descriptor. If tell is called for a file greater than 64K long, its return value is subject to arithmetic overflow. See otell and rtell.

LOWER

```
char tolower(c)

     char c;
```

Returns the lower case equivalent of c if c is an upper case alphabetical ASCII character; otherwise, it returns c. (K&R, pp. 145, 156.)

TOPOFMEM

```
char *topofmem()
```

Returns CCEDATA (see brk, evnbrk, and sbrk).

TOUPPER

```
char toupper(c)
    char c;
```

Returns the upper case equivalent of c if c is a lower case alphabetical ASCII character; otherwise, it returns c (see K&R, p. 156).

UBRK

```
char *ubrk(u)
    unsigned u
```

Returns a pointer to a memory region of size u. Since u is unsigned, ubrk cannot "give back" any allocated memory. It returns ERROR if it cannot locate a free region of proper size. See sbrk, evnbrk, wrdbrk, and brk also described in this chapter.

UGETCHAR

```
ugetchar(c)
    char c;
```

Causes the next call to getchar to return c. Calling ugetchar more than once between successive calls to getchar will have no effect on the state of the standard input.

UNGETC

```
RESULT ungetc(c, fd)
    char c;
    STREAM fd;
```

Writes the character c into the most recently read byte of the I/O buffer associated with a file opened for buffered input via fopen. ungetc also decrements the pointer to the next byte to be read from the file I/O buffer so that it points to the byte that was just written.

ungetc, if successful, returns an undefined value. ungetc returns ERROR (-1) if it could not perform its function: if the file specified was not opened for buffered input via fopen, for example.

To call ungetc for a file serves no purpose unless either fgets, fscanf, getc, or getw (the buffered file input functions) has been previously called for the same file. Only one call to ungetc between calls to the buffered file input functions for a given file can be guaranteed to have the desired effect (see K&R, p. 156).

UNLINK

```
RESULT unlink(fspec)

    FILESPEC fspec;
```

Deletes the file specified in fspec from the file system. unlink returns SUCCESS (0) if the file was successfully deleted. unlink returns ERROR (-1) and does not delete the file if: (1) the file specification given is invalid or (2) the file could not be deleted due to an error at the operating system level (see K&R, p. 163).

WAIT

```
RESULT wait(pid)

    unsigned pid;
```

Blocks the execution of the process until the completion of the process with process id pid. Returns ERROR (-1) if no such process id is waiting. Always returns immediately with an ERROR value.

WRDBRK

```
wrdbrk(u)

    unsigned n;
```

Returns a pointer to a memory region of size u. Since u is unsigned, wrdbrk cannot "give back" any allocated memory. It returns ERROR(-1) if it cannot locate a free region of the proper size. See sbrk, evnbrk, ubrk, and brk also described in this chapter.

WRITE

```
int write(fd, buffer, num_bytes)

    FILE *fd;
    TYPE *buffer;
    int num_bytes;
```

Outputs the number of bytes specified in num_bytes from the area pointed to by buffer. Output is to a file opened for direct (unbuffered) output. The file is specified by a file descriptor, fd.

write returns the actual number of bytes written. This may be less than num_bytes. If the file descriptor is invalid or the file cannot be written, a value of ERROR (-1) is returned to indicate an error.

Every file descriptor contains a pointer to the next record to be accessed in file I/O operations. A call to write advances that pointer by the number of bytes written. A subsequent call to read or to write will begin at the new position of this pointer. By calling seek, you may alter the position of this file I/O pointer without reading or writing. (See K&R, p. 160).

XMAIN

xmain()

The first C function called upon program start up. It sets up the arguments to main and does I/O redirection if the switch REDIRECT is set in CUSTOMIZ.H before CRUNT2.C is recompiled. I/O redirection is the ability to redirect the console input, the console output, or command arguments to or from files. Console input is redirected by specifying a filename preceded by a '<' on the command line. Console input is then taken from the file. Console output is redirected by specifying a filename preceded by a '>' on the command line. Console output is then sent to that file. Command argument redirection is specified by preceding a filename with an '@' on the command line. Command arguments are then taken from the file.

Command line arguments are passed to the program by setting up two parameters in the call to main. The left parameter is the count of the number of arguments. The right parameter is an array of pointers to strings, one string for each argument.

If the command line and I/O redirection code is not desired, or if a different action is required, a program can be specified with its top level procedure being xmain rather than main.

XREV

```
TYPE *xrev(narg)

     TYPE nargs[]
```

Reworks the parameters in list functions. Under SuperSoft C, the count of pointers is assumed to be in nargs[0] and an array of parameters starts at &nargs[1]. xrev returns a pointer to the argument list. Turn to Chapter 2, USING LIST FUNCTIONS, to see xrev in use.

XPRINTF

```
xrprintf(line, args)

     char *line;
     TYPE args[];
```

Does all the work for printf et al. (See printf described earlier.) It expects a char array (line) to write its output string to an array of arguments (args). The first element of args should be a format string. (See printf described earlier).

0,24 XRSCANF

```
xrscanf(kind, u_kind, where, args)

     int (*kind)(), (*u_kind)(), where;
     TYPE args[];
```

Does all the work for scanf et al. (See scanf described earlier in this chapter). xrscanf expects to be passed two functions. kind(where) should return a byte from the input. u_kind(c, where) should push back a byte into the input. args is a format string (see scanf) followed by an array of addresses. xrscanf cannot be called recursively, because it makes use of a global static for some inter-function communication.

Appendix A

The Differences between SuperSoft C and Standard C

SuperSoft is committed to implementing the full C language. All purchasers of SuperSoft C will receive notice as updates are available. The standard C language features not yet implemented are: TYPEDEF declarations; declaration and use of bit fields; initializations. STATIC declarations are recognized; however, they have no initial value. Otherwise, EXTERN STATICs operate as expected and local STATICs do not have their values retained across function invocations.

LONG, FLOAT, DOUBLE and LONG FLOAT variables may be defined. These data types may be used in any declaration except as a function's formal argument. This includes declaration of STRUCTs, UNIONs, and arrays. The address operator is the only available operator on these data types. Note that there are library functions (DOUBLE.C and LONG.C) that operate on these data types.

SuperSoft C expects parameters to functions to be pushed onto the stack in right to left order, followed by the return address (presumably by the call instruction of the machine). Either right to left order or left to right order are allowed under the language as defined in K & R, however, a number of compilers push arguments in the reverse order to SuperSoft C. In particular, the original DEC PDP-11 implementation pushes left to right.

All the formal arguments to a function must be declared within that function. That is:

```
func(aa,bb,cc)
 int bb;
 int aa, cc;
{

...is accepted, but

func(aa,bb,cc)
{

...will generate an error.
```

The code generator does not yet attach any unique prefixes or suffixes to variable names. Thus global identifiers in your C source code may conflict with the assembler's names and keywords. You must therefore avoid the use in your C source code of any of your assembler's reserved symbols or keywords. This deficiency will be remedied in a future release.

Parameterized #DEFINEs and #IF (with expression) preprocessor directives are not yet supported. Unlike standard C, lines are only scanned once for #DEFINE macro substitutions. This means that there is no way to get into indefinite looping during preprocessing. However, this also means that the order of #DEFINEs is significant. For instance, in--

```
#define x       y
#define y       z
```

--x will be replaced with y and y will be replaced with z. However, in--

```
#define y       z
#define x       y
```

--both y and x will be replaced with z.

In SuperSoft C, there must be no intervening newline between a label and its associated ':'. Array declarations can contain only limited expressions. In particular, SIZEOF and parenthesized expressions are not allowed. Because of the way SuperSoft C parses local declarations, a variable declared to be REGISTER will not truly refer to a register unless it is the first REGISTER declaration and it is declared by itself. So the following will declare x as a true register variable, but y will be declared as an ordinary local:

```
funct()
{
        register char x;
        register char y;
```

The following will not succeed in declaring any true register variable:

```
funct()
{
        register char x, y;
```

Dup and fork as available under UNIX are not implemented, mostly due to the inconvenience of single process operating systems. Link operates differently than UNIX's link. Fopen and fdopen have an additional parameter over the UNIX implementation, allowing for clearer buffered I/O usage. This poses no problems if the additional parameter is used in most UNIX C implementations, since the final, additional parameter is ignored.

Release 1.2 of SuperSoft C has fuller UNIX compatibility than the previous release (1.1). There are no longer any record size limitations on file I/O. This has affected read() and write(). Also there has been a mode parameter added to creat(), and an arg0 parameter added to execl() to bring SuperSoft C into line with UNIX. This release of C also contains a much fuller set of preprocessor directives.

Appendix B

Currently Available Machine and Operating System Configurations of the SuperSoft C Compiler

Current operating systems are: CP/M-80, MP/M-80, CP/M-86, Concurrent CP/M-86, CP/M+, CP/M-80 3.0, MP/M-86 (and compatible); PC-DOS, MS-DOS (and compatible); UNIX, XENIX (and compatible); and Central Data ZMOS.

Current host or target CPUs are: Intel 8080, Intel 8085, Intel 8086, Intel 8088, Intel 186, Intel 188, Zilog Z80, Zilog Z8001, Zilog Z8002, Zilog Z8003, and Zilog Z8004.

SuperSoft supplies any valid combination of host, target, and operating system. Those interested in such systems should contact SuperSoft for information about availability.

Appendix C

Some Common Problems and Solutions

1. Problem: During assembly or link, a function is undefined.
   Solution: Remember to include the files containing the needed functions. See Chapter 4.

2. Problem: During assembly or link, some variables are undefined.
   Solution: Remember to declare all variables.

3. Problem: During assembly or link, a function or variable is flagged with duplicate definition errors.
   Solution: Have you accidentally defined a function or variable more than once? Have you used a name that the C support uses? How many significant letters (truncation limit) does your assembler use?

4. Problem: During assembly a "P", or phase, error is indicated.
   Solution: A variable or function has been defined more than once.

5. Problem: During assembly a symbol is indicated as being public and external at the same time.
   Solution: This variable or function is misspelled and the assembler is not catching it because the difference is an upper-lower case change or a difference beyond the truncation limit of the assembler. For example, TPuts and Tputs may be the same to the assembler. verylongname and verylongname1 may be the same, also.

6. Problem: During linking a symbol that should be in a library is not found.
   Solution: Scan the library twice during linking. If the symbol is then found, you need to reorder the library. In the interim, double scanning should be adequate. If the symbol is still not found, then the symbol is not in the library.

7. Problem: alloc0 is undefined during link.
   Solution: You are calling functions without their "matching" precursor. For instance, you are calling fclose without fopen or free without malloc.

8. Problem: Execution of a C program cannot be terminated with Control-C under CP/M.
   Solution: This is because CP/M is usually not interrupt driven and cannot respond to characters from the keyboard in an arbitrary circumstance. However, during output to the console, typing Control-S and then Control-C will work.

9. Problem: The linker dies, halts your system, or says that it is out of space.
   Solution: Find more memory or switch linkers. Under CP/M-80, we suggest SuperSoft's ELINK, a disk-based linker that should not run out of space.

Appendix D

Locations of Supplied Functions and Externals

--ALLOC.C

alloc	free	malloc	calloc
realloc	isheap		

-C2.RH, C2.RT, C2I86.RH, C2I86.RT, MDEP.C, C2I86.RTB, C2I86.RTM, C2PRE.ASM, C2POST.ASM, C2.RTM

bdos	bios	brk	ccall
ccalla	exit	inp	inp16
longjmp	outp	outp16	reset
setexit	setjmp	streq	errno
comlen	comline	ccexit	

--CRUNT2.C

evnbrk	getchar	gets	isalpha
isdigit	islower	isupper	iswhite
movmem	putchar	puts	sbrk
setmem	strlen	toupper	ubrk
assert	ugetchar	wrdbrk	xmain

--FUNC.C

abs	absval	atoi	perror
getval	index	initb	initw
isalnum	isascii	iscntrl	isnumeric
isprint	ispunct	isspace	kbhit
min	max	pause	peek
poke	putdec	qsort	rand
rindex	sleep	srand	strcat
strcmp	strcpy	strncat	strncmp
strncpy	substr	tolower	topofmem

--STDIO.C

close	cpmver	creat	exec
execl	fabort	fclose	fflush
fgets	fopen	fputs	fread
fwrite	getc	getw	access
link	open	otell	pgetc
pputc	putc	putw	read
rename	rtell	seek	tell
ungetc	unlink	write	wait
lock	nice	swab	isatty
mktemp	clearerr	freopen	fdopen
isfd	ferror	fileno	fgetc
fputc	get2b	put2b	chmod

FORMATIO.C

fprintf	fscanf	printf	scanf
sprintf	sscanf	xrprintf	xrscanf
xrev			

DOUBLE.C (Double Floating Point. See Appendix G.)

badd	bsub	bdiv	bmul
bmodulo	bxtofy	s2bcd	bcd2s
bneg	buneg	babs	buabs
bint	bentier	blog	bsqr
bexp	bfac	bsin	bcos
btan	barctan	bsign	blt
bgt	beq	bne	ble
bge	btest	bseterrflg	bmant
bexpo	bmov	bround	int2bcd
bcd2int	bcheby		

LONG.C (Long Integer Functions. See Appendix H.)

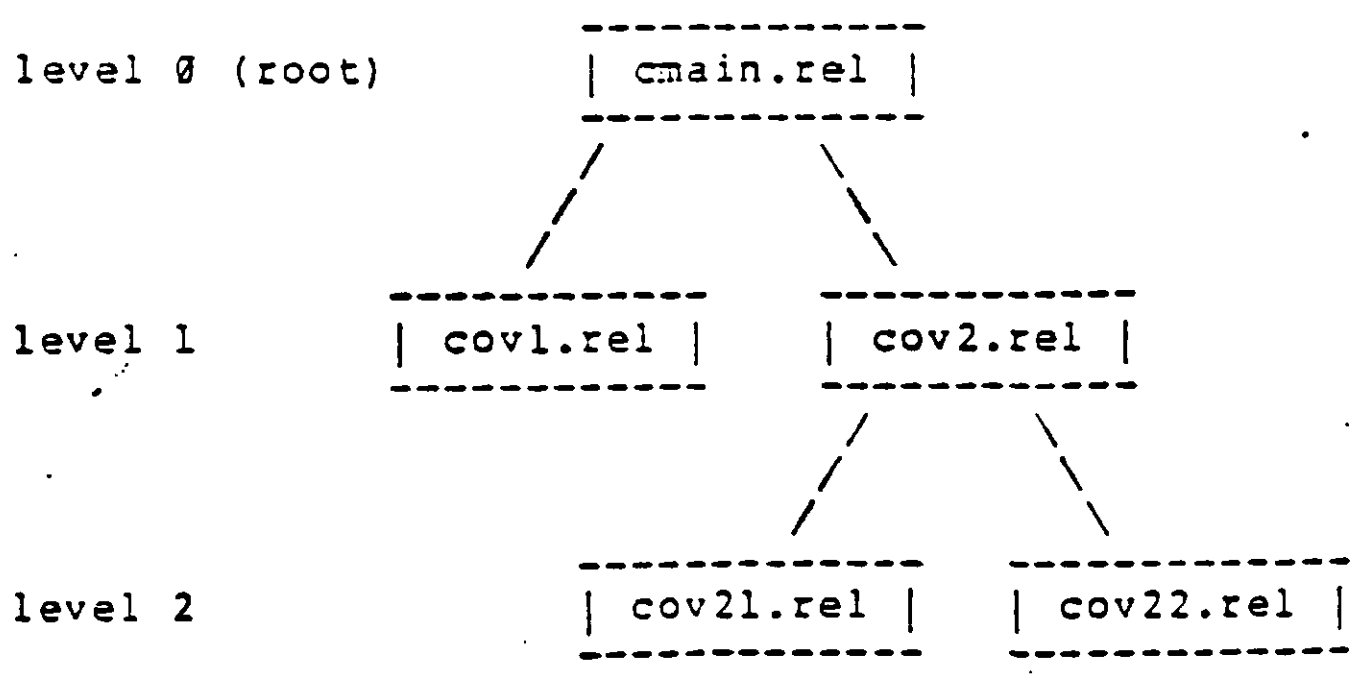
cc1prt	cc1neg	cc1inc	cc1dec
cc1com	cc2mov	cc2add	cc2com
cc2or	cc2xor	cc2cmp	cc2mul
cc2imul	cc2iadd	cc2and	cc2sub
cc2or	cc2neg	cc3add	cc3and
cc3or	cc3xor	cc3sub	cc3div

Appendix E

Using Overlays under CP/M-80

SuperSoft's ELINK (which is available as an additional cost option to the C Compiler) allows the creation of overlayed programs from existing C and assembler relocatable modules. ELINK is a disk based, stand-alone, multiple segment (overlay) linkage editor and loader. ELINK produces executable segments which run under CP/M-80 and is able to process relocatable files produced by various assemblers including M80 (Microsoft) and RMAC (Digital Research). Relocatable files may be gathered to libraries and searched. Large programs (up to 64k) may be linked, even under CP/M systems with less than 64k memory. Even larger programs may be divided into segments and overlayed. Overlayed programs obviate many of the storage related limitations in your programs and do so without chaining or changing your code.

Creating overlays is a fairly automatic process. The C source code does not have to be altered at all to take into account the differences between a call to a function in the same overlay and a call to a function in another overlay. The structure of the overlay is defined exclusively by commands to ELINK, which separate modules into overlays and a root executable .COM file. An overlay structure might look like this (following the example in the ELINK documentation):



An overlay structure is a tree structure in which levels are important. Each overlay (represented by a box in the above diagram) can contain any number of local functions that will be accessible to itself and its children. Each function in an overlay can call any number of functions in other overlays. You can have up to 85 overlays with up to 255 levels deep.

Overlays have the added benefit of requiring discipline in structuring the modularity of a program. The tree structure imposed by overlays is fortunately ideal for supporting a number of types of systems, such as menu driven programs. The overlays can mirror exactly the structure of the program.

By putting the C libraries in the root, significant savings can be made because they occur in only one place and do not have to be loaded with each overlay.

ELINK does NO CALL PATH CHECKING and assumes that the parent in the overlay tree is resident with the child. This means simply that each overlay structure can only call routines in the entire path to the root up the tree. You must avoid having a function make an AUTO load call to a function in another overlay at the same level in the structure. This would cause the return to go to a non-resident function because the function that was called overlayed the caller.

ELINK contains a modified C2.RH and C2.RT which use various features of ELINK and are required for overlaying. The changes are very simple and deal only with the locating of the top of the user program. (ALLOC and several other functions will not work properly without these modified files.)

To create and execute an overlayed program perform the following steps.

1. Compile all of the .C files with SuperSoft C and assemble with M80. (RMAC may also work, but we have no verification of this.)

2. With all the ".REL" files on the current disk, type:

```
ELINK so(cex);
```

   This will cause ELINK to read CEX.LNK for commands that will create the overlay structure defined above. (For furthur reference, see the ELINK documentation).

CMAIN.COM and CMAIN.OVL are created. The .COM file should contain the root and C libraries along with the ELINK run time package. The source of this run time is included with ELINK in case you wish to customize your environment. CMAIN.OVL contains all of the overlay code in absolute form along with a directory to the overlays in the file. CP/M 2.2 is required for single file overlays and AUTO load entry points. Direct overlay of files is supported under CP/M 1.4. See the ELINK manual for more details.

Simply typing CMAIN will execute the program. The overlays will be called as needed.

Appendix F

Error Messages

"A formal arg is not declared" (CC)

The function declaration that precedes this error has a formal argument without a corresponding type declaration. For instance:

```
func(a,b)
 char *a;
{
}
```

will generate this error.

"Already defined" (CC)

This is a function, formal argument, local (auto), or structure element that was previously defined.

"Array or pointer being lengthened" (CC)

This declaration increases the stated size of the array being referenced. In the case of a pointer this means that the object that it points to will appear bigger than before, causing any pointer arithmetic to multiply or divide by the new size. This is not always in error, as it is common to declare externs with arbitrary size:

```
extern char arr[];
```

and then elsewhere declare a specific size:

```
char arr[99];
```

"Bad break" (CC)

A break was encountered outside of all loops (for, while, do, or switch).

"Bad register op" (C2 or COD2COD)

U-code error. A register is used on an opcode that does not have a register mode.

"Bad register struct or union" (CC)

A register variable may be a struct or a union, but its size currently must be no larger than an int.

Bad register type" (CC)

Too complex a declaration is being applied to a register. For instance:

```
register int arr[99];
```

is not possible.

Bad usage" (CC)

This usage of a variable is inconsistent with its declaration.

Can't open include file" (CC)

A #include preprocessor directive has been encountered but the specified file does not exist.

"Can't subscript" (CC)

Attempt to subscript any expression (including a variable) that is not an array or a pointer.

"Can't take address" (CC)

Attempt to take the address of an expression.

"Can't write to output file" (CC, C2, or COD2COD)

Attempt to write to the output file, but an I/O error has occurred. Usually indicates that the output file system is full. See the -ofile option for a remedy.

"Cast" (CC)

An attempt to use casts. May not work.

'Continue without matching loop" (CC)

A continue has been encountered outside of an enclosing for, while, or do statement.

"Declaration mismatch" (CC)

A redeclaration of a variable is different from a previous declaration. This message will not appear if a variable is redeclared identically to previous declarations.

"Divide by Ø" (C2 or COD2COD)

During optimization (constant folding) a divide by zero was detected. Your algorithm should not require a divide by zero.

"Don't add pointers" (CC)

An attempt to add a pointer (or array) to a pointer (or array). This is not an appropriate use of pointer arithmetic, which expects one side of the binary operator to be a pure value.

"Don't negate pointers" (CC)

A subexpression of the form int-pointer was found. Perhaps you meant pointer-int.

"Duplicate default" (CC)

More than one default "label" has appeared in the context of the current switch. Nested switches can each have one default.

pecting formal arg" (CC)

This error appears when, inside of the argument list declaration of a function, an identifier is expected. That is, a '(', or a ',' has been seen.

xpecting function body" (CC)

All of the formal arguments for this function declaration have been declared. The first '{' of the function body is expected at this point.

xpecting ',' or ')'" (CC)

Inside of a function formal argument list, neither a ',' or a ')' was found when expected.

xpecting ',' or ';'" (CC)

A declaration of the type of a formal argument was not followed by a ',' or a ';'. Sometimes indicates confusion on the part of the compiler with regard to your intentions in declarations. It appears that the C syntax leaves little opportunity for error recovery in declarations.

"Expecting ')' or ',' in function call" (CC)

Refers to a badly formed function call.

"Expecting type declaration" (CC)

A struct element declaration is expected. This should start with a base type declaration such as int or char.

"File close error" (CC, C2, or COD2COD)

An output error has occurred while writing the last few bytes to the output file or during the actual close. Usually indicates that there is just short of enough space on the output file system. See the -ofile option for a remedy.

"gen err t1" (C2I86)

An invalid U-code has been seen. Either an invalid U-code was read by C2I86 or there is a memory problem internal to C2I86.

"gen err t2" (C2I86)

An invalid U-code has been seen. Either an invalid U-code was read by C2I86 or there is a memory problem internal to C2I86.

"gen err t3" (C2I86)

An invalid U-code has been seen. Either an invalid U-code was read by C2I86 or there is a memory problem internal to C2I86.

"gen err t6" (C2I86)

An invalid U-code has been seen. Either an invalid U-code was read by C2I86 or there is a memory problem internal to C2I86.

"gen err t7" (C2I86)

An invalid U-code has been seen. Either an invalid U-code was read by C2I86 or there is a memory problem internal to C2I86.

"Internal error: close on bad fd" (CC)

A close on an include file that should have succeeded didn't. This should never happen and usually means that the compiler's stack has been overwritten. Add more memory to your system or decrease the number of declarations in this compile (for instance by splitting the program into more files, each separately compiled). Also see the -bufsiz option for decreasing the compiler's I/O buffer sizes.

nternal error: extra free" (CC)

An expression left a sub-expression dangling. Should never happen. Perhaps means that the compiler's heap has been overwritten. Add more memory to your system or decrease the number of declarations in this compile (for instance by splitting the program into more files, each separately compiled). Also see the -bufsiz option for decreasing the compiler's I/O buffer sizes.

:nternal error: missing free" (CC)

An expression contained a spurious sub-expression. Should never happen. Perhaps means that the compiler's heap has been overwritten. Add more memory to your system, or decrease the number of declarations in this compile (for instance by splitting the program into more files, each separately compiled). Also see the -bufsiz option for decreasing the compiler's I/O buffer sizes.

"Internal error: optdel" (C2 or COD2COD)

An attempt was made to delete a U-code from an invalid location in the internal memory of the optimizer. This should never happen. May mean a memory overrun has happened. Reduce the size of the largest function or use the -o option. Also see the -bufsiz option for decreasing the compiler's I/O buffer sizes.

"Internal warning: basic block" (C2 or COD2COD)

Ignore this warning: indicates a minor inconsistency in the optimizer basic block processing.

"inval pseudo op type" (C2I86)

An invalid U-code has been seen. Either an invalid U-code was read by C2I86 or there is a memory problem internal to C2I86.

"Line too long" (CC)

A source line is too long or a line becomes too long after preprocessing. What the compiler "sees" is printed with the +l option.

Missing bracketing symbol" (CC)

A bracketing symbol, usually ']', ')', ':', or '}', is missing. The exact symbol is printed in the output file.

Missing '}'" (CC)

End of file was seen before the last '}' was seen. This is an insidious error, as it can involve an extra '{' or a missing '}' that is other than the last '}'. Check the proper closure of all previous compound statements.

Missing ':'" (CC)

A ':' is missing while trying to parse the ternary if operator, '?'.

"Missing label" (CC)

A label was missing from a goto statement.

"Missing quote or apostrophe" (CC)

A string is being read that has no terminating quote or a character literal is being read that has no terminating apostrophe.

"Missing ';'" (CC)

A semicolon is expected at the end of a statement or declaration.

"Missing while" (CC)

Missing while at end of a do statement.

Must be a constant" (CC)

Expecting a constant in an array declaration. Currently sizeof is not allowed in array declarations. Parenthesized expressions are not currently allowed. Variables never will be allowed.

Must be lvalue" (CC)

A valid left-hand-side is expected at this point. A left-hand-side must have an address and must be able to hold a variable value. Note that an array name may not be assigned to, and thus is not a good, left-hand-side.

'Nonsensical pointer operation" (CC)

The pointer operation you are performing would not normally be considered valid or sensible, but it will be performed anyway.

"No symbol table room" (CC)

The compiler's heap is filled up or has been overwritten. Add more memory to your system or decrease the number of declarations in this compile (for instance by splitting the

program into more files, each separately compiled). Also see the -bufsiz option for decreasing the compiler's I/O buffer sizes.

"Not a label" (CC)

Attempt to perform a goto to something that is not a label.

"Not an array or pointer" (CC)

Attempt to subscript a variable that is not an array or pointer.

"Not a pointer or array" (CC)

An attempt to use the indirection operator, '*', on something other than a pointer or array.

Not a pointer to a function" (CC)

A call to a function is being attempted, but the called function has been previously declared as something other than a function. For instance, the following will cause this message:

```
int fn;

fn();
```

Not a struct or union element" (CC)

An attempt to use a '.' operator or a '->' operator, but the right hand side of the operation is not a struct or union element.

"Not declared" (CC)

An attempt to use an undeclared variable. This error message will appear only on the first such occurrence of the variable.

"Opcode error" (C2 or COD2COD)

Either an invalid U-code was input to C2 (or COD2COD) or the optimizer generated a bad opcode: most likely an internal error.

"Open failure" (CC, C2I86, COD2COD, or C2)

Can't open the output file. Such an error could be caused if the file system does not exist, if it is not correctly set-up (i.e. no disk in drive), if there is no room on the disk, or if the filename is incorrectly formed.

"Operation on incompatible pointers" (CC)

Two pointers are used in an arithmetic operation, but they point to different objects.

"Optimizer table overflow" (C2 or COD2COD)

An attempt was made to add a U-code to the internal memory of the optimizer but there was no room. This indicates that the optimizer is out of memory. Reduce the size of the largest function or use the -o option. Also see the -bufsiz option for decreasing the compiler's I/O buffer sizes.

"Optout empty" (C2 or COD2COD)

Attempt to emit a U-code from the internal memory of the optimizer, but none was found. This should never happen and even if it does, it should have no effect on the correctness of the code that is generated.

"Out of heap" (CC)

This usually means that the compiler's heap is out of room or has been overwritten. Add more memory to your system, or decrease the number of declarations in this compile (for instance by splitting the program into more files, each separately compiled). Also see the -bufsiz option for decreasing the compiler's I/O buffer sizes.

"psu err t4" (C2I86)

An invalid U-code has been seen. Either an invalid U-code was read by C2I86 or there is a memory problem internal to C2I86.

"psu err t5" (C2I86)

An invalid U-code has been seen. Either an invalid U-code was read by C2I86 or there is a memory problem internal to C2I86.

edeclaration of a label" (CC)

A label has been declared more than once inside of a single function. Labels are declared just by following them by ':'. You probably shouldn't be using labels anyway.

Redeclaration of struct type" (CC)

A struct type has been declared more than once:

```
struct x {
        struct x {
                int i;
                char j;
        }
}
```

In the above, x has been redeclared.

0,24

Redefined" (CC)

A #defined identifier has been redeclared. This message will only appear if the new declaration is a different string than the old.

"Registers have no address" (CC)

An attempt has been made to take the address ('&' operator) of a register variable. This error message will appear if and only if the register variable is a true register variable.

"shli error" (C2 or COD2COD)

An invalid U-code has been seen. Either an invalid U-code was read by C2I86 or there is a memory problem internal to C2I86.

"String size exceeded" (CC)

A string is larger than the compiler can handle. The string may be unterminated. Otherwise, try using strcat at run time.

"Too many '&'s" (CC)

Attempt to take the address, using the '&' operator, of an address.

"Undefined struct" (CC)

The keyword struct has been followed by an identifier that has never been defined.

"Unrecognized '#'" (CC)

A # is followed by a directive that is not recognized. Some valid directives are if, ifdef, ifndef, else, endif, include, and define.

"Variable or constant expected" (CC)

A variable or a constant was expected at this point in the program. Most likely a badly formed expression has been used.

Appendix G

Double Floating Point Functions

The functions in DOUBLE.C can be divided into four groups:

Group 1 contains the three operand arithmetic functions of addition, subtraction, multiplication, division and modulus (remainder). These functions return the computed value in their first argument. The return value is the address of this first argument.

Group 2 functions are for the most part two argument functions which return the computed value in the second argument. These commands are double to string, string to double, integer truncation, sine, cosine, tangent, square root, exponentiation, natural logarithm, negation, absolute value, and factorial. Radians are used for functions requiring or returning angle measurement.

Group 3 contains, for the most part, two operand conditional and testing functions. These functions return TRUE (1) if the given condition is met between the arguments; otherwise, FALSE (Ø). Group 3 commands are equal, not equal, greater than, greater than or equal to, less than, and less than or equal to. Also available is a test function which returns POSITIVE (1), ZERO (Ø), or NEGATIVE (-1) depending upon the argument. These functions do not set Berrflag.

Group 4 contains miscellaneous functions.

---Group 1---

```
double *
Badd(dest, arg1, arg2) double *dest, *arg1, *arg2;
```

Three operand addition:

```
*dest = *arg1 - *arg2;
```

```
double *
Bsub(dest, arg1, arg2) double *dest, *arg1, *arg2;
```

Three operand subtraction:

```
*dest = *arg1 + *arg2;
```

```
double *
Bdiv(dest, arg1, arg2) double *dest, *arg1, *arg2;
```

Three operand division:

```
*dest = *arg1 / *arg2;
```

```
double *
Bmul(dest, arg1, arg2) double *dest, *arg1, *arg2;
```

Three operand multiplication:

```
*dest = *arg1 * *arg2;
```

```
double *
Bmodulo(dest, arg1, arg2) double *dest, *arg1, *arg2;
```

Three operand modulus:

```
*dest = *arg1 % *arg2;
```

Bmodulo assumes *arg2 is positive.

```
double *
Bxtofy(dest, arg1, arg2) double *dest, *arg1, *arg2;
```

Three operand exponentiation:

$$*dest = *arg1^{*arg2}$$

---Group 2---

```
double *
s2Bcd(ddest, strsource) double *ddest; char *strsource;
```

string to double:
ddest is the address of a double variable in which is placed the value of number in the string strsource.

```
char *
Bcd2s(strdest, dsource) char *strdest; double *dsource;
```

double to string:
This function is the inverse of s2bcd, where strdest must be long enough to fit the representation of the given dsource number. The maximum size for a double number is currently twenty-one bytes. This includes the exponent and the two signs, plus an extra byte for null termination.

```
double *
Bneg(dest, arg1) double *dest, *arg1;
```

Two operand negation:

*dest = -*arg1

```
double *
BUneg(dest) double *dest;
```

One operand negation:

*dest = -*dest

```
double *
Babs(dest, arg1) double *dest, *arg1;
```

Two operand absolute value:

*dest = |*arg1|

```
double *
BUabs(dest) double *dest;
```

One operand absolute value:

```
*dest = |*dest|
```

```
Bint(dest, arg1) double *dest, *arg1;
```

Returns the integer part of *arg1 in *dest (towards 0).

```
Bentier(dest, arg1) double *dest, *arg1;
```

Returns the floor (entier) of *arg1 into *dest (towards -infinity).

```
double *
Blog(dest, arg1) double *dest, *arg1;
```

Returns the natural log of *arg1 in *dest.

```
*dest = log(*arg1);
```

```
double *
Bsqr(dest, arg1) double *dest, *arg1;
```

Return the square root of *arg1 in *dest.

```
*dest = sqrt(*arg1)
```

*arg1 is tested against being negative.

```
double *
Bexp(dest, arg1) double *dest, *arg1;
```

Raises e (e == 2.7...) to the *arg1 power:

$$*dest = e^{*arg1}$$

*arg1 is tested against being too large.

```
double *
Bfac(dest, arg1) double *dest, *arg1;
```

Takes the factorial of *arg1:

```
*dest = (*arg1)!;
```

*arg1 is tested against being negative or too large.

transcendental functions:

```
double *
Bsin(dest, arg1) double *dest, *arg1;
```

Returns the sine of *arg1 in *dest:

```
*dest = sin(*arg1);
```

```
double *
Bcos(dest, arg1) double *dest, *arg1;
```

Returns the cosine of *arg1 in *dest:

```
*dest = cos(*arg1);
```

```
double *
Btan(dest, arg1) double *dest, *arg1;
```

Returns the tangent of *arg1 in *dest:

```
*dest = tan(*arg1);
```

```
double *
Barctan(dest, arg1) double *dest, *arg1;
```

Sets *dest to the arc tangent of *arg1:

```
*dest = arctan(*arg1);
```

---Group 3---

```
Bsign(x)
 double *x;

        Returns  1 if *x is positive;
                 0 if *x is zero;
                -1 if *x otherwise.
```

```
Blt(x, y) double *x, *y;

        Returns 1 iff *x < *y
```

```
Bgt(x, y) double *x, *y;

        Returns 1 iff *x > *y
```

```
Beq(x, y) double *x, *y;

        Returns 1 iff *x == *y
```

```
Bne(x, y) double *x, *y;

        Returns 1 iff *x != *y
```

```
Ble(x, y) double *x, *y;

        Returns 1 iff *x <= *y
```

```
Bge(x, y) double *x, *y;

        Returns 1 iff *x >= *y
```

```
Btest(x, y) double *x, *y

        Returns  1 if *x > *y;
                 0 if *x == *y;
                -1 if otherwise.
```

Bseterrflg(e)

Sets the global Berrflg to e if it is clear.

---Group 4---

```
double *
Bmant(dest, n) double *n; char *dest;
```

Places the mantissa of *n into *dest. The functions Bmant and Bexpo return values a and b respectively where the argument x is of the form x = a * 10**b for .1 <= |a| < 1. (except when x == 0).

```
Bexpo(x) double *x;
```

Returns the integer exponent of *x.

The functions Bmant and Bexpo return values a and b respectively where the argument x is of the form x = a * 10**b for .1 <= |a| < 1. (except when x == 0).

```
Bmov(dest, arg1) double *dest, *arg1;
```

Copies *arg1 into *dest:

```
*dest = *arg1;
```

```
Bround(dest, i) double *dest; int i;
```

Rounds *dest by adding 5.0Ei to the *dest (mantissa 5 with exponent i).

```
int2bcd(dest, i) double *dest; int i;
```

Converts from integer to double.

```
Bcd2int(arg1) double *arg1;
```

Returns the integer part of *arg1, rounded away from zero.

```
double *
Bcheby(res, x, coef, n) double *res, *x, coef[n][BCDS]; int
```

Returns in res the nth approximation of the function whose Chebyshev coefficients are in coef evaluated at *x.

Appendix H

Long Integer Functions

The following functions are available in the file LONG.C.

cclprt(a) long *a;

Prints *a in hexadecimal.

cclneg(d) long *d;

One operand negate:

*d = -*d;

cclinc(a) long *a;

One operand increment:

++*a

ccldec(a) long *a;

One operand decrement:

--*a

cclcom(a) long *a;

One operand complement:

*a = ~*a

```
cc2mov(a,b) long *a, *b;
```

Two operand copy:

```
*a = *b
```

```
cc2add(a,b) long *a, *b;
```

Two operand addition:

```
*a += *b
```

```
cc2com(a,b) long *a, *b;
```

Two operand complement:

```
*a = ~*b
```

```
cc2or(a,b) long *a, *b;
```

Two operand logical or:

```
*a |= *b
```

```
cc2xor(a,b) long *a, *b;
```

Two operand xor:

```
*a ^= *b
```

```
cc2cmp(a,b) long *a, *b;
```

Two operand comparison:

```
Returns  1 if *x > *y;
         0 if *x == *y;
        -1 if otherwise.
```

cc2mul(pa,pb) long *pa, *pb;

Two operand multiplication:

*pa *= *pb;

cc2imul(x,b0,b1) long *x; int b0, b1;

Two operand immediate multiplication where b0, b1 are integers that form the most significant and least significant part (respectively) of the immediate long:

*x *= (b0,b1)

cc2iadd(x,b0,b1) long *x; int b0, b1;

Two operand immediate addition where b0, b1 are integers that form the most significant and least significant part (respectively) of the immediate long:

*x += (b0,b1)

cc2and(a,b) long *a, *b;

Two operand logical and:

*a &= *b

cc2neg(a,b) long *a, *b;

Two operand negation:

*a = -*b

cc2sub(a,b) long *a, *b;

Two operand subtraction:

*a -= b

cc3add(q,a,b) long *q, *a, *b;

Three operand addition:

```
*q = *a + *b
```

cc3and(a,b,c) long *a, *b, *c;

Three operand logical and:

```
*a = *b & *c
```

cc3or(a,b,c) long *a, *b, *c;

Three operand logical or:

```
*a = *b | *c
```

cc3xor(a,b,c) long *a, *b, *c;

Three operand exclusive or:

```
*a = *b ^ *c
```

cc3sub(a,b,c) long *a, *b, *c;

Three operand subtraction:

```
*a = *b - *c
```

cc3div(d,s,ccxdrem) long *d, *s, *ccxdrem;

A form of three operand signed division with remainder:

```
*d /= *s
*ccxdrem = *d % *s
```