BDSC

USER'S GUIDE

BDSoftware, Inc.

C Compiler v1.5

BD Software, Inc.
P.O. Box 2368
Cambridge, MA 02238
(617) 576-3828

Contents

Chapter 1 Introduction 1
- 1.1 Hello There . . . 1
- 1.2 Quick Start . . . 1
- 1.3 Support . . . 3
- 1.4 No Royalties, Of course! . . . 4
- 1.5 Objectives and Limitations . . . 4
- 1.6 System Requirements . . . 5
- 1.7 Potential System Incompatibilities . . . 5
  - 1.7.1 Systems with a Non-Standard User Number Range . . . 5
  - 1.7.2 CDB and Your System's Restart Vectors . . . 6
  - 1.7.3 BDOS and BIOS Calls On Some CP/M "Look-Alike" Systems . . 6
- 1.8 How to Use The Compiler . . . 7
  - 1.8.1 The Commands and Primary Data Files . . . 7
  - 1.8.2 Configuration . . . 7
    - 1.8.2.1 Compiling CCONFIG.C . . . 8
    - 1.8.2.2 CC and CLINK configuration . . . 8
    - 1.8.2.3 CC2 Configuration . . . 11
    - 1.8.2.4 Run-Time Package Options . . . 11
    - 1.8.2.5 STDIO.H and HARDWARE.H Configuration . . . 12
  - 1.8.3 A Sample Compilation . . . 13
  - 1.8.4 CC — The Parser . . . 13
  - 1.8.5 CC2 — The Code Generator . . . 17
  - 1.8.6 CLINK — The C Linker . . . 18
  - 1.8.7 CLIB — The C Librarian . . . 22
- 1.9 CP/M "Submit" Files . . . 26
- 1.10 Operational Caveats . . . 26
- 1.11 Last Words . . . 27

Chapter 2 The CRL Function Format and Other Low-Level Mechanisms 29
- 2.1 Introduction . . . 29
- 2.2 The CRL Format in Detail . . . 29
  - 2.2.1 CRL Directories . . . 29
  - 2.2.2 External Data Area Origin and Size Specifications . . . 30
  - 2.2.3 Function Modules . . . 30
    - 2.2.3.1 List of Needed Functions . . . 31
    - 2.2.3.2 Length of Body . . . 31
    - 2.2.3.3 Body . . . 31
    - 2.2.3.4 Relocation Parameters . . . 32
- 2.3 Register Allocation and Function Calling Conventions . . . 32

2.3.1 The Stack 32
2.3.1.1 The Stack Pointer 32
2.3.1.2 How Much Space Does the Stack Take Up? 32
2.3.2 External Data 33
2.3.3 Function Entry and Exit Protocols 33
2.4 Re-entrant Coding 35
2.5 Helpful Run-Time Subroutines Available in C.CCC (See CCC.ASM) . 35
2.5.1 Local and External Fetch Routines 35
2.5.2 Formal Parameter Fetches 36
2.5.3 Arithmetic and Logical Subroutines 36
2.5.4 System Source Files 36
2.6 Debugging Object Command Files Under CP/M 37
2.6.1 Loading Programs and Setting Breakpoints 37
2.6.2 Tracing Execution and Dumping the Values of Variables 39
2.6.3 A Sample SID Debugging Session 39

Chapter 3 The BDS C Standard Library on CP/M: A Function Summary 43
3.1 General Purpose Functions 43
3.2 Character Input/Output 54
3.3 Character and String Processing 59
3.4 File I/O 63
3.4.1 Introduction to BDS C File I/O Functions 63
3.4.2 Filenames 63
3.4.2.1 The Disk Designator Prefix 63
3.4.2.2 The User Area Prefix 63
3.4.3 Error Handling 64
3.4.3.1 The Errno/Errmsg Functions 64
3.4.3.2 Random-Record Overflow 64
3.4.4 The Raw File I/O Functions 64
3.4.5 The Buffered File I/O Functions 69

Chapter 4 Notes to APPENDIX A of “The C Programming Language” 75
4.1 Introduction 75
4.2 Notes to Appendix A 76

Chapter 5 The RED Screen Editor 89
5.1 How To Install RED 89
5.1.1 Run the Configuration Program 89
5.1.1.1 Setting Defaults 90
5.1.1.2 Selecting Control Keys 91
5.1.1.3 Describing Your Terminal 91
5.1.2 Compile and link RED 92
5.1.3 Test and use RED 93
5.1.4 (Optional) Run STEST 93
5.1.5 Additional Features for RED Under BDS C v1.6 93
5.2 Reference Manual 94
Starting RED 94
Using Function and Control Keys 95
Changing Modes 96

Inserting Characters With Insert and Overtype Modes 96
Inserting New Lines 97
Moving The Cursor 97
Deleting Characters and Lines 98
Undoing Mistakes 98
Splitting and Joining Lines 98
Inserting Control Characters 99
Repeating the Previous Function 99
Using Commands 99
Creating, Saving and Loading Files 100
Leaving RED 101
Searching for Patterns 101
Moving Blocks of Lines 104
Setting Tab Stops 105
Enabling and Disabling Line Wrapping 105
Listing the Buffer 105
Deleting Multiple lines 105
Choosing How RED Switches Modes 106
Edit Mode Functions And Escape Sequences 106
What To Do About Error Messages 108

Chapter 6 CDB: A Debugger for BDS C 109
6.1 An Explanation of CDB Components 109
6.2 Constructing the Debugger 110
6.2.1 Constructing L2 110
6.2.2 Constructing CDB2 110
6.2.2.1 The CDBCONFG Utility 110
6.2.2.2 CDB System Description 111
Constructing CDB 111
Constructing CDB2 111
Changing the restart number 112
6.3 How to Invoke the Debugger 113
6.3.1 Compilation: The –K Option of CC 113
6.3.2 Linkage: The –D and –S Options of L2 113
Invoking CDB 114
6.3.3 Summary 114
6.4 Debugging Commands: How to Use the Debugger 115
6.4.1 Breakpoints 115
6.4.2 Executing code 117
6.4.3 Dumping variables 117
6.4.4 Setting variables 119
6.4.5 The list command — various items of information 119
The quit command 120
6.5 Alphabetical Listing of Debugger Commands 120
6.6 An Example — A CDB Debugging Session 121

Chapter 7 Tutorials and Tips 129
7.1 BDS C File I/O Tutorial 129
7.1.1 Introduction 129

7.1.2 The Raw File I/O Functions 129
7.1.3 The Buffered File I/O Functions 133
7.2 BDS C Console I/O: Some Tricks, Clarifications and Examples 136
7.2.1 Introduction 136
7.2.2 Elementary Console Interfacing 137
7.2.3 The BDOS and How It Complicates Things 137
7.3 Some Mistakes Commonly Made By Beginning C Programmers and Other Things Deserving Clarification 141
7.3.1 '=' versus '==' 141
7.3.2 Character Constants within Literal Strings 142
7.3.3 The Precedence of Assignment Operators 142
7.3.4 Array Subscripting 143
7.3.5 How NOT To Use a Pointer 143
7.3.6 Functions Shouldn't Return Pointers to Their Automatic Data . 143
7.3.7 Understanding Formal Parameters 144
7.3.8 Dependence on Parameter Evaluation Order 145
7.3.9 Function Calls MUST Have Parentheses 145
7.4 Miscellaneous Notes 145

Chapter 8 Auxiliary BDS C Package Programs 151
8.1 The CASM Assembly-language-to-CRL-Format Preprocessor For BDS C 151
8.1.1 Creating CASM.COM 152
8.1.2 Command Line Options 152
8.2 The L2 Linker 154
8.3 The CMODEM Telecommunications Program 157
Installation 157

Chapter 9 Auxiliary BDS C Libraries 161
9.1 BDS C v1.5 Compatibility Library 161
9.2 A BCD Function Package For BDS C 161
9.2.1 Description of Files 161
9.2.2 Data Representation 162
9.2.3 Testing For Zero 163
9.2.4 Rounding and Accuracy 163
9.2.5 Error Handling 164
9.2.6 The Return Values 164
9.2.7 Transportability 165
9.2.8 Configuration 165
9.2.9 Changing Precision 166
9.2.10 Rebuilding BCD.CRL 166
9.2.11 Linking to the BCD Functions 166
9.2.12 BCD Package Function Summary: 167
9.3 A Long Integer Package for BDS-C 172
9.3.1 Introduction 172
9.3.2 Implementation Details 175

Appendix A Dynamic Overlays in C Programs 177

Appendix B Customizing The Run-Time Environment 183
B.1 Standard vs. Customized Environments 183
B.2 Simple Run-Time Package Customization 183
B.3 Creating New Customized Environments 184
B.4 Making Code Run Without CP/M 186

Appendix C BDS C Evolution: A Version-By-Version Update Summary 189
C.1 Changes for BDS C v1.6 189
Library Changes 189
Run-Time Package 191
Compiler 191
Utilities 192
C.2 Changes for BDS C v1.5 193
C.3 Changes For BDS C v1.4 197

Appendix D Error Messages Explained 199
D.1 CC Error Messages 199
File I/O Errors 199
Overflow Conditions 200
Preprocessor Errors 200
Syntax Errors 201
Declaration Errors 203
Miscellaneous errors 204
D.2 CC2 Error Messages 204
File I/O, Syntax, Overflow and Other Miscellaneous Errors 205
Errors in Expressions 206
D.3 CLINK Error Messages 207

Index 211

Chapter 1

Introduction

Leor Zolman
BD Software
74 Marblehead Street
North Reading, MA 01864
(978) 664-4178
leor@bdsoft.com

1.1 Hello There

Thank you for purchasing BDS C. This software package is one programmer's personal implementation of the systems programming language **C**, geared exclusively for microcomputers running the CP/M-80 operating system. The primary design goal with BDS C was to create a C development environment that would allow programmers to move forward at a rapid, **efficient** pace as structured programs are developed—even on floppy-based systems. To this end, the compiler and linker were designed to perform as much of their work as possible in memory, with a minimum of disk activity. The result is a development cycle fast enough to support repetitive program compilation, linkage and execution without inducing nervous disorders in its users.

1.2 Quick Start

If you have just opened up your package and would like to see the compiler "do something" as quickly as possible, then follow these steps:

1. First, put write protect tabs on your master disk set. Make sure that the disks are *readable*; if they aren't, then check with your software vendor to make sure you got the correct disk format for your computer system. Once you've determined that the disks are OK, make copies immediately and set aside your original disks for safekeeping. It is **very easy** for files (and even entire disks) to be accidentally erased during the process of debugging certain types of C programs. Don't take chances; **make back-ups ofyour workfrequently!**

2. Choose a convenient working area (either an applications drive, or a clean user area on a hard disk system) having at least 100K free. Into this area, copy the following files from the first distribution diskette: CC.COM, CC2.COM, CLINK.COM,

C.CCC, DEFF.CRL, DEFF2.CRL, STDIO.H and TAIL.C. After copying these files, make sure there are at least 20K bytes free.

3. Make the work area chosen above be the currently logged drive/user area. Give the following sequence of commands:

```
cc tail
clink tail
type tail.c
tail tail.c -5
```

4. When you have completed the sequence of commands listed above, the final command should have caused the final five lines of the file TAIL.C to be printed on the console output. These five lines should be an identical match to the last five lines of output from the previous TYPE command. If this is indeed what happened, then you are up and running, and you've also got yourself a new general purpose utility named TAIL.COM that will instantly print out the last n lines of any text file, no matter how large the file.

5. If the TAIL command did not work, or if you got any errors during any of the previous commands, then check the checksums (using CRCK.COM and CRCKLIST.CRC) of all files in the work area that were copied from the distribution disk. If the checksums do not match, then the file(s) whose checksums didn't match were incorrectly written; contact your dealer to obtain replacement files. If the checksum values do indeed match those listed in the CRCKLIST.CRC file for all files being used, then there is something really wrong and you should obtain help from either your dealer, a local BDS C guru, or from BD Software directly.

6. You are now ready to set up a permanent working environment for your BDS C package. Read the section later in this chapter entitled **Configuration** in order to familiarize yourself with all the configuration options, especially the concept of the *Default Library Area*. Note that the selection of an explicit default library area (separate from your C source program work area) is optional, depending on how much disk storage you have on your computer system. If you have less than 250K of storage per floppy and no hard disk system, don't worry about selecting a default library area. If you have over 500K per floppy and/or a hard disk, then it is definitely recommended that you select a default library area. If you have no hard disk and a data capacity somewhere between 250K and 500K per floppy, then whether or not you choose to organize your support files in a default library area depends on your own particular taste.

7. Now select a default library area if you wish, or else just pick a particular disk drive and user area where you are going to be doing all your BDS C development work. Make that area the currently logged drive/user area, and copy all the files listed in step 3 (except TAIL.C) here. If the file CCONFIG.COM is provided on your distribution disk set (it would be the last file of the package), then copy it into your currently logged drive/user area and skip the rest of this step. If CCONFIG.COM is not provided, then copy in the files CCONFIG.H, CCONFIG.C, and CCONFIG2.C. Make sure you have at least 50K of free disk space, then issue the commands:

```
cc cconfig.c -e5000
cc cconfig2.c -e5000
clink cconfig cconfig2
```

8. Enter the command

```
cconfig
```

9. You should now be under the control of the CCONFIG utility. If you have chosen to use a default library area, then pick options 0) and 1) to define that area. Then go through as many of the other options as you wish to customize. When you are done, give the "q" command to quit, and let CCONFIG write out the changes. You now have new versions of CC.COM and CLINK.COM that have been customized for your system. (Note that in Chapter 1 of the User's Guide, the sequence given for compiling CCONFIG is incorrect. Be sure to use the sequence shown in the previous step, unless you were provided with a pre-compiled CCONFIG.COM.)

10. If your CP/M system knows about command path searching (as, for example, when running under ZCPR or MicroShell), then place copies of the CC.COM and CLINK.COM commands just created, plus CLIB.COM from the distribution package, into the system command directory. (You may also wish to place TAIL.COM there.)

11. If you have chosen to select a default library area separate from your BDS C work area, then you can now test the default library area feature. To do so, choose an *empty* drive/user area combination. If your system does not support command path searching, then place copies of **only** the two files CC.COM and CLINK.COM, as created above, into this new empty area. Next, copy TAIL.C into this new area, and then issue the same sequence of commands as shown in step 4 above. Things should happen exactly as described in step 5.

12. Now finish reading the rest of the User's Guide, because everything is working perfectly.

1.3 Support

For emergency technical help with BDS C, the author is generally available for telephone consultation free of charge. If you call and get the answering machine, please briefly describe your problem and note the best time and place where you may be reached. I will do my best to return your call as soon as possible.

It is often necessary for me to see the exact code you have written in order to determine the nature of an obscure bug; i.e., is it a compiler bug or a coding error? If you have a program or program fragment you would like to have me look at, please put it on a disk (if typing it in would take longer than a minute) and include a) an example of what it does, and b) your idea of what it *should* do. If what you have is indeed a compiler bug, I will fix it and get an update out to you free of charge as soon as possible. If the problem is yours, you'll be so informed.

The **C User's Group** is an organization that manages a library of contributed software, provides compiler updates to registered BDS C users and publishes an excellent magazine to announce library contributions and provide state-of-the-art C language news and articles. While not affiliated with BD Software, the CUG has for several years been the central depository for user-submitted applications written in BDS C. The group is now expanding to support other C compilers and related applications. A membership application form for the CUG is included with this BDS C package.

1.4 No Royalties, Of course!

There are no library usage licenses or royalty contracts connected with this package. Users are free to develop software in BDS C and market the resulting object code, along with any functions that may have been taken from the BDS C library, without paying any royalties to BD Software. The whole idea behind this policy is to encourage potential software vendors to use C for their development work. Note that this has **always** been the policy with BDS C, since the package first went on sale in 1979. Other vendors have had to learn the hard way…

If software authors using BDS C for their product development would please mention that fact in the documentation for their products, it would be highly appreciated. In the past, I've been both flattered and perturbed to find literal pieces of BDS C library source code in the libraries of **other** C compiler packages; this probably wouldn't have bothered me had I at least been given some **credit** for my code.

1.5 Objectives and Limitations

The BDS C Compiler is the implementation of a healthy subset of the C Programming Language originally developed at Bell Laboratories in conjunction with the Unix operating system[1]. The compiler itself runs on 8080/Z80 microcomputer systems equipped with the CP/M-80 operating system, and generates code to be run either under CP/M or in any kind of dedicated 8080/Z80 hardware environment containing at least a token amount of RAM for stack and scratch pad memory.

The main objective of this product is to translate a bit of the powerful, structured programming philosophy on which the Unix operating system is based from the minicomputer to the microcomputer environment. BDS C provides an efficient and friendly environment in which to develop CP/M utility applications or stand-alone system software, with emphasis on an elegant, efficient human interface for both compiler and end-application usage.

1. See The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie (Prentice Hall, 1978) for a complete description of the language. This guide deals only with details specific to the BDS C implementation; it does *not* attempt to teach the C language.

1.6 System Requirements

The practical minimum hardware configuration required by BDS C is a 40K **CP/M 2.x** system with either two disk drives of at least 100K capacity each, or at least one drive of 200K capacity. While it is **possible** to use the package on a system with only one low-capacity disk drive, such use is not recommended.

BDS C loads an entire source file into memory at once (making use of the entire available memory space) and performs the compilation directly in RAM, as opposed to passing the source text "through a window" (operating on disk files). This allows a compilation to proceed quite rapidly (in contrast to conventional compilers). A consequence of this scheme is that a source file must fit entirely into memory for the compilation. While this may sound restrictive at first, consider: a *program* in C is actually a collection of many smaller *functions*, stemming from a single **main** function at the top level. Each function is treated as an independent entity by the compiler, and may be compiled separately from the other functions that make up a complete program. Thus, a single program may be spread out over many source files, where each source file contains a variable number of functions. Partitioning a program into several files actually helps to minimize compilation time following minor changes, as well as keep the individual source files from overflowing available memory restrictions.

1.7 Potential System Incompatibilities

This section warns of several potential compatibility problems and describes how to reconfigure the package "around" those problems.

The BDS C "run-time package" is a 1.5K binary file named "C.CCC" that is always attached onto the beginning of compiled programs during linkage. The source code to C.CCC, written in assembly language compatible with ASM.COM, MAC.COM and M80.COM, is provided as "CCC.ASM". In the following discussion of system incompatibilities, the solution to a given problem may involve making a change in the run-time package and "creating a new C.CCC". The exact sequence of commands necessary to create a new C.CCC from the CCC.ASM source varies depending on whether you are using ASM/LOAD, MAC/LOAD or M80/L80. See the comments at the beginning of CCC.ASM for detailed assembly instructions using each of these different assembler packages.

1.7.1 Systems with a Non-Standard User Number Range

The v1.6 run-time package file-I/O mechanism for BDS C presumes that the target programs are being run on a standard CP/M system where the "user area" numbers may range from user 0 to user 31. Since only 5-bits are allocated for user-number memory within the internal file descriptor routines, standard BDS C generated COM files may not run on certain "CP/M-like" operating systems which support user numbers larger than 31. In order to fix this problem, the symbol USAREA has been added to the configuration area at the start of the run-time package source file (CCC.ASM). If you are experiencing problems opening files on your non-standard CP/M system, try changing this symbol to FALSE, re-assembling CCC.ASM to yield a new C.CCC, and re-linking your C program with this new version of C.CCC. Note that while this fix

will allow your programs to run, you will no longer be able to make use of the user-number prefix feature when specifying filenames at run-time. All files will be expected to reside in the currently-logged user area. Drive prefixes may still be specified, of course.

1.7.2 CDB and Your System's Restart Vectors

The CDB symbolic debugger system requires, for its standard operation, the availability of an unused restart vector down in low system memory. Out of the eight physical restart vectors on every system, two (RST 0 and RST 7) are always reserved for use by CP/M; the others may or may not be used depending on your system implementation. By default, CDB comes pre-configured to use RST 6 (location 0030h) for its debugging operation. If RST 6 is already used by your system (this may be the case if your system uses interrupt-driven console I/O), then you will have to pick an alternate restart location. Consult your system documentation to find out which restart vectors are unused.

The run-time package can be configured to initialize the CDB restart vector to act as a CDB "no-op", allowing C programs compiled using the CC option **–k** to execute properly whether or not they are invoked under CDB. By default, the package does **not** initialize restart locations; this prevents compiled programs from coming up and wiping out what might possibly be vital interrupt handling vectors on your system. Once you have decided on a safe restart vector to assign to CDB operation, then 1) change the equated symbol "USERST" in CCC.ASM to TRUE, 2) set the "RSTNUM" symbol to the value of the restart location you want to use, and 3) create a new C.CCC. Remember that if you decide to use a restart other than RST 6, then you must rebuild CDB as described in the CDB chapter.

1.7.3 BDOS and BIOS Calls On Some CP/M "Look-Alike" Systems

The *bios*, *bdos* library functions are provided with BDS C as general-purpose hooks into the low-level system interface of CP/M. These functions grant the C programmer access to **all** CP/M 2.x system calls, and most system calls on CP/M "look alike" and "extended" operating systems.

To achieve total generality, though, the *bdos* and *bios* functions make certain assumptions about which registers CP/M's BDOS and BIOS use to return results from system calls. These assumptions are valid under all CP/M systems, but do not necessarily hold true under certain CP/M "look-alike" systems (such as SDOS or CDOS). If you are trying to use the *bios* or *bdos* functions and things don't seem to be working correctly, check to make sure your operating system returns BDOS values in the HL register and BIOS values in the A register…if this is not the case, you must rewrite the *bdos* and/or *bios* functions to obtain operating system return values from the proper registers.

The *bios* library function as supplied with BDS C makes the assumption that location 0000h of your system (the first instruction of the CP/M base page) contains a direct jump to the second entry (**wboot**) in the BIOS jump vector table. If this is not the case, such as on the Xerox 820, the *bios* function will not work correctly on your system and you must rewrite it to compute the address of the BIOS vector table in whichever manner is appropriate for your particular system.

1.8 How to Use The Compiler

1.8.1 The Commands and Primary Data Files

The main BDS C package consists of four executable commands:

CC.COM	C Compiler -- phase 1
CC2.COM	C Compiler -- phase 2
CLINK.COM	C Linker
CLIB.COM	C Librarian

and three data files that are usually required by the linker:

C.CCC	Run-time initializor and subroutine module
DEFF.CRL	Standard ("Default") function library
DEFF2.CRL	More standard library functions

CC.COM and CC2.COM together comprise the actual compiler. CC reads in a given source file from disk, processes it, leaves an intermediate file in memory, and automatically loads in CC2 to finish the compilation and produce a CRL (C relocatable object module) file as output.[2] The CRL file contains the generated machine code in a special relocatable format.

The linker, CLINK, accepts a CRL file containing a "main" function and proceeds to conduct a search through all given CRL files (then DEFF.CRL, DEFF2.CRL and DEFF3.CRL if present, automatically) for needed subordinate functions. When all such functions have been linked, a COM file is produced.

For convenience, the CLIB librarian program is provided for the manipulation of CRL object libraries.

1.8.2 Configuration

> Make sure to have your master distribution disk safely tucked away somewhere before attempting any of the modifications described in this section!

As distributed, BDS C commands should simply come up running under any CP/M system (CP/M version 2 or above, that is). There *are* several user-configurable, system-dependent features of the compiler and linker that may be controlled by the user to "custom-fit" the package to specific systems. This subsection explains each of those options and how to select them. Note that **no** special configuration procedure should be needed in order to run the compiler "right out of the box".

2. If desired, the intermediate file produced by CC may be written to disk and processed by CC2 separately; in that case, the intermediate file is given the extension .CCI

1.8.2.1 Compiling CCONFIG.C

There are several user-configurable features in CC.COM and CLINK.COM controlled by specific bytes of memory very close to the beginning of each command file. The program CCONFIG.C has been provided to automate process of configuring these option bytes for your specific system.

To compile CCONFIG.C, first copy the following seven files into a free work area on your system: CC.COM, CC2.COM, DEFF.CRL, DEFF2.CRL, C.CCC, STDIO.H, CCONFIG.C. You should have at least 40K free in order to perform the following compilation.

Now enter the command:

```
A>cc cconfig.c -e5000
```

The compiler should respond with something like:

```
BD Software C Compiler v1.60  (part I)
   3K elbowroom
BD Software C Compiler v1.60 (part II)
  14k left over
A>
```

There will be a delay of about 45 seconds after the first line is printed, and another delay of about 10 seconds after the third line (assuming a 4 MHz CPU clock rate). When control returns to command level, enter the command:

```
A>clink cconfig
```

The Linker will respond with:

```
BD Software C Linker  v1.6

Last code address: 4EAE
Externals start at 5000, occupy 012C bytes, last byte at 512B
Top of memory: D305
Stack space: 81DA
Writing output...
  24K link space remaining
A>
```

You should now find that the file CCONFIG.COM has been created. Don't erase it; you'll be using it in the next section.

1.8.2.2 CC and CLINK configuration

Read this entire section first so that you understand what the function of each CC/CLINK options is, then run CCONFIG.COM with copies of CC.COM and CLINK.COM being present in the current directory in order to configure CC and CLINK for your own system.

Both CC.COM and CLINK.COM contain an identically structured ten-byte configuration block.[3] The structure of the block is as follows:

3. The starting address of the block for CC.COM is 0155h, and for CLINK it is 0103h.

Addr.	Function	Default value [hex]
base+0	Default library disk	FF (current)
base+1	Default library user area	FF (current)
base+2	Disk where SUBMIT files are processed	00 (disk A)
base+3	Poll console for interrupts?	01 (yes)
base+4	Suppress warm boot when finished?	00 (yes)
base+5	Strip parity bits from source input?	01 (yes)
base+6	Recognize user area mechanism?	00 (yes)
base+7	Write RED error file?	00 (don't)
base+8	Optimization mode	80 (short code)
base+9	Default CDB restart vector	06 (rst 6)

Each configuration item in the block is exactly one byte in length. Note that although all possible numeric values for the options are explained in detail below, it is not necessary for the first-time user to be concerned with those exact values as CCONFIG.COM will prompt with verbal explanations and accept non-numerical responses.

The first two configuration bytes specify a default disk drive and user area to be treated as a "default library area" by CC and CLINK. CC.COM searches this area to find the files named in **#include** directives when the filename is enclosed in angle brackets[4], and also for CC2.COM (the second phase of compilation). CLINK searches the default library area for the files DEFF.CRL, DEFF2.CRL, DEFF3.CRL (if present), C.CCC, and other CRL files named on the CLINK command line that cannot be found in the directory where the "main" CRL file resides.[5] For the default library disk, a value of 0 specifies drive A, 1 specifies drive B, etc., and a value of FFh (255 decimal) specifies that the *currently-logged* disk (at the moment CC or CLINK is invoked) is to be used as the default library disk. For the default library user area, the values 0-31 denote the corresponding user area, and a value of FFh (255 decimal) specifies that the *current* user area (at the moment CC or CLINK is invoked) is to be the default library user area. Both the library disk and user area come configured to FFh; thus, the v1.5 (and later) distribution versions of compiler and linker assume that the currently logged drive and user area contain all library files.

To summarize: the first two configuration bytes allow users with large-capacity disks to pick some particular drive and user area in which to keep all standard header and library files. The library disk and user area bytes should be considered together as a unit; if you change one, you'll probably also want to change the other.

The third configuration byte designates the disk drive containing the $$$.SUB file that exists during "Submit File" processing. The possible values are the same as for the default library disk described above.

CLINK always tries to erase pending submit files when an error occurs, while CC only tries to do so when the **–x** option is given. Since most systems always place the $$$.SUB file on drive A, that is the way CC and CLINK are configured by default. But, if the user has customized his system to put the $$$.SUB file on, say, the current drive instead of always on drive A, then this byte should be changed from 01h to 0FFh.

---

4. Filenames enclosed in double quotes always cause the **#include** directive to search the current directory for the named file, regardless of configuration.

5. the L2 linker may also be configured to search a default drive/user area; to do so, it is necessary to customize the L2.C source file and recompile L2 as per the compilation instructions specified in the comments at the start of L2.C

The fourth configuration byte is a flag telling CC or CLINK whether or not the system console should be polled for the interrupt character (control-C) during execution of the command. If enabled (non-zero), then any input typed on the console by the user during execution of the command will be ignored *unless* control-C is typed, in which case the command will be immediately aborted and control will return to command level. If disabled (zero), then the console will never be polled. This is useful under certain interrupt driven systems that can recognize type-ahead and handle interruption on their own without requiring transient commands to poll the console.

The fifth configuration byte controls whether CC and CLINK perform a "warm-boot" when finished with their tasks, or return directly to the CCP without any disk activity. The commands come configured to return directly to the CCP, but on certain "fake" CP/M systems (I've been told the CROMIX CP/M emulator is one example), directly returning to the CCP does not work correctly. This is probably because the operating system doesn't pass a valid stack pointer to transient commands, and when CC or CLINK tries to return, it crashes the system. If you run the compiler and it bombs after writing a correct output file, try setting the warm-boot byte to 01 to force warm boots on program termination.

The sixth byte controls the stripping of bit 7, the parity bit, from each character of source file input to the compiler. If true, as configured by default, all parity bits are stripped during compilation. This avoids the problems caused by text editors that use parity bits to convey formatting information. Certain applications, though, require parity to **not** be stripped. For example, some foreign-language or dual-language computers use bit 7 to represent alternate character sets in string operations. If you have such a system, change this configuration byte to false (zero). Note: this byte has no effect in CLINK's configuration block.

The seventh configuration byte is intended for use only in those customized operating system environments that perform transparent user area selection for transient programs. MICROSHELL is one system under which this option would be useful. If true (non-zero), then all user-area selection (for source and library files) is inhibited during compilation and linkage, allowing your operating system to select a user area as per its own particular algorithms. If this option is set to true, then the default library user area configuration byte (the second byte of the configuration blocks) has no effect.

The eighth configuration byte controls whether or not an error documentation file is automatically written for the benefit of the RED text editor. If set to true (01), then whenever there are errors in the source file, the RED error file PROGERRS.$$$ is automatically written to disk. Then the user may, by simply typing the command RED, invoke RED.COM upon the buggy source file and have all errors pointed out automatically. If the RED error control byte is set to false (00), then the compiler option –W must be specified to cause RED error file output.

The ninth configuration byte controls object code optimization for the current compilation:

- A value of 00 specifies optimization for speed only, so that the fastest possible code sequences are used. This will usually result in longer code than normal.

- A value of 80h (the default) specifies optimization for space, resulting in the compacting of certain code sequences into calls to equivalent run-time package subroutines.

- If any of the lower 7 bits of the configuration byte are true, then "restart optimization" is to be performed by taking advantage of the corresponding restart vector on the target computer. The mapping of bits is b0-rst 1, b1-rst2, …, b6-rst7. In order for restart optimization to work, a special version of the run-time package must be created with the corresponding restart vector equates set TRUE so that proper initialization of the target system's restart vectors is performed on application start-up.

The tenth (and final) configuration byte specifies the default restart vector to be used for CDB (C Debugger) interfacing. The value specified becomes the default argument to the –K compiler option. The default value is 6 (use rst 6), and the distributed compiled version of CDB is correspondingly set up to use rst 6.

1.8.2.3 CC2 Configuration

CC2.COM requires no configuration; CC.COM passes all relevant information to CC2.COM upon automatic transfer of control during compilation.

1.8.2.4 Run-Time Package Options

CCC.ASM, the source file of the BDS C run-time package, contains some equated symbols which control options that may be customized by the user. If you decide to alter any of the described options, you must re-assemble the run-time package object module according to the directions given in the comments at the beginning of CCC.ASM, resulting in the creation of a new run-time package object module, C.CCC.

The options described in this section may be altered in the run-time package **without** requiring any modification to the standard library. For information on customizing the run-time package and standard library for a drastically different environment (e.g., for a ROM-based application), see the "Customized Environments" appendix.

- The USAREA symbol specifies whether or not the target system recognizes the "user area" mechanism of CP/M. Setting this to FALSE eliminates all calls to the get/set user area BDOS function.

- The USERST symbol specifies whether or not a restart vector is reserved for use by the CDB debugger. If TRUE, then the RSTNUM symbol specifies the restart vector used, and causes an initialization of that restart vector so that programs compiled with –K (the CDB debugger option) run stand-alone (i.e., not under control of CDB) can still execute correctly.

- The USERST and RSTNUM symbols in CCC.ASM control the mechanism whereby a restart location in the CP/M base page (usually RST 6, at location 0030h) is set up allow C programs compiled with the **–k** CC debugger option *but executed independently of CDB (the c debugger)* to execute properly. As distributed, USERST is set to FALSE, and programs compiled using **–k** will not run unless invoked by CDB. To allow these programs to execute independently of CDB, first change the USERST symbol to TRUE. If restart vector 6 is available on your system for use by BDS C, then everything is all set. If the RST 6 location is used for I/O on your system, then you must find a restart

vector that is not in use, alter RSTNUM accordingly, and make sure to customize and recompile CDB to reflect the new restart vector assignment. CCC.ASM will then need to be reassembled to create a new C.CCC run-time package object file.

- The ZOPT1, ZOPT2, …, ZOPT7 symbols control the initialization of up to seven restart vector optimization sequences. In conjunction with the **–z** compiler option, this mechanism allows for the collapse of several common code sequences into one-byte restart instructions in the compiled object code. To utilize this mechanism, first choose which restart vectors are available on the target computer system. Then set the corresponding ZOPTn symbols true in CCC.ASM, and follow the procedure for re-creating the run-time package object module (C.CCC). The new C.CCC may then be linked with CRL files created via compilations using the **–z** option (with the available rst vectors specified as arguments to **–z**).

  > Note: object code produced under this procedure is extremely **non-portable**, and thus should not be distributed except for use under known hardware configurations where the restart vectors utilized in the code are **known** to be available for use by application programs.

- The NFCBS symbol specifies the maximum number of files that may be open at any one time. This is set to 8 for the distribution version; if you need more files open at once, change NFCBS to the desired value (each additional file makes the run-time package about 38 bytes longer.)
  IMPORTANT: If you change this value, you must re-assemble the entire assembly language library (as described in Chapter 2) in order to align addresses

1.8.2.5 STDIO.H and HARDWARE.H Configuration

The standard I/O header file **stdio.h** contains the defined constant NSECTS, which controls the number of 128-byte sectors kept in memory for each file opened under the standard buffered I/O library. NSECTS comes configured to 8, so that a full 1024 bytes of data are buffered during buffered I/O operations before disk activity occurs. If you are running a system that has 1K sector blocking/de-blocking in the BIOS (Basic Input/Output System) portion of CP/M, then you might want to change NSECTS from 8 to 1 in order to eliminate the redundant buffering and gain 7/8 K bytes of free memory per open file (under buffered I/O only). If you do decide to change NSECTS, afterwards don't forget to recompile STDLIB1.C (the source file for the functions in which the NSECTS symbol is referenced). Then, use CLIB.COM to combine STDLIB1.CRL, STDLIB2.CRL and STDLIB3.CRL to create a new DEFF.CRL. See the "Customized Environments" appendix for more details about recompiling/reassembling the BDS C standard library.

The HARDWARE.H header file provides a generalized interface to system-dependent hardware devices, such as direct I/O port control to both the console and modem devices. Before any programs which include HARDWARE.H are compiled, HARDWARE.H should be customized to reflect the hardware characteristics of the target computer system.[6]

6. As distributed, only the TELED.C sample program (with its associated files) actually uses HARDWARE.H. If you do not know the port and mask values of your machine's I/O ports, everything will still work except for the TELED program.

1.8.3 A Sample Compilation

As an example, here is the sequence for compiling and linking a simple source file named TAIL.C:

The compiler is invoked with the command:

```
A>cc tail.c <cr>
```

After printing its sign-on message, CC will read in the file TAIL.C from disk and crunch for a while. If there are no errors, CC will then give a memory usage diagnostic and load in CC2.COM. CC2 will do some more crunching and, if no errors occur, will write the file TAIL.CRL to disk.

The next step brings in the linker:

```
A>clink tail -n <cr>
```

This invocation of CLINK links TAIL.CRL with any needed library functions from DEFF.CRL and DEFF2.CRL. The file TAIL.COM should be produced, ready for execution. The **–n** option directs CLINK to make TAIL.COM do a "quick return" to CP/M without performing a warm-boot. To test TAIL.COM, say:

```
A>tail -10 tail.c <cr>
```

If everything is OK, then the last 10 lines of the text file TAIL.C should appear on the console and control should return (silently and instantly) to CP/M command level.

> **IMPORTANT:** The command lines for all COM files in the package should be typed in to CP/M **without leading blanks**. This also applies to COM files generated by the compiler, where leading blanks on the command line will cause *argc* and *argv* to be miscalculated.

Following are the detailed command syntax descriptions:

1.8.4 CC — The Parser

Command format:

```
A>cc name.ext [options] <cr>
```

Any name and extension are acceptable, although the conventional extension for C programs is "**.C**". CC will first try opening the file exactly as named; if no extension at all is given, and the file cannot be opened exactly as specified, then CC will append a ".C" extension onto the filename and try once more to open it with the newly constructed name.

If an explicit disk designator is given for the filename (e.g. "b:foo.c") then the source file is assumed to reside on the specified disk, and the compiler output also goes to that disk. Filenames

given in double quotes to the **#include** directive, with no explicit user-area/drive specification used, are obtained from the same disk as the master filename given on the command line.

Typing a control-C at any time after invoking CC will abort the compilation and return to command level **unless** CC has been configured to ignore the console, as described in the configuration section above.

Following the source filename may appear a list of compilation options, each preceded by a dash. The currently supported options are:

–a ***d*** **[*n*]** Auto-loads CC2.COM from disk *d*, user area *n*, following successful completion of CC's processing. By default, CC2 is assumed to reside either on the currently logged-in disk or on the default drive/user area as defined in the configuration procedure. If the letter "z" is given for the disk designator, then an intermediate ".CCI" file is written to disk for later processing by an explicit invocation of CC2, and no attempt is made to auto-load CC2.

–c Disables the "comment nesting" feature, causing comments to be processed in the same way as by UNIX C. I.e., when **–c** is given, then lines such as

```
/* printf("hello"); /* this prints hello */
```

are considered **complete** comments. If **–c** is **not** used, then the compiler would expect another ***/** sequence before such a comment would be considered terminated.

–d ***x*** Causes the CRL output of the compiler to be written to disk *x* if no errors occur during CC or CC2. If the **–a z** option is also specified, then **–d** specifies onto which disk the .CCI file is written. The default destination disk is the same disk from which the source file was obtained.

–e ***xxxx*** Allows the specification of the exact starting address (in hex) for the external data area at run time. Normally, the externals begin immediately following the last byte of program code, and all external data are accessed via indirection off a special pointer installed by CLINK into the run-time package. When **–e** is used, the compiler can generate code to access external data directly (using **lhld** and **shld** instructions) instead of using the external data pointer. This will shorten and enhance the performance of programs having much external data. Suggestion: don't use this option while debugging a program; once the program works reasonably, *then* compile it once with **–e**, putting the externals at the same place that they were before (since the code will get shorter the next time around.) Observe the "Last code address" value from CLINK's statistics printout to find out by how much the code size shrunk, and then compile it all again using the appropriate lower address with the **–e** option. Don't cut it too close, though, since you'll probably make mods to the program and cause the size to fluctuate, perhaps overlapping the

explicitly specified external data area (a condition that CLINK will now detect and report).
CC2 must be successfully auto-loaded by CC in order for **–e** to have any effect.
See also the CLINK option **–e** for related details.
Note that CLINK will now print a warning message if the external data area overlaps either part of the program or the operation system in the final command file.

–m ***xxxx***

Specifies the starting location, in hex, of the run-time package (C.CCC) when using the compiler to generate code for non-standard environments.
The run-time package is expected to reside at the start of the CP/M TPA by default. If an alternative address is given by use of this option, be sure to reassemble the run-time package and machine language library for the given location before linking, and give the **–l**, **–e** and **–t** options with appropriate address values when using CLINK. See Chapter 2 for more details on customizing BDS C object code for non-standard environments.
C.CCC, which always resides at the start of a generated COM file, cannot be separated from **main** and other (if any) root segment functions.
CC2 must be successfully auto-loaded by CC in order for **–m** to have any effect.

–o

Causes the generated code to be optimized for speed. Normally, the code generator replaces certain awkward code sequences with calls to equivalent subroutines in the run-time package; while this reduces the length of the code, it also slows execution down because of subroutine linkage overhead. If **–o** is used, then many of those subroutine calls are replaced by in-line code. This results in faster (but longer) object programs.
For the **fastest** possible code, the **–e** option should be used in conjunction with **–o**. For the **shortest** possible code, use **–e** (and **–z** if applicable) but **don't** use **–o**.
CC2 must be successfully auto-loaded by CC in order for **–o** to have any effect.

–p

Causes the source text to be displayed on the user's console, with line numbers automatically generated, after all **#define** and **#include** substitutions have been completed. Note that this output may be directed to the CP/M "list" device by typing control-P before invoking CC.

–r x

Reserves *x*K bytes for the symbol table. If an "Out of symbol table space" error occurs, this option may be used to increase the amount of space allocated for the symbol table. Alternatively, if you draw an "Out of memory" error then **–r** may be used to decrease the symbol table size and provide more room for source text. A better recourse after running out of memory, though, would be to break the source file up into smaller chunks. The default symbol table size is 10K (as if –r10 were specified).

–w Causes an RED-compatible error file, named PROGERRS.$$$, to be written to disk if there are any fatal compilation errors, so RED may then be invoked to quickly peruse and correct the errors for the next pass through the compiler.
Note: If you have already configured CC (with CCONFIG) to always write the RED file upon error, then **–w** forces the RED file to **not** be written.

–x Causes the deletion of pending CP/M "SUBMIT" batch activity following a compilation in which any errors have occurred. Whenever CC is used from a SUBMIT file, **–x** should appear on the CC command line to erase the "$$$.SUB" temporary file before returning to command level following an erroneous compilation. When CC is used stand-alone, **–x** would just cause needless disk activity and should not be used.

–z This option is used to specify a list of restart vectors that will be available on the target system for use by the compiled object program to shorten certain common code sequences. **–z** takes a list of digits, separated by commas, corresponding to the restart vectors (from among the seven vectors 1 through 7) to be used at run-time. For example, to specify that rst 2, rst 3 and rst 5 are to be available, use:

```
-z2,3,5
```

Note that the restart vector list specifies must correspond to a customized run-time package object module (C.CCC) assembled with the corresponding ZOPTn symbols set to TRUE. The object code generated using this –z option must then be linked with the customized C.CCC. Code generated using –z will not work if linked with the standard distribution version of the run-time package object module!

A single C source file may not contain more than 63 function definitions; remember, though, that a C *program* may be made up of any number of source files, each containing up to 63 functions.

If any errors are detected by CC, the compilation process will abort immediately instead of proceeding to the second phase of compilation or writing the .CCI file to disk (depending on which options were given).

Execution speed: about 20 lines text/second. After the source file is loaded into memory, no disk accesses will take place until after the processing is finished. Don't assume a crash has occurred until at least (n/20) seconds, where n is the number of lines in the source file, have elapsed since the last disk activity was noticed… **Then** worry.

Examples:

```
A>cc foobar.c -r12  -ab  <cr>
```

invokes CC on the file foobar.c, setting symbol table size to 12K bytes. CC2.COM is auto-loaded from disk B.

```
A>cc c:belle.c -p  -o  <cr>
```

invokes CC on the file belle.c, from disk C. The text is printed on the console (with line numbers) following **#define** and **#include** processing. Unless CC finds errors, CC2.COM is auto-loaded from either the currently logged disk or the default drive/user area (configured as per section 1.8.2). The resulting code is optimized for speed.

1.8.5 CC2 — The Code Generator

Command format:

```
cc2 name <cr>
```

Normally CC2.COM is loaded automatically by CC.COM and this command need not be used. If given explicitly, then the file *name.CCI* will be loaded into memory and processed. If no errors occur, an output file named *name.CRL* will be generated and *name.CCI* (if present) will be deleted.

CC2 does not take any options.

As with CC, an explicit disk designator on the filename causes the specified disk to be used for input and output.

When CC auto-loads CC2, several bytes within CC2 are set according to the options given on the CC command line. If CC2 is invoked explicitly (i.e., not auto-loaded by CC) then the user must see to it that these values are set to the desired values before CC2 begins execution. Typically this will not be necessary, but if you're very low on disk storage and need to invoke CC2 separately, here are the data values that need to be set:

```
Addr    Default Option  Function
0103    00      -a      True if CC2 has been auto-loaded, else 0
0104    01      -o      0 if -o used (optimize for speed), else 1
0105-6  0100h   -m      Base address of C.CCC at object run-time.
0107-8  none            Explicit external address (if -e used)
0109    00      -e      True if -e used, else 0
```

The 16-bit values must be in reverse-byte order (low order byte first, high last). Note that not all CC command line options can be set up for a stand-alone run of CC2, only the ones shown above.
This information is provided for completeness only; only very rarely should any user have to think about going in and explicitly setting these values for a CC2 run.

CC2 execution speed: about 70 lines/second (based on original source text.)

If a control-C is typed on the console input at any time during execution, then compilation will abort, control will return to command level, and any pending submit file activity will be halted.

Example:

```
A>cc2 foobar <cr>
```

1.8.6 CLINK — The C Linker

Command format:

```
A>CLINK name [other names and options] <cr>
```

The file *name.CRL* must contain a **main** function; *name.CRL* and all other CRL files named (up to the appearance of a **–f** option) will have **all** their functions loaded into the linkage. If the **–f** option appears on the command line, then all CRL files named following it are **scanned** for needed functions; i.e, only those functions known to be needed by previously loaded functions (either from previous CRL files or from the one currently being scanned) are loaded into the linkage. When all explicitly named CRL files have been searched, the standard library files DEFF*.CRL will be scanned automatically for needed library functions. The order in which the library files are searched is always the same: first DEFF.CRL, then DEFF2.CRL, and finally, if supplied by the user, DEFF3.CRL. If the user writes functions having the same name as those in any automatic library file, then such functions should always be placed in one of the CRL files named explicitly on the command line. If placed in DEFF3.CRL, they would not get used unless the similarly named functions in DEFF.CRL and DEFF2.CRL were deleted from those files.

By default, CLINK assumes all explicitly named CRL files reside on the currently logged disk, and all library files (C.CCC and DEFF*.CRL) reside on the default drive and user area as defined by the configuration block. If an explicit drive designator prefixes the main filename on the command line, then the given drive becomes the default for all CRL files named on the command line. Each additional CRL file may contain a disk designator of the form "d:", and/or a user area prefix of the form "nn/", to specify an explicit place to find the file. If both prefixes are used, the user area prefix must come first.

If a named CRL file cannot be found according to the search rules above, then the directory specified by the default library drive and user area is also searched. This allows the user to place commonly used library files in one default drive/user area and have them be accessible during linkages performed in different drives and user areas.

If any unresolved references remain after all given CRL files have been searched, CLINK will enter an "interactive mode". Here the user will be shown the names of all missing functions and be given the opportunity to specify other CRL files to search.

Control-C may be typed during execution to abort the linkage and return to command level.

Intermixed with the list of file names to search may be certain linkage options, preceded by dashes. Note that multiple single-letter options may be combined following a single dash. The currently implemented options are:

–c *d* [*n*] Instructs CLINK to obtain library files (DEFF.CRL, DEFF2.CRL, C.CCC and possibly DEFF3.CRL) and any CRL files named on the command line but not found in the current drive/user area (or on the drive specified as prefix to the "main" CRL filename) from disk *d* and user area *n*. This option is used to override the default drive/user area specification hard-wired into the CLINK configuration block (see section 1.8.2).

–d ["args"]

"Debug" mode: For quick testing, **–d** causes the COM file produced by the linkage to be executed immediately instead of getting written to disk as a COM file. If a list of arguments is specified enclosed in quotes, then the effect is just as if the program was invoked from the CCP with the given command line parameters.
–d should **not** be used for segments having load addresses other than at the base of the TPA (i.e., **–d** should only be used for root segments).
Due to internal conflicts, **–d** will be ignored if the **–n** option is also given.

–e xxxx

Forces the base of the external data area to be set to the value *xxxx* (hex). Normally the external data area follows immediately after the end of the generated code, but this option may be given to override that default. This is necessary when chaining is performed (via *exec*, *execl* or *execv*) to make sure that the new command's notion of where the external data begins is the same as the old command's. To find out what value to use, first CLINK all the CRL files involved with the **–s** option, but without the **–e** option, noting the "Data starts at:" address printed out by CLINK for each file. Then link them again, using the **maximum** of all those addresses as the operand of the **–e** option for all files except the one that had the largest "Data starting address" during the first pass.
When generating code for ROM, this option should be used to place externals at an appropriate location in r/w memory.
If the main CRL file (*name.CRL*) was compiled with the **–e** CC option specified, then CLINK will automatically know about the address then specified on the CC command line; but if any of the other CRL files specified in the linkage contain functions compiled by CC without use of the **–e** option, or with the value given to **–e** being different from the value used to compile the main function, the resulting COM file will not work correctly. CRL files compiled without use of the CC **–e** option may be included in a linkage **only** if **–e** is specified to CLINK with an argument exactly equal to the CC **–e** argument used to compile the main CRL file.

–f (filename…)

Causes all following named CRL files to be **scanned** instead of **loaded**. CLINK automatically loads all functions in each CRL file named on the command line, until this option is encountered, at which point all following CRL files are scanned. This means that only functions which have been previously referenced by other functions, in some earlier file or in the current file, are linked into the program. Note: This new **–f** option works differently from the **–f** of pre-1.50 versions of BDS C. **–f** now works identically to the **L2** linker's "-L" option.

–l *xxxx*

Causes the load address of the generated code to be *xxxx* (hex). This option is only necessary when generating an overlay segment (in conjunction with **–v**) or creating code to run in a non-standard environment. In the latter case, CCC.ASM must have been reconfigured for the appropriate location and assembled (and loaded) to create a new

version of C.CCC having origin *xxxx*. In this case the **–e** and **–t** options should also be used to specify the appropriate r/w memory areas. **–t** *xxxx* Set start of reserved memory to *xxxx* (given in hexadecimal). The instruction **lxi sp,*xxxx*** is placed at the start of the generated COM file[7]. Under CP/M, the value should be large enough to allow all program code and local/external data to fit below it in memory at run-time. If you are generating code to run in ROM, then the value given here should be the highest address **plus one** of the read/write memory to be used for the stack.

–n Makes the resulting COM file preserve the CP/M CCP (Console Command Processor) at run-time, instead of overlaying the CCP with the run-time stack. This reduces the available run-time memory by 2K bytes, but allows the program to return instantly to command level after execution **without having to perform a warm-boot from disk**. Therefore, **–n** is useful for programs that are used often and do not require every last bit of memory in the system. Note that this option has exactly the same effect as running the NOBOOT command on the resulting COM file; NOBOOT is provided so that programs linked with other linkers, such as L2, may also be made to return to the CCP without performing a warm-boot.
–n is ignored if the **–t** option is also used, because the mechanisms conflict and **–t** is given priority.

–o newname Causes the COM file output to be named *newname.COM*. If a disk designator precedes the name, then the output is written to the specified disk. By default, the output goes to the currently logged-in disk. If a single-letter disk designator followed by a colon is given without a filename, then the COM file is written to the specified disk without affecting the name of the file.

–r xxxx Reserves xxxx (hex) bytes for the forward-reference table (defaults to about 600h). This option may be used to allocate more table space when a "ref table overflow" error occurs.

–v Specifies that an overlay segment is being created. The run-time package is not included in the generated code, since it is assumed that an overlay will be loaded into memory while a copy of the run-time package is already resident either at the base of the TPA by default, or at the address specified in the **–m** option to CC.

–w Writes a symbol table file with name *name.SYM* to disk, where *name* is the same as that of the resulting COM file. This symbol file contains the names and absolute addresses of all functions involved in the linkage. It may be used with SID for debugging purposes, or by the **–y** option when creating overlay segments (see below.)

---

7. Normally, when **–t** is not used, the generated COM file begins with the sequence:

```
lhld base+6     ;get BDOS pointer from base page
sphl            ;initialize stack pointer to BDOS base
```

–y sname Reads in ("yanks") the symbol file named *sname.SYM* from disk and uses the addresses of all function names defined there for the current linkage. The **–w** and **–y** options are designed to work together for creating overlays, as follows: when linking the "root" segment (the part of the program that loads in at the TPA, first receives control, and contains the run-time utility package), the **–w** option should be given to write out a symbol table file containing the addresses of all functions present in the root. Then, when linking the overlay segments, the **–y** option is used to read in the symbol table of the "parent" root segment and thereby prevent multiple copies of common library functions from being present at run-time. This procedure may extend down more than one level: while linking an overlay segment, the **–w** option can be given along with the **–y** option, causing an augmented symbol file to be written containing everything defined in the read-in symbol file along with new locally defined functions. Then the overlay segment can do some overlays of its own, and so on down as many levels as is desired (or practical.) Note that the position of the **–y** option on the CLINK command line is significant; i.e, the symbol file named in the option will be searched only after any CRL files specified to the left of the **–y** option have been searched. Thus, for best results specify the **–y** option immediately after the main CRL file name. If, upon reading in the symbols from a SYM file, a symbol is found having the same name as an already defined symbol, then a message to that effect is printed on the console and the old value of the symbol is retained.
For more information on using **–y** for generating overlay segments, see the appendix on overlays.

–z Inhibits clearing of the external data area to zero during run-time initialization. If **–z** is used, then all externals come up with random values. Otherwise, externals come up all zeros.

Examples:

```
A>clink lisa -t 6000  -o joyce <cr>
```

Here, CLINK expects the file LISA.CRL to contain a **main** function, which is then linked with all functions from LISA.CRL and any needed functions from DEFF.CRL, DEFF2.CRL and, if it exists, DEFF3.CRL[8]. A statistics summary is printed out when finished, the run-time stack is set to start at 6000h and grow down (leaving memory at 6000h and above untouched by the COM file when running), and the COM file itself is to be named JOYCE.COM.

```
A>clink b:nola 6/c:liz -f kathy  -s  <cr>
```

In this example, CLINK loads all functions from NOLA.CRL (on drive B:) and LIZ.CRL (in user area 6 on drive C:), links in any needed functions from KATHY.CRL (from disk B, since the disk where NOLA.CRL was obtained is the default for this linkage), and DEFF.CRL, DEFF2.CRL and perhaps DEFF3.CRL (from the default disk/user area configured as per section 1.8.2), and prints out a statistics summary when done. Since no **–t** option is given, CLINK

8. DEFF3.CRL is automatically scanned as a user-supplied library file if it exists and there are still unresolved references after DEFF.CRL and DEFF2.CRL have been scanned. If DEFF3.CRL is not found, no complaint is lodged by the linker.

assumes all the TPA (Transient Program Area) is available for code and data. The COM file generated is named NOLA.COM by default (since no **–o** option was given) and the file is written to the currently logged in disk.

NOTE: When several files that share external variables are linked together, then the file containing the *main* function **must** contain **all** declarations of external variables used in all other files. This is because the linker obtains the size of the external area from the main source file, and this value is used to set up the appropriate parameter in the resulting COM file so that the library function *endext()* returns the correct value. Also, because external variables in BDS C are actually more like FORTRAN COMMON than UNIX C externals, the ordering of external declarations should be identical within each individual source file of a program. Typically, a single header file containing all external declarations is included by each file of a program, to insure compatibility.

1.8.7 CLIB — The C Librarian

Command format:

```
A>CLIB <cr>
```

The CLIB program is provided to let you a) transfer functions between CRL files, b) rename, delete, and inspect individual functions, c) create new CRL files, and d) inspect CRL file contents.

Before delving into CLIB operation, it is helpful to understand the structure of CRL (C ReLocatable) files:

A CRL file consists of a set of independently compiled C functions, each a binary 8080 machine code image having its origin set at 0000. Stored along with each function is a list of "relocation parameters" for use by CLINK to resolve relocatable addresses. Also stored with each function are the names of all subordinate functions called by the given function. Collectively, the code, relocation list, and needed functions list are termed a *function module.*

The first four sectors of a CRL file make up the *directory* for that file, containing a list of all function modules appearing in the file and their positions within the file. The *directory* space needed for any function in a CRL file is equal to the number of characters in that function's name, plus two bytes for an address pointer into the CRL file. Thus, the total number of functions that any given CRL file can hold is usually limited by the length of the names of those functions. The *total size* of a CRL file cannot exceed 64K bytes (because function modules are located via two byte addresses), but optimum efficiency is achieved by limiting a CRL file's size to that of a single CP/M file extent (16K bytes).

For more detailed information about CRL files, see chapter 3.

When CLIB is invoked, it will respond with an initial message and a "function buffer size" announcement. The buffer size tells you how much memory is available for intermediate storage of functions during transfers. Attempts to *transfer* or *extract* functions of greater length will fail.

Following initialization, CLIB will prompt with an asterisk (*****) and await a command.

To open a CRL file for manipulation, use

```
*open  file#  [d:]filename<cr>
```

where file# is a single digit identifier (0-9) specifying the "file number" to be associated with the file *filename* as long as that file remains open. Up to ten files, therefore, may be open simultaneously.

Note that a disk designator may be specified for the filename, allowing CLIB to operate with CRL files on any physical disk.

To close a file (making permanent any changes that were made to it), say

```
*close file# <cr>
```

The given file number then becomes free to be assigned to a new file via **open**. A backup version of the altered file is created having the name *name.BRL*. Note that the **close** operation may take some time to perform, and will cause your disk drive to thrash annoyingly when large files are involved.

It is not necessary to close a file unless either changes have been made to it or you need the extra file number. For example, a file opened just to be copied from need not be closed.

When a CRL file is opened, a copy of the file's *directory* (first 4 sectors) is loaded into memory. Any alterations made to the file (via the use of the **append**, **transfer**, **rename**, and/or **delete** commands) cause the in-core directory to be modified accordingly, but the file must be closed before the updated directory gets written back onto the disk. Thus, if you do something you later wish you hadn't, and you haven't closed the file yet, you can abort all the changes made to the file simply by making sure not to close it. Undoing **append**s and **transfer**s requires a little bit of extra work; this will be explained later.

To see a list of all open files, along with some relevant statistics on each, say

```
*files <cr>
```

To list the contents of a specific CRL file and see the length of each function therein, say

```
*list file# <cr>
```

There are several ways to move functions around between CRL files. When all files concerned have been opened, the most straightforward way to copy a function (or set of functions) is

```
*transfer source-file# destination-file# function-name <cr>
```

This copies the specified function[s] from the source file to the destination file, not deleting the original from the source file. function-name may include the special characters ***** and **?** if an ambiguous name is desired. All functions matching the ambiguous name will be transferred.

An alternative approach to shuffling files around is to use the "extract-append" method. The **extract** command has the form

```
*extract file# function-name <cr>
```

It is used to pull a single function out of the given file and place it in the function buffer (in memory). To write the function out to a file, say

```
*append file# [name] <cr>
```

where *name* is optional and should be given only to change the name under which the function is to be saved;

```
*append file# <cr>
```

is sufficient to write the function out to a file without changing its name.

Only one file# may be specified at a time with **append**; to write the function out to several CRL files, a separate **append** must be done for each file.

To rename a function within a particular CRL file, say

```
*rename file# old-name new-name <cr>
```

Note that this constitutes a change to the file, and a **close** must be done on the file to make the change permanent.

To create a new (empty) CRL file, say

```
*make filename <cr>
```

This creates a file on disk called *filename.CRL* and initializes the directory to empty. To write functions onto it, first use **open**, and then use either **transfer** orthe **extract/append** method described above. CLIB will not allow the creation a new CRL file having the same name as an existing CRL file in the same directory.

To delete a function (or set of functions) from a file, use

```
*delete file# function-name <cr>
```

Again, the function name may be specified ambiguously using the ***** and **?** characters. The file must be subsequently closed to finalize the deletion. Note that deleting a function does *not* free up any directory space in the associated CRL file until that file is actually closed. Thus if a CRL file directory is full and you wish to replace some of the functions in it, you must first delete the unneeded functions, then close and re-open the file to transfer new functions into it.

A command syntax summary may be seen by typing the command

```
*help <cr>
```

To exit CLIB and return to command level, give the command

```
*quit <cr>
```

and respond positively to the confirmation message that CLIB then prints out.

Note: All CLIB commands may be abbreviated to a single letter.

Should you decide you really didn't want to make certain changes to a file, but it is already after the fact, then the **quit** command may be used to get out of editing the file and abort any changes made. As long as you haven't appended or transferred into the file, typing

```
*quit file# <cr>
```

is sufficient to abort all operations on that file, and frees up the file# as if a **close** had been done.

If you *have* appended or transferred into a file and you wish to abort, then the **quit** command should still be used, but in addition you should re-open the file directly after quitting and then **close** it immediately. The rationale behind this procedure is as follows: when you do an **append** or a **transfer**, the function being appended gets written onto the end of the CRL file. Then, when you abort the edit, the old directory is left intact, but the appended function is still there, hanging on in the data area, even though it doesn't appear in the directory. By opening and immediately closing the file, only those functions appearing in the directory remain with the file, effectively getting rid of those "phantom" functions.

Here is a sample session of CLIB, in which the user wants to create a new CRL file named NEW.CRL on disk B: containing all the functions in DEFF.CRL beginning with the letter "p":

```
A>clib
BD Software C Librarian v1.6
Function buffer size = xxxxx bytes

*open 0 deff
*make b:new
*open 1 b:new
*transfer 0 1 p*
*close 1
*quit
(Quit) Are you sure? y

A>
```

1.9 CP/M "Submit" Files

To simplify the process of compiling and linking a C program (after the initial bugs are out and you feel reasonably confident that CC and CC2 will not find any errors in the source file), CP/M "submit" files can be easily created to perform an entire compilation. The simplest form of submit file, to simply compile, link and execute a C source program that is self contained (doesn't require other special CRL files for function linkages) would appear as follows:

```
cc $1.c
clink $1 -s
$1
```

Thus, if you want to compile a source file named, say, LIFE.C, you need only type

```
A>submit c life <cr>
```

(assuming the submit file is named *C.SUB*.)

1.10 Operational Caveats

1. When invoking any COM file in the BDS C package or any COM file generated by the compiler, your command line (as typed in to CP/M) must **never** contain any leading blanks or tabs. It seems that the CCP (console command processor) does not parse the command line in the proper manner if leading white space is introduced.

2. The **argc** and **argv** values passed to the *main* function by the BDS C run-time package will never include an entry for the command name itself (argv[0]) because CP/M does not make that information available to transient programs on start-up. The *argc* value is always positive, equal to the number of parameters passed to the command **plus one**, and argv[0] is left undefined. For any given value of **argc**, therefore, the meaningful entries in **argv** are argv[1] through argv[argc – 1]. If *argc* is equal to 1, then *argv* contains no meaningful information.

3. If the STDIO.H header file is required in a particular program, and it usually is, then it **must** be included as the very first header in every source file of the program. **It is crucial that no data declarations be placed physically before STDIO.H in any program where STDIO.H is used.**

4. If you're running MP/M II, you must re-assemble the run-time package (CCC.ASM —> C.CCC) with the "MPM2" equate set to TRUE. This makes sure that the run-time package actually closes all files opened during the course of execution of a C program, so that the system doesn't run out of file slots. Normally, under non-MPM2 systems, the BDS C run-time package does not bother to close files that were open only for reading, in order to save the time that would be required for the disk access.

1.11 Last Words

This package is no longer being supported. The following is my current (2002) contact information; feel free to contact me with feedback, but don't expect bug fixes:

Leor Zolman
BD Software
74 Marblehead Street
North Reading, MA 01864-1527
(978) 664-4178
leor@bdsoft.com

For the latest retro release information, check the BDS C section of the BD Software web site at:

http://www.bdsoft.com/resources.html#bdsc

I am grateful to the following individuals for their invaluable feedback and support over the years of BDS C's evolution:

Lauren Weinstein	Sid Maxwell
Leo Kenen	Bob Mathias
Rick Clemenzi	Bob Radcliffe
Tom Bell	The *Real* Cat
Jon Sieber	Al Mok
Scott Layson	Phillip Apley
Tony Gold	Charles F. Douds
Ed Ziemba	Robert Ward
Scott Guthery	Les Hancock
Earl T. Cohen	Ted Nelson
Sam Lipson	Ward Christensen
Dan MacLean	Jerry Pournelle
Mike Bentley	Will Colley
Carlos Christensen	Richard Greenlaw
Perry Hutchinson	Tim Pugh
Paul Gans	Steve Ward
John Nall	Tom Gibson
Mark Miller	Roger Gregory
Jason Linhart	Don Lucas
Calvin Teague	Rev. Stephen L. de Plater
Bob Shapiro	Nigel Harrison
Cal Thixton	Gary Kildall
Stu Heiss	Stefan Badsteubner
Jeff Prothero	Dan Grayson
William A. Richards	Steve Graves
Dave Roscoe	Gene Mallory
Rick Rump	John Franks

Special thanks to Dennis M. Ritchie, Ken Thompson and the entire staff of the Computing Science Research Center at Bell Laboratories for developing UNIX and the original C. Good work.

Chapter 2

The CRL Function Format and Other Low-Level Mechanisms

2.1 Introduction

This Chapter is directed toward assembly/machine language programmers who need the ability to link machine code subroutines in with normally compiled C functions. It describes the CRL format in detail, as well as the procedure for making CRL format relocatable files out of assembly language source functions. The parameter-passing and function-calling conventions used for C functions are also described, as are some convenience routines present in the run-time package relating to the function linkage, parameter passing and data-access mechanisms.

2.2 The CRL Format in Detail

Included on the standard BDS C distribution disk is a program called CASM.C, for use with Digital Research's ASM assembler under CP/M. This program allows assembly language functions to be written in a special "CSM" ("C aSseMbly") format consisting basically of ASM.COM's instruction format, plus the addition of several special pseudo-ops for the purpose of naming and delimiting individual functions. CASM.COM serves to converts the .CSM source file into an ".ASM" source file for direct assembly by ASM.COM. Another utility supplied with the package, CLOAD, is then used to convert the .HEX file output of ASM into a .CRL file suitable for linkage by either of the two BDS C linkers (CLINK and L2). A CP/M "Submit" file named CASM.SUB is provided to automate this entire procedure.

Although it is not absolutely necessary to know how a CRL file is organized in order to effectively use CASM and ASM to produce CRL files, a detailed description of the CRL format is now provided here for completeness.

2.2.1 CRL Directories

The first four sectors of a CRL file[9] make up the *CRL directory*. Each function module in the file has a corresponding entry in the directory, consisting of the module's name (up to eight

9. If you are using DDT or SID to examine the file, these sectors appear in memory locations 0100h – 02FFh.

characters, with the high-order bit set only on the last character) and a two-byte value indicating the module's byte address within the file[10].

After the last entry must be a null byte (0x80) followed by a word indicating the next available address in the file. Padding may be inserted after the end of any physical function module to make the next module's address line up on an even (say, 16 byte) boundary, but there must never be any padding **in the directory itself**.

Example: if a CRL file contains the following modules,

```
Name    Length
foo        0x137
yipee      0x2C5
blod       0x94A
```

then the directory for that file might appear as follows [11]:

```
46   4F   CF   05   02   59   49   50   45   C5
F    O    O'   nn   nn   Y    I    P    E    E'

50   03   42   4C   4F   C4   20   06   80   70   0F
nn   nn   B    L    O    D'   nn   nn   null-entry
```

2.2.2 External Data Area Origin and Size Specifications

The first five bytes of the fifth sector of a CRL file (locations 0x200-0x204 relative to the start of the file) contain information that CLINK uses to determine the origin (if specified explicitly to CC via the **–e** option) and size of the external data area for the executing program at run-time. This information is valid **only** if the CRL file containing it is treated as the "main" CRL file on the CLINK command line; otherwise, the information is not used.

The first byte of the fifth sector has the value 0xBD if the **–e** option was used during compilation to explicitly set the external data area; else, the value should be zero. The second and third bytes contain the address given as the operand to the **–e** option, if used.

The fourth and fifth bytes of the fifth sector contain the size of the external data area declared within that file (low byte first, high byte second.) CLINK always obtains the size of the external data area from these special locations within the "main" CRL file (i.e., the CRL file containing the "main" function for the program). In CRL files which do not contain a "main" function, these bytes are unused.

2.2.3 Function Modules

Each function module within a CRL file is an independent entity, containing the binary machine-code image of the function itself plus a set of relocation parameters for the function and a list of names of any other functions that it may call.

---

10. The function module addresses within a CRL file are all relative to 0x0000, with the directory residing from 0x0000 to 0x01FF. Locations 0x200 – 0x204 are reserved, so the lowest possible function module address is 0x205.

11. Note that the *last* character of each name has bit 7 set high.

A function module is *address-independent*, meaning that it can be physically moved around to any location within a CRL file (as it often must be when CLIB is used to shuffle modules around.)

The format of a function module is:

list of needed functions
length of body
body
relocation parameters

2.2.3.1 List of Needed Functions

If the function you are building calls other CRL functions, then a list of those function names must be the first item in the module. The format is simply a contiguous list of upper-case-only names, with the high-order bit (bit 7) high on the last character of each name. A zero byte terminates the list. A null list (as when the function does not call any other functions) is just a single zero byte.

For example, suppose a function *foobar* calls functions named *putchar*, *getchar*, and *setmem*. *Foobar*'s list of needed functions would appear as follows:

```
47   45   54   43   48   41   D2   50   55   54   43
g    e    t    c    h    a    r'   p    u    t    c

48   41   d2    53   45   54   4D   45   CD   00
h    a    r'    s    e    t    m    e    m'   (end)
```

2.2.3.2 Length of Body

Next comes a 2-byte word value specifying the exact length (in bytes) of the *body*, to be defined next. The length word is stored low-byte first, high-byte last.

2.2.3.3 Body

The *body* portion of a function module contains the actual 8080 code for the function, with the origin of the code always at 0000.

If the list of needed functions was null, then the code starts on the first byte of the body. If the list of needed functions specified *n* names, then a dummy jump vector table (consisting of *n* **jmp** instructions) must be provided at the start of the body, preceded by a jump instruction *around* the vector table.

For example, the beginning of the body for the hypothetical function *foobar* described above would be:

```
jmp 000Ch
jmp 0000
jmp 0000
jmp 0000
<rest of code>
```

```
C3 0C 00 C3 00 00 C3 00 00 C3 00 00 <rest of function code>.
```

2.2.3.4 Relocation Parameters

Directly following the body come the relocation parameters, a collection of addresses (relative to the start of the body) pointing to the operand fields of each instruction within the body that references a local address. CLINK takes every word being pointed to by an entry in this list, and adds to it the run-time base address of the function.

The first word in the relocation list is a count of how many relocation parameters are given in the list. Thus, if there are *n* relocation parameters, then the length of the relocation list (including the length byte) would be 2*n*+2 bytes.

For example, a function which contains four local jump instructions whose opcodes are located at, respectively, locations 0x22, 0x34, 0x4F and 0x61) would have the following relocation list:

```
04 00 23 00 35 00 50 00 62 00
```[12]

2.3 Register Allocation and Function Calling Conventions

2.3.1 The Stack

All argument passing on function invocation, as well as all local (automatic) storage allocation, take place on a single stack at run time.

2.3.1.1 The Stack Pointer

The stack pointer is kept in the SP register, and is initialized to the top of user-accessible memory area at run-time. Where the compiler thinks the end of available memory is depends on which options are given during linkage; by default, the stack pointer is initialized to the base of the CP/M BDOS, and grows down wiping out the CCP and requiring a warm-boot following program execution (to bring the CCP back into memory). If the **–t** option is used at link time, the value given as argument to **–t** is used to initialize the SP. If the **–n** option is used, then the SP is initialized to the base of the CCP, yielding 2K less stack space than the default but allowing a return to command level after execution **without** requiring a warm-boot to be performed.

2.3.1.2 How Much Space Does the Stack Take Up?

The single stack scheme has all local (automatic) data storage, formal parameters, return addresses and intermediate expression values residing on one stack. This stack begins in high memory and grows downward.

The maximum amount of space the stack can ever consume is roughly equal to the amount of local data storage active during the worst case of function nesting, plus a few hundred bytes or so (in the worst case) for miscellaneous intermediate expression values.

---

12. Note that the addresses of the instructions must be incremented by one to point to the actual address operands needing relocation.

If we call the amount of local storage in the worst case *n*, then the amount of free memory available to the user may be figured by the formula

topofmem() – endext() – (*n* + *fudge*)

where a *fudge* value of around 500 should be pretty safe. *Topofmem* and *endext* are library functions which return, respectively, a pointer to the highest memory location used by the running program (the top of the stack) and a pointer to the byte following the end of the external data area. The value of *endext()* is thus a pointer to the first byte of memory available for storage allocation and/or general purpose use.

2.3.2 External Data

External storage usually sits directly on top of the program code, leaving all of memory between the end of the external data and the high-memory stack free for storage allocation.

2.3.3 Function Entry and Exit Protocols

When a C-generated function receives control, it will usually perform the following tasks in the given order: push BC, allocate space for local data on the stack (decrement SP by the amount of local storage needed), and copy the new SP value into the BC register for use as a constant base-of-frame pointer. The reason for copying the SP into BC instead of just addressing everything relative to SP is that the SP fluctuates madly as things are pushed and popped, making variable address calculation rather confusing.

Note that the old value of BC must always be preserved for the calling routine.

Let's say the called function requires *nlocl* bytes of local stack frame space. After pushing the old BC, decrementing SP by *nlocl* and copying SP to BC (in that order), the address of any automatic variable having local offset *loffset* may be easily computed by the formula

(BC) + *loffset*

If the function takes formal parameters, then the address of the *n*th formal parameter may be obtained by

(BC) + *nlocl* + 2 + 2n

where *n* is 1 for the first value specified in the calling parameter list, 2 for the second, etc. This last formula is obtained by noting that parameters are always pushed on the stack in reverse order by the calling routine, and that pushing the arguments is the last thing done by the caller before the actual call. After the called function pushes the BC register, there will be four bytes of stuff on the stack, composed of two 16-bit values, between the current SP and the first formal parameter: a) the saved BC register and b) the return address to the calling routine. Note that this scheme requires that each formal parameter takes exactly 2 bytes of storage. Thus, single byte parameters (**char** variables) are always converted into 16-bit values (by zero-ing the high order byte, **not** sign-extending) before being passed as parameters.

Upon completing its chore (but before returning), the called function de-allocates its local storage by incrementing the SP by *nlocl*, restores the BC register pair by popping the saved BC off the stack, and returns to the caller.

The caller will then have the responsibility of restoring the SP value to that which it was before the formal parameter values were pushed; the called function can't do this because there is no way for it to determine how many parameters the caller had pushed (for example, consider the *printf* function, which takes a variable number of parameters).

Formally, the responsibilities of the calling function are:

1. Push formal parameters in reverse order (last arg first, first arg last)

2. Call the subordinate function, making sure not to have any important values in either the HL or DE registers (since the subordinate function is allowed to bash DE and may return a value in HL). The BC register may be considered "safe" from alteration by the subordinate function since, by convention, the function that is called should always preserve the BC register value that was passed to it. All functions produced by the compiler do this, as do all assembly-language-coded functions supplied in the BDS C package.

3. Upon return from the function: restore SP to the value it had before the formal parameters were pushed, taking care to preserve HL register pair (containing the returned value from the subordinate function). The simplest way to restore the stack pointer is just to do a **pop d** for each argument that was pushed.

The protocol required of the called, subordinate function is:

1. Push the BC register if there is any chance it may be altered before returning to the caller.

2. If there are any local storage requirements, allocate the appropriate space on the stack by decrementing SP by the number of bytes needed.

3. If desired, copy the new value of SP into the BC register pair to use as a base-of-frame pointer. Don't do this if BC wasn't saved in step 1!

4. Perform the required computing.

5. When finished, de-allocate local storage by incrementing SP by the local frame size.

6. Pop old BC from the stack (if saved in step 1).

7. Return to caller with the returned value (if any) in the HL register.

2.4 Re-entrant Coding

Since BDS C was not explicitly designed with re-entrancy of functions in mind, special steps must be taken when this feature is desired. The most common application of re-entrant code with BDS C is for the implementation of interrupt service routines in C programs.

The problem with using C functions to perform the interrupt handling is that there are several mechanisms in the run-time package which maintain *state variables* within the run-time package scratch pad RAM area. Specifically, the multiplicative arithmetic operators (multiply, divide and mod) were optimized for speed and therefore do not bother to preserve the previous contents of their scratch pad variables on the stack. To allow C programs to be re-entrant, the run-time package must be modified so that the multiplicative operators take the appropriate re-entrancy precautions. This can only be accomplished by patching each multiplicative routines to push its local data before computation, and then restore it when finished. This **must** be done without altering the starting address of any routine in the run-time package occurring before the *init:* routine; i.e., patching is required.

2.5 Helpful Run-Time Subroutines Available in C.CCC (See CCC.ASM)

There are several useful subroutines in the run-time package available for use by assembly language functions. The routines fall into three general categories: the local-and-external-fetches, the formal-parameter fetches, and the arithmetic and logical routines.

2.5.1 Local and External Fetch Routines

The first group of six subroutines may be used for fetching either an 8- or 16-bit object, stored at some given offset from either the BC register or the beginning of the external data area, where the offset is specified as either an 8- or 16-bit value. For example: the intuitive procedure for fetching the 16-bit value of the external variable stored at an offset of *eoffset* bytes from the base of the external data area (the pointer to which is stored at location *extrns*) would be

```
lhld    extrns          ;get base of external area into HL
lxi     d,eoffset       ;load offset into DE
dad     d               ;add offset to base pointer
mov     a,m             ;perform 4-step
inx     h               ;  indirection to
mov     h,m             ;    fetch value at
mov     l,a             ;       (HL) into HL.
```

Using the special call for retrieving an external variable, the same result may be accomplished with

```
call    sdei            ; 8-bit offset, 16-bit value external
db      eoffset         ;   indirection, with eoffset < 256
```

The second sequence takes up much less memory; 4 bytes versus 11, to be exact. If the value of *eoffset* were greater than 255, then the ldei routine would be used instead, with *eoffset* taking a dw instead of a db to represent. See the CCC.ASM file for complete listings and documentation on the entire repertoire of these value-fetching subroutines.

2.5.2 Formal Parameter Fetches

The second class of subroutines are used primarily for fetching the value of a function argument off the stack into the HL and A registers (the low order byte is placed in both the A and L registers, while the high byte is placed only in the H register). For example: say your assembly function has just been called; a call to the subroutine *ma1toh* would fetch the first argument into HL and A. *ma1toh* (mnemonic for "Move Argument 1 TO H") always fetches the 16-bit value present at location SP+2 (as your function sees the SP.) A call to the *ma2toh* ("Move Argument 2 to H") routine would retrieve the second 16-bit argument off the stack in HL and A. If you push the BC register before fetching a parameter off the stack, then all items on the stack will be offset by another 2 bytes from the SP value and you'd have to call *ma2toh* in order to fetch the *first* argument, *ma3toh* to fetch the second, and so on. Thus, it is important to keep track of stack depth when using these subroutines.

A less confusing way to deal with function arguments is to call the routine called *arghak* as the very first thing you do in your function, **especially** before pushing BC or anything else on the stack. *Arghak* copies the first seven function arguments off the stack to a 14-byte buffer in the r/w memory area (normally within C.CCC itself), making those values accessible via simple **lhld** operations for the duration of the function's operation…that is, assuming your function doesn't call another function which also uses *arghak* to copy **its** arguments down there, overwriting those of the calling function. After *arghak* has been called, the first argument will be stored at absolute location *arg1*, the second at *arg2*, etc. These symbols are defined in BDS.LIB, as described below.

2.5.3 Arithmetic and Logical Subroutines

The final category of subroutines is the arithmetic and logical group, all of which take arguments passed in HL and DE and return a result in HL. I won't take up space with details on these functions here; examine the run-time package source file (CCC.ASM) to see the subroutines that are available.

2.5.4 System Source Files

The source code to the various modules which make up an integrated compilation environment can be thought of as broken up into four separate categories:

1. The C source code to the application program to be compiled, plus any related CSM ("C aSseMbly language")-formatted assembly source code, are collectively the "source program";

2. STDLIB1.C, STDLIB2.C and STDLIB3.C (#including STDIO.H) contain the source code to all C-coded portions of the standard BDS C library (compiled into DEFF.CRL);

3. DEFF2A.CSM, DEFF2B.CSM and DEFF2C.CSM (all #including BDS.LIB) contain the source code to all CSM-coded portions of the standard BDS C library (assembled into DEFF2.CRL, with a few functions placed into DEFF.CRL to eliminate backward-references during linkage)

4. CCC.ASM contains the source code to the BDS C run-time package module, assembled into C.CCC.

The run-time package source file, CCC.ASM, contains the code and documentation of all the helpful run-time subroutines described above. The header file BDS.LIB contains definitions of all entry points to the routines within C.CCC (assembled from CCC.ASM) as provided in the distribution version of the package. All CSM-format source files should contain the directive

```
#include <bds.lib>
```

so that the necessary subroutines may be referred to directly by name in CSM modules. If you need to modify CCC.ASM in order to customize the run-time package, be sure to also modify BDS.LIB to reflect the new addresses, and check to make sure all named symbols assemble to equal values in both CCC.ASM and BDS.LIB. For instructions on generating code for placement into ROM, execution at arbitrary locations in memory, and with or without CP/M in residence, see the appendix entitled "Customized Run-Time Environments".

2.6 Debugging Object Command Files Under CP/M

There are two general approaches to interactive debugging of an object file created using BDS C. The first is simply to use the CDB symbolic debugger provided with the package, as described in the appendix devoted to CDB. CDB allows symbolic references to all functions and variables in a program, making the tracing of execution fairly straightforward. But, problems may arise if your object program is too big to fit in memory together with the large CDB module.

The other way to debug a program involves the use of SID.COM, the digital Research symbolic debugger, or any symbolic debugger that accepts a standard ".SYM" symbol table file as written out by CLINK.COM (when the **–w** option is used), L2.COM (-sym) or MAC.COM ($pn). A .SYM file contains the names and starting addresses of each function in an object program. When SID is invoked with a command file and its companion .SYM file as arguments, then the starting address of each function in the command file may be referred to directly by name under SID. If SID.COM is not available, then DDT.COM may be used instead provided a printed .SYM file is available for visual cross-reference of symbol values.

2.6.1 Loading Programs and Setting Breakpoints

To debug a .COM file using SID, begin by invoking SID in the following manner:

```
sid <filename>.com <filename>.sym
```

This will load up the target program and its associated .SYM file. Next, enter the command line that the target program will see upon startup by using the SID command **i**:

```
-iarg1 arg2 arg3...
```

Now, debugging may commence through careful setting of breakpoints at key function entry points. For example, to begin execution at the start of the target program (run-time package initialization) and stop as soon as the MAIN function is reached, use the command:

```
-g,.main
```

As soon as execution stops at the start of any particular function, it is possible to look at the arguments passed to that function by analyzing the memory locations pointed to by the SP register. In general, the following command may be entered as soon as execution has stopped at a breakpoint set at the start of a function, where <sp> is the value contained in the SP register:

```
-d<sp>,+8
```

SID will respond with a dump of the form:

```
<sp>:  nn nn 1l 1h 2l 2h 3l 3h ...
```

where <sp> is the SP value you typed in to the **d** command, "nn nn" is the return address to the calling function, "1l 1h" are the low-order and high-order bytes, respectively, of the first function parameter passed to the current function, "2l 2h" is the second parameter, and so forth. If this were being done for the MAIN function, then "nn nn" would be the return address to the place in the run-time package from which MAIN was called, "1l" would be the value of *argc*, "1h" would be zero (the high-order byte of argc), and "2l 2h" would be the address of the *argv* vector table. To actually see the text of the command line parameters, you'd then dump memory at location "2l 2h" (by reversing the two bytes and entering the 4-digit address as the parameter to the **d** command), yielding a sequential list of 16-bit pointers to the actual command line parameters. You'd then dump the location of each parameter in turn to see the actual text.

Another useful tool when debugging with SID is the command "**g,**^", or "return to caller". For example, if you wish to see exactly what the *printf* function produces at the very next time it is called, first you'd set a breakpoint at the start of *printf* by saying:

```
-g,.printf
```

when execution halts at the start of printf, you'd then enter the command:

```
-g,^
```

as soon as you hit return, the *printf* function will take off, and execution will halt upon return to the memory location following the call to *printf*. This command, "g,^", says to SID that execution should continue until the address that is currently at the top of the stack is reached.

Since the address at the top of the stack upon entry to the *printf* function is the address of the instruction following the **call** instruction used to call *printf*, that is where SID will next halt execution. Note that DDT does not recognize this shorthand; to perform the same operation under DDT, you must dump the memory location pointed to by SP and use the **g,nnnn** command to explicitly set a breakpoint at the memory location indicated by the top value on the stack.

2.6.2 Tracing Execution and Dumping the Values of Variables

If a C command is to be debugged using SID/DDT, then it should be compiled in a special manner in order to make debugging as straightforward as possible. First of all, the CC option **–e xxxx** should be used to fix the external data area at some absolute memory location. This will shorten/simplify the code generated to access external variables, as the LHLD and SHLD ops can be used for this purpose by the compiler. Secondly, the CC option **–o** should also be used, to eliminate the space-saving calls to the run-time package for the purpose of loading the addresses of automatic data. Since the form of the space-saving calls is

```
call <routine_name>
db nn (or) dw nn
```

it becomes confusing to try and trace these calls, with the in-line data bytes immediately following the call instructions creating confusion for the debugger. By specifying **–o**, all local variable addresses are computed in-line, eliminating sequences such as the one above, and it then becomes easier to follow execution with SID or DDT.

There are no symbolic references available for variable names, so it becomes necessary to compute the absolute memory addresses of external variables, and the relative offset values of local variables, by hand.

Local (automatic) data is stored, in the order declared, immediately following the end of formal parameter storage on the stack. At the start of each C-generated function, the address of the automatic local variable storage area is copied into the BC register and left there untouched for the duration of that function's execution. Therefore, after the initial sequence, the BC register always points to the start of the first automatic variable belonging to the currently executing function. It makes debugging easier if you always declare the variable you'll need to watch the most first, and then declare other automatic variables.

External data is stored sequentially beginning at the address specified as argument to the CC option **–e**. Again, it makes life easier for you to declare the externals you need to watch the most as the first things in the external data area. You may want to insert some *printf* statements at the start of the program just to print out the addresses of the external data objects you need to know the locations of. This will eliminate the need to recompute the addresses by hand each time you may change the location of the external data area, or the order of the items declared therein.

2.6.3 A Sample SID Debugging Session

Here is a sample session using SID and a trivial C program called TEST.C, which prints out the command line parameters on the console:

```
---------------------------------------------------------------------
A>type test.c

/*
        TEST.C: Echo command line arguments
*/

#include <stdio.h>

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

        for (i = 1; i < argc; i++)
                printf("Arg #%d = %s\n",argv[i]);
}

A>cc test.c
BD Software C Compiler v1.60
xxK elbowroom
BD Software C Compiler v1.60
xxK left over

A>clink test -w                 ;-w option causes TEST.SYM to be written
BD Software C Linker   v1.60
.
.  (link statistics printed here)
.

A>sid test.com test.sym         ;invoke SID with object file and symbol file
SID VERS 1.4
SYMBOLS
NEXT  PC  END
0F80 0100 9AB1                  ;(exact numbers may vary with release version)
#ithis is a test                ;fill CP/M command line buffer
#l                              ;disassemble start of program
   0100  LHLD 0006
   0103  SPHL
   0104  NOP
   0105  NOP
   0106  JMP  010C
   0109  JMP  0000
   010C  CALL 0362
   010F  CALL 08B8 .MAIN
   0112  JMP  047B
   0115  STAX D
   0116  RRC

#g,.main                        ;execute until MAIN is entered

*08B8 .MAIN

#x                              ;display registers
```

```
  -Z-E- A=00 B=0084 D=0F12 H=010F S=9AAC P=08B8 JMP  08BE

#d9aac,+5                          ;look at stack upon entry to MAIN function
9AAC: 12 01 05 00 ....             ; 0112 = return addr, 0005 = argc, 087A = argv
9AB0: 7A 08 z.

#d87a,+f                           ;look at argv vectors
087A: 32 D0 F9 07 FE 07 2.....     ;argv[0] = D032, argv[1] = 07F9, etc.
0880: 01 08 03 08 A5 29 CA E8 28 CD .....)..(.

#d7f9,+5                           ;dump argv[1]
07F9: 74 68 69 73 00 69 this.i     ; t h i s <null>

#d7fe,+5                           ;dump argv[2]
07FE: 69 73 is                     ; i s <null>
0800: 00 61 00 74 .a.t

#d801,+5                           ;dump argv[3]
0801: 61 00 74 65 73 74 a.test     ; a <null>

#d803,+5                           ;dump argv[4]
0803: 74 65 73 74 00 5E test.^     ; t e s t <null>

#g,.printf                         ;execute until PRINTF is entered first time

*0924 .PRINTF

#x                                 ;display registers upon entry to PRINTF
  --M-- A=01 B=9AA8 D=08F6 H=0916 S=9AA0 P=0924 JMP  092D

#d9aa0,+f                          ;display stack upon entry to PRINTF
9AA0: FE 08 16 09 01 00 F9 07 01 00 84 00 12 01 05 00 ................

#d916,+f                           ;look at first argument (8FE is return addr)
0916: 41 72 67 20 23 25 64 20 3D 20 Arg #%d =
0920: 25 73 0A 00 C3 2D %s...-     ;string is "Arg #%d = %s\n"

#d7f9,+5                           ;second arg is 0001, third arg is this:
07F9: 74 68 69 73 00 69 this.i     ; t h i s <null>

#g,^                               ;continue execution until return from PRINTF
Arg #1 = this                      ;this is printed out by PRINTF

*08FE                              ;execution halts at 8FE, after call to PRINTF

#t                                 ;trace one instruction
  -Z-E- A=00 B=9AA8 D=0EE3 H=0000 S=9AA2 P=08FE POP  D
*08FF

#g,.printf                         ;continue executing until next entry to PRINTF

*0924 .PRINTF

#g,^                               ;continue executing until return from PRINTF
```

```
Arg #2 = is                         ;PRINTF prints this

*08FE

#t
  -Z-E- A=00 B=9AA8 D=0EE3 H=0000 S=9AA2 P=08FE POP  D
*08FF

#g,.printf                          ;do it again

*0924 .PRINTF

#g,^
Arg #3 = a

*08FE

#t
  -Z-E- A=00 B=9AA8 D=0EE3 H=0000 S=9AA2 P=08FE POP  D
*08FF

#g,.printf                          ;and again

*0924 .PRINTF
#g,^
Arg #4 = test

*08FE

#x                                  ;look at registers
  -Z-E- A=00 B=9AA8 D=0EE3 H=0000 S=9AA2 P=08FE POP  D

#l100                               ;disassemble start of program again
    0100  LHLD 0006
    0103  SPHL
    0104  NOP
    0105  NOP
    0106  JMP  010C
    0109  JMP  0000
    010C  CALL 0362
    010F  CALL 08B8 .MAIN
    0112  JMP  047B
    0115  STAX D
    0116  RRC

#g,112                              ;continue until return from MAIN

*0112                               ;this is it (nothing printed by program)

#g                                  ;now continue with run-time cleanup

A>                                  ;and we're back at command level
```

Chapter 3

The BDS C Standard Library on CP/M: A Function Summary

In the BDS C package, the files DEFF.CRL and DEFF2.CRL contain the object code of the standard library. DEFF.CRL contains the compiled object code for all the C-coded functions from STDLIB1.C, STDLIB2.C and STDLIB3.C, and DEFF2.CRL contains all the object code for the assembly language functions from DEFF2A.CSM, DEFF2B.CSM and DEFF2C.CSM (assembled using the CASM facility). CLINK automatically searches these .CRL library files[13] **after** all other CRL files explicitly named on the command line have been searched. Thus, any functions you explicitly define in a source file that happen to have the same name as library functions will **take precedence** over the library versions, as long as CLINK finds your version of the function before getting around to scanning the library.

In the following summary of all the major functions in DEFF.CRL and DEFF2.CRL, each function is described both in words and in a loose C-like notation intended to illustrate how a **definition** of that function might appear in a C program. Such notation provides, at a glance, information such as whether or not the function returns a value (and if so, of what type) and the types of any parameters that the function may take. Here are some rules of thumb: if a function is listed without a type, then it doesn't return a value (for example, *exit* and *poke* return no values.) Any formal parameters lacking an explicit declaration are implicitly of type **int**, although in many cases only the low-order 8 bits of the parameter are used and a value of type **char** may be passed to the function. Note that it isn't always easy to describe the exact type of a formal parameter…is a memory pointer of type **unsigned**, or is it a character pointer? As long as you don't try to pass a **char** variable in the position of a 16-bit memory address parameter, things will probably work right no matter what the declared type of the parameter is in the calling program.

There are only a few cases where it is actually necessary to *declare* a library function before it is used in a C program. One case is when the function returns a value having a type other than **int**, and the function call is placed inside an expression where the type of the return value needs to be other than **int** in order for the expression to work (as in pointer arithmetic, for example.) A bit of experience will help to clarify when it is proper or unnecessary to declare certain functions; many of these decisions are a matter of style and/or portability.

Here is a summary of all major functions available in DEFF.CRL and DEFF2.CRL:

3.1 General Purpose Functions

13. If desired, the user may configure CLINK to search for the library files in an arbitrary CP/M disk drive and user area, allowing linkages to be performed in any drive and user area without needing to have all the library files there also.

char csw()

Returns the byte value (0-255) of the console switch register (port 0xFF on some mainframes).

exit()

Closes any open files and exits from an executing program, re-booting CP/M. Does **not** automatically call *fflush* on files opened for buffered output.

int bdos(c,de)

Calls the standard BDOS system entry point (location 0005h on most systems), first setting CPU register C to the value *c*, and register pair DE to the value *de*.
Return value is the 16-bit value returned by the BDOS in HL. For CP/M systems, the low-order byte is the value returned by the BDOS in A, and the high-order byte is the value returned by the BDOS in B (or zero for 8-bit return values.) See the "Miscellaneous Notes" appendix for some details on incompatibilities with non-CP/M systems (e.g., SDOS).

char bios(n,c)

Calls the *n*th entry in the BIOS jump vector table, where n is 0 for the first entry (BOOT), 1 for the second (WBOOT), 2 for the third (CONST), etc., first setting CPU registers BC to the value *c*.
Result is the value returned in register A by the BIOS call.
Note that the cold-boot function (where n is 0) should never actually be used, since the CCP will be bashed and probably crash the system upon entry.
There are some BIOS calls that require a parameter to be passed in DE, and that return their result in HL. Use the *biosh* function (described next) for those calls.
WARNING: Both the *bios* and *biosh* functions assume that the instruction at the start of the base page of CP/M on your system is a **jmp** directly to the "warm-boot" entry point of your BIOS jump vector table. If this is not the case (e.g. on the Xerox 820), you must modify these functions (in CSM format) to correctly compute the address of the BIOS jump vector table in whichever manner is appropriate for your system, then re-install the new versions of *bios* and *biosh* in the DEFF2.CRL library file.

unsigned biosh(n,bc,de)

Calls the *n*th entry in the BIOS jump vector table, as above, first setting CPU registers BC to the value *bc* and setting CPU registers DE to the value *de*. Result is the value returned in registers HL by the BIOS call.
See the WARNING above, about how this function computes the location of the BIOS jump vector table on your system.

char peek(n)

Returns contents of memory location *n*. Note that in applications where many consecutive locations need to be examined, it is more efficient to use indirection on a character pointer than it is to use *peek*.
This function is provided for the occasional instance when it would be cumbersome to declare a pointer, assign an address to it, and use indirection just to access, say, a single memory location.

poke(n,b)

Deposits the low-order eight bits of *b* into memory location *n*. This can also be more efficiently accomplished using pointers, as in

```
*n = b;
```

(where n is a pointer to characters.)

char inp(n)

Returns the eight-bit value present at input port *n*.
Both the *inp* and *outp* functions perform port I/O by constructing a three-byte subroutine sequence (consisting of a two-byte I/O operation and a single-byte return operation) in the run-time package data area, and then calling it as a subroutine to actually perform the I/O. This limits the port number range to 8 bits, since the 8080 "in" and "out" ops are used for general compatibility. Adapting *inp* and *outp* for 16-bit port number operation can be accomplished without too much difficulty by modifying the functions to execute Z80 input/output operations, and then reassembling them using the CASM processor and associated commands.
For memory-mapped input, use the *peek* function.

outp(n,b)

Outputs the eight-bit value *b* to output port *n*.

See *inp* above for a note about 16-bit port addressing.

For memory-mapped output, use the *poke* function.

pause()

Sits in a loop until CP/M console input interrogation indicates that a character has been typed on the system console. The character itself is **not** sampled; before *pause* can be used again, a *getchar* call must be made to clear the status. There is no return value.

sleep(n)

Sleeps (idles) for n/20 seconds at 4 MHz, or n/10 seconds at 2 MHz. The only way to abort out of this before completion is to type control-C, which aborts the program and returns to command level. There is no return value.

int call(addr,a,h,b,d)

Calls a machine code subroutine at location *addr*, setting CPU registers as follows:

```
HL <-- h;
A  <-- a;
BC <-- b;
DE <-- d
```

Return value is whatever the subroutine returns in registers HL. The subroutine must, of course, maintain stack discipline.

char calla(addr,a,h,b,d)

Just like the *call* function, except the result is the value returned by the subroutine in register A (instead of HL.)

int abs(n)

Returns absolute value of *n*.

int max(n1,n2)

Returns the greater of two integer values.

int min(n1,n2)

Returns the lesser of two integer values.

srand(n)

If *n* is non-zero, this function
initializes the pseudo-random number generator by setting the internal
seed to the value *n*.
If *n* is zero, then
srand prints a message asking the user to type a carriage return,
then begins to count very fast internally.
When a key is finally hit by the user,
the current value of the
count is used to initialize the random seed. The character typed by the
user is gobbled up (lost), and status is cleared.

srand1(string)
char *string;

Like *srand(0)*, except that instead of the
canned "Hit return after a few seconds:" message,
the provided string is used as a prompt.
Unlike *srand*, though, the character typed by the user in
response to the prompt is **not**
gobbled up; you must do a *getchar* call to sample the character
and/or clear the console status.

int rand()

Returns next value (ranging: 0 < rand() < 32768) in a pseudo-random
number sequence initialized by *srand* or *srand1*.
To get a value between 0 and n-1 inclusive, use the subexpression:

```
rand() % n
```

nrand(-1,s1,s2,s3)
nrand(0, prompt-string)
int nrand(1)

A "better quality" random number generator.
The first form sets the internal 48-bit seed equal to the 48 bits of data specified by s1, s2 and s3 (ints or unsigneds.)
The second form acts just like the *srand1* function: the string pointed to by *prompt-string* is printed on the console, and then the machine
waits for the user to type a character while constantly incrementing an internal 16-bit counter. As soon as a character is typed, the value of the counter is plastered throughout the 48-bit seed. Note that the console input is **not** cleared; a subsequent *getchar* call is required to actually sample the character typed and clear the console status.
The final form simply returns the next value in the random sequence, with the range being

```
0 < nrand(1) < 32768.
```

Note that the internal seed maintained by *nrand* is separate from the seed used by *srand*, *srand1* and *rand*, which use the first 32 bits of the area labeled *rseed* within the run-time package data area. *Nrand* maintains its own distinct internal seed.

setmem(addr,count,byte)
char byte, *addr;

Sets *count* contiguous bytes of memory beginning at *addr* to the value *byte*. This is efficient for quick initialization of arrays and buffer areas.

movmem(source,dest,count)
char *source, *dest;

Moves a block of memory *count* bytes in length from *source* to *dest*. This function will handle any configuration of source and destination areas correctly, knowing automatically whether to perform the block move head-to-head or tail-to-tail. If run on a Z80 processor, the Z80 "block move" instructions are used. If run on an 8080 or 8085, the normal 8080 ops are used. This all happens automatically.

memcmp(block1, block2, length)
char *block1, *block2;

Does a quick comparison of two blocks of memory having size *length*, returning 1 (TRUE) if the blocks are exactly similar or 0 (FALSE) if there are any differences.

qsort(base,nel,width,compar)
char *base;
int (*compar)();

Does a "shell sort" on the data starting at *base*, consisting of *nel* elements each *width* bytes in length. *compar* must be a pointer to a function of two pointer arguments (e.g. x,y) which returns

```
 1  if *x > *y
-1  if *x < *y
 0  if *x == *y.
```

Elements are sorted in ascending order.

int exec(prog)
char *prog;

Chains to (loads and executes) the program *prog*.COM. *Prog* must be a null-terminated string pointer specifying the file to be chained (the ".COM" need not be present in the name). A string constant (such as "foo") is perfectly reasonable, since it evaluates to a pointer. If the program to be *exec*ed was generated by the C compiler and it needs to share external variables with the *exec*ing program, then it should have been linked with the CLINK option **–e** to locate common external data at the same address. See the CLINK documentation for details on the proper usage of the **–e** option. There may be **no** transfer of open file ownership through an *exec* call. The only possible shared resource under this scheme is external data as described above. Returns –1 on error…but then, if it returns at all there must have been an error.

int execl(prog,arg1,arg2,…,0)
char *prog, *arg1, *arg2, …

Allows chaining from one C COM file to another with parameter passing through the **argc** & **argv** mechanism. *Prog* must be a null-terminated string pointing to the name of the COM file to be chained (the ".COM" need not be present in the name), and each argument must also be a null-terminated string. The last argument **must be zero**. *Execl* works by creating a command line out of the given parameters, and proceeding just as if the user had typed that command line in to the command processor of CP/M. For example,

```
execl("foo", "bar", "zot", 0);
```

would have the same effect as if the CP/M command line

```
A>foo bar zot <cr>
```

were directly typed. Unfortunately, the built-in CP/M commands (such as "dir", "era", etc.) cannot be invoked with *execl*. The total length of the command line constructed from the given argument strings must not exceed approximately 80 characters. If the constructed command line exceeds this length, a message to that effect will be printed on the console and the program will abort.
–1 returned on error (again, though, if it returns at all then there must have been an error.)

execv(filename,argvector)
char *filename;
char *argvector[];

This function allows chaining with a variable number of arguments to be performed, similarly to *execl*, except that the parameter text is specified in an array instead of in the calling sequence explicitly. The *argvector* parameter must be a pointer to an array of string pointers, where each string pointer points to the next argument and the last pointer **has a value of zero** (as opposed to being a pointer to a null string.)
Returns –1 on error, though any return at all implies an error.

int swapin(filename,addr)
char *filename;

Loads in the file whose name is the null-terminated string pointed to by *filename* into location *addr* in memory. No check is made to see if the file is too long for memory; be careful where you load it! This function may be used, for example, to load in an overlay segment for later execution via an indirection on a pointer-to-function variable.
Returns –1 if there is an error in reading in the file. Control is **not** transferred to the loaded file.

char *codend()

Returns a pointer to the first byte following the end of root segment program code. This will normally be the beginning of the external data area unless the CLINK option **–e** is used to explicitly locate the external data (see the *externs* function below.)

char *externs()

Returns a pointer to the start of the external data area. Unless the **–e** option was used with CC and/or with CLINK, this value will be the same as that returned by the *codend* function.

char *endext()

Returns a pointer to the first byte following the end of the external data area. This is start of the area from which the *sbrk* function obtains free memory.

char *topofmem()

Returns a pointer to the last byte of the user memory. This is normally the top of the stack, which is either immediately below the BDOS (if the **–n** option is not given to CLINK at linkage time) or immediately below the CCP (if **–n** *is* used at linkage time).
The value returned by *topofmem* is **not** affected by use of the **–t** option at linkage time.

char *alloc(n)

Returns a pointer to a free block of memory *n* bytes in length, or 0 if *n* bytes of memory are not available. This is roughly the storage allocation function from chapter 8 of Kernighan & Ritchie, simplified due to the lack of type-alignment restrictions. See the book for details. The standard header file STDIO.H must be **#include**d in **all files** of a program that uses *alloc* and *free* pair, since external data used by *alloc* and *free* is declared therein. The external variable _allocp, defined in STDIO.H, may be set to NULL in order to reset the *alloc/free* storage allocation mechanism. In order to fully re-initialize system storage allocation, though, it is also necessary to reset the low-level sequential storage allocator; this is accomplished by calling the *sbrk* function with an argument of –1. See the *sbrk* section below.

free(allocptr)
char *allocptr;

Frees up a block of storage allocated by the *alloc* function, where *allocptr* is a value obtained by a previous call to *alloc*. *Free* need not be called in the reverse order of previous *alloc* calls, since the linked-list data structure can tolerate any order of allocation/de-allocation. Never call *free* with an argument not previously obtained by a call to *alloc*.

char *sbrk(n)

This is the low-level storage allocation function, used by *alloc* to obtain raw memory storage. It returns a pointer to *n* bytes of memory, or –1 if *n* bytes aren't available. The first call to *sbrk* returns a pointer to the location in memory immediately following the end of the external data area; each subsequent call returns a block contiguous with the last, until *sbrk* detects that the locations being allocated are getting dangerously close to the current stack pointer value. By default, "dangerously close" is defined as 1000 bytes. To alter this default, see the next function.
If you plan to use the *alloc* and *free* functions in a program, but would also like some memory immune from allocation to be available for scratch space, use *sbrk()* to request the desired memory instead of *alloc*. *Sbrk* calls may be made at any time (independent of any *alloc* and *free* calls that may have been made).
If *sbrk* is called with *n* equal to –1, this is a special case that causes a complete reset of the internal memory allocation pointer to its initial value (a pointer to the memory location following the last byte of the external data area).

rsvstk(n)

This function causes the storage allocation functions to reject any allocation calls which would leave less than *n* bytes between the end of the allocated area and the current value of the stack pointer (remember that the stack grows down from high memory.) *Rsvstk*, if needed, should be called **before** any calls are made to either *sbrk* or *alloc*.
If *rsvstk* is never used, then storage allocation is automatically prevented from approaching closer than 1000 bytes to the stack (just as if an "rsvstk(1000)" call had been made).

int setjmp(buffer)
char buffer[JBUFSIZE];

longjmp(buffer, val)
char buffer[JBUFSIZE];

When *setjmp* is called, the current processor state is saved in the provided buffer (the symbolic constant JBUFSIZE is defined in STDIO.H) and a value of 0 is returned. When a subsequent *longjmp* call is performed from anywhere in either the current or any lower level function, then the CPU state is restored to that which it had at the time the original *setjmp* call was performed with the given buffer as parameter. The program resumes execution by "returning" to the original *setjmp* call, and the value *val* (as passed to *longjmp*) is returned. To allow programs to distinguish between *setjmp* initialization calls and transfers of control, the value of *val* passed to *longjmp* should be non-zero. A typical use of *setjmp*/*longjmp* is to exit up through several levels of function nesting without having to return through each level in sequence; e.g., to insure that a particular exit routine (say, *dioflush* from the DIO.C package) is always performed.

3.2 Character Input/Output

The console I/O mechanism for BDS C v1.6 provides a built-in ability to dynamically choose whether or not certain special characters are detected and processed during routine low-level input and output calls. In previous releases of BDS C, the standard versions the of *getchar* and *putchar* functions always detected Control-C being typed on the console input and caused an immediate return to CP/M command level when this happened. For v1.6, a new function named *iobreak* has been added to control this interrupt detection mechanism. As described below, calling *iobreak* with an argument of 0 will disable detection of Control-C during console I/O calls. This prevents the end user from inadvertently (or purposely!) aborting the execution of a program by typing Control-C during console I/O. Note that a side-effect of calling *iobreak(0)* is that the use of ^S/^Q for flow control during console output calls is also disabled. By default, a program will come up with Control-C detection enabled (as if *iobreak(1)* had been called) for compatibility with earlier source code.

Another new feature of the low-level console I/O for v1.6 is the option of selecting between two different modes of console input: "line buffered" mode or "normal single character" mode. This choice is made through use of the new *cmode* function. In line buffered mode, console input is always collected a line at a time (i.e., until the user either types a RETURN or runs out of internal line buffer space), then doled out a byte at a time for each subsequent *getchar* call. This mechanism allows the user to line-edit his input text before it is recognized by the program; also, the usage of functions such as *scanf* is made more powerful by the fact that a single line of console input may be processed by several separate *scanf* calls (with each subsequent *scanf* call picking up from where the previous one left off processing the input stream).

While substantially more flexible than the *getchar/putchar* mechanism of previous releases, there are still some things that the standard functions provided here cannot do by themselves.

For example, it may be desirable to alloc ^S/^Q flow control on console output while still not allowing the option of terminating a program by typing Control-C. To achieve this type of subtle control over the console I/O mechanism, you must create your own customized versions of *getchar/putchar* using the techniques described in the Appendix entitled "BDS C Console I/O: Some Tricks and Clarifications".

int cmode(n)

Chooses between either line buffered console input mode (if n is 1) or single-character console input mode (if n is 0). Value returned is the previous value of the character mode (1 or 0).
Default setting on start-up is *cmode(0)*.
Calling *cmode*, regardless of the mode selected, clears both the line input buffer and the single-character push-back buffer (used by *ungetch*).

iobreak(n)

As described in the introduction to this section, *iobreak* selects whether or not Control-C is allowed to terminate program execution during console I/O and return control to CP/M command level. If *n* is 0, then Control-C (as well as ^S/^Q flow control) is ignored. If *n* is 1, then Control-C and ^S/^Q are recognized. Note that in buffered input mode (when *cmode(1)* has been called), Control-C typed in at any position other than as the **first** character of the input line will **not** abort the program until it is actually sampled by a subsequent *getchar()* call. This is because the input buffering is done via a call to the operating system, and that is just how the BDOS does things.

int getchar()

Returns next character from standard input stream (CP/M console input.)
Console input is either buffered up a line at a time or returned character by character, depending on how the *cmode* function has been used. If *cmode* has not been called, then the default mode is character by character.
If *iobreak(1)* has been used, then *getchar* detecting Control-C on the console input
causes the immediate termination of the executing program and the return of control to command level.
A "Carriage return" (CR, or RETURN) echoes CR-LF to the console output and returns a newline ('\n', or LF) character.
A value of –1 is returned for control-Z; note that the return value from *getchar* must be treated as an integer (as opposed to a character) if the –1 return value is to be recognized as such.
If instead you declare *getchar* as returning a character value, or assign its return value to a character variable, then the original –1 value will turn into a value of 255.
Note that in this case an actual data value of 255 would be indiscernible from an EOF marker.

char ungetch(c)

Causes the character *c* to be returned by the next call to *getchar*.
Only one character may be "ungotten" between consecutive *getchar* calls.
Normally, zero is returned. If there was already a character ungotten since the last *getchar* call, then the value of that character is returned.

int kbhit()

Returns true (non-zero) if input is present at the standard input (keyboard character hit);
else returns false (zero).
In no case is the input actually sampled; to do so requires a subsequent *getchar* call.
Note that *kbhit* will also return true if the *ungetch* function was used to push back a character to the console since the last *getchar* call, or if console input mode is line buffered and there are characters remaining in the buffer.

putchar(c)
char c;

Writes the character *c* to the standard output (CP/M console output).
The newline ('\n') character is expanded into a CR-LF combination on output.
Unless *iobreak(0)* has been called, a control-C detected on console **input** during a *putchar* call will cause program execution to halt and control to return to command level. This allows the end-user to abort any program in the process of performing console output (via *putchar* calls) by typing a control-C on the console keyboard.
Since the provided *putchar* function uses BDOS calls to check for input at the console (unless *iobreak(0)* has been called), the special CP/M flow-control characters (control-S, control-Q) are recognized and may be used to freeze/unfreeze console output.

puts(str)
char *str;

Writes out the null-terminated string *str* to the standard output. No automatic newline is appended.

int getline(strbuf, maxlen)
char *strbuf;

Collects a line of text from the console input, up to a maximum line length of *maxlen* characters. The return value is the length of the entered line. On return, the input line is terminated by a null byte only, so an empty line has length 0 (when the user types only a carriage-return character). There is no newline character returned in the buffer; this is a deviation from the *getline* function described in Kernighan & Ritchie.
If the number of characters entered reaches the given maximum minus one (to allow room for the terminating null), then the line will be considered complete and control will immediately return to the caller without waiting for a carriage-return to be typed. This happens because BDOS function 10 is used to read the console.

char *gets(str)
char *str;

Collects a line of input from the console and places it, null terminated, into memory at location *str*. The newline typed by the user to terminate the input line is **not** copied into the buffer; the character before the newline is immediately followed by the terminating null.
The return value is a pointer to the beginning of *str*.
The size of the provided buffer must be at least 1 byte longer than the longest string you ever expect entered, because of the terminating null.
Caution dictates making the buffer **large**, since an overflow here would most probably destroy neighboring data.
If the number of characters entered reaches 135, the line will be considered terminated.

printf(format,arg1,arg2,…)
char *format;

Formatted print function. Output goes to the standard output. Conversion characters supported in the standard version (must be lower case):

d	decimal integer format
u	unsigned integer format
c	single character
s	string (null-terminated)
o	octal format
b	binary format
x	hex format

Each conversion is of the form:

```
%  [-]  [[0] w]  [.n]  <conv. char.>
```

where *w* specifies the width of the field, and *n* (if present) specifies the maximum number of characters to be printed out of a string conversion. Default value for *w* is 1.
The field will be right justified, unless the dash is specified following the percent sign to force left justification.
If the value for *w* is preceded by a zero, then zeros are used as padding on the left of the field instead of spaces. This feature is useful for printing, say, hexadecimal addresses.
The '%' character may be specified literally in a format conversion by typing it twice in a row ("%%").
An enhanced version of *_spr* (the low-level formatting driver used by *printf*, *sprintf*, *fprintf* and *lprintf*) incorporating the *e* and *f* format conversions for floating point values used in the BCD floating point package, is available for compilation in the file BMATH.C

lprintf(format,arg1,arg2,…)
char *format;

Like *printf*, except the output is directed to the CP/M "LIST" device (printer) instead of to the console.

int scanf(format,arg1,arg2,…)
char *format;

Formatted input. This is analogous to *printf*,
but operates in the opposite direction.
The **%u** conversion is not recognized; use **%d** for both signed and
unsigned numerical input.
The arguments to *scanf* **must be pointers!!!!!**.
Note that input strings (denoted by a **%s** conversion specification
in the format string) are now terminated by any white space character
in the input stream, and that field width specifications are now supported
(both of these features are new for v1.6).
Returns the number of items successfully assigned. If console input is in
line buffered mode (through use of the *cmode(0)* call), then a single line of
input may be processed by as many successive calls to *scanf* as needed;
to make sure there isn't any stray extra input text in the console input
buffer make a *cmode(0)* call (only in line buffered mode.)
For a more detailed description of *scanf* and *printf*, see
Kernighan & Ritchie, pages 145-150.

3.3 Character and String Processing

int isalpha(c)
char c;

Returns true (non-zero) if the character *c* is alphabetic,
false (zero) otherwise.

int isupper(c)
char c;

Returns true if the character *c* is an upper case letter,
false otherwise.

int islower(c)
char c;

Returns true if the character *c* is a lower case letter,
false otherwise.

int isdigit(c)
char c;

Returns true if the character *c* is a decimal digit,
false otherwise.

int toupper(c)
char c;

If *c* is a lower case letter, then *c*'s upper case equivalent is returned. Otherwise *c* is returned.

int tolower(c)
char c;

If *c* is an upper case letter, then *c*'s lower case equivalent is returned. Otherwise *c* is returned.

int isspace(c)
char c;

Returns true if the character *c* is a "white space" character (blank, tab or newline). Otherwise returns false.

sprintf(string,format,arg1,arg2,…)
char *string, *format;

Like *printf*, except that the output is written to the memory location pointed to by *string* instead of to the console.

int sscanf(string,format,arg1,arg2,…)
char *string, *format;

Like *scanf*, except the text is scanned from the string pointed to by *string* instead of the console keyboard.
Returns the number of items successfully assigned. Remember that the arguments must be **pointers** to the objects requiring assignment.

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

Concatenates *s2* onto the tail end of the null terminated string *s1*. There must, of course, be enough room at *s1* to hold the combination.

int strcmp(s1,s2)
char *s1, *s2;

Returns a positive value if (s1 > s2), zero if (s1==s2), or a negative value if (s1 < s2). The standard ASCII collating sequence is used for comparisons; a string is "greater" if it comes later in alphabetical order.

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

Copies the string *s2* to location *s1*, returning a pointer to s1.
For example, to initialize a character array named *foo*
to the string "barzot", say

```
strcpy(foo,"barzot");
```

Note that the statement

```
foo = "barzot";
```

would be incorrect since an array name should **not** be used as
an lvalue without proper subscripting. Also, the expression
"barzot"
has as its value a *pointer* to the string "barzot", *not*
the string itself. So, for the latter construction to work,
foo must be declared as a pointer to characters instead of as an array.
This approach is dangerous, though, since the
natural method to append something onto the end of *foo* would be

```
strcat(foo,"mumble");
```

overwriting the six bytes following "barzot" (wherever "barzot"
happens to be stored within the code of the function),
probably with dire results.
There are two viable solutions. You can figure out the largest number
of characters that can possibly be assigned at *foo* and pad the initial
assignment with the appropriate number of blanks, such as in

```
foo = "barzot          ";
foo[6] = NULL;
```

or, you can declare a character array of sufficient size with

```
char work[200], *foo;
```

then have *foo* point to the array by saying

```
foo = work;
```

and assign to foo using

```
strcpy(foo,"mumble-fraz");
```

For an additional tool for use in initializing strings and string tables,
see the *initptr* function below.

int strlen(string)
char *string;

Returns the length of *string* (the number of characters encountered
before a terminating null is detected).

int index(string, substring)
char *string, *substring;

Returns position of *substring* in *string*, or –1 if not found.

int atoi(string)
char *string;

Converts the ASCII string to its corresponding integer (or unsigned) value. Acceptable format: Any amount of white space (spaces, tabs and newlines), followed by an optional minus sign, followed by a consecutive string of decimal digits. First non-digit terminates the scan.
A value of zero is returned if no legal value is found.

initw(array,string)
int *array;
char *string;

This is a kludge to allow initialization of integer arrays. *Array* should point to the array to be initialized, and *string* should point to an ASCII string of integer values separated by commas. For example, the UNIX C construct of

```
int values[5] =  -23,0,1,34,99;
```

can be simulated by declaring *values* normally with

```
int values[5];
```

and then inserting the statement

```
initw(values, "-23,0,1,34,99");
```

somewhere appropriate.

initb(array,string)
char *array, *string;

The equivalent of the above *initw* function for single-byte numeric values represented by elements in a character array.
String is of the same format as for *initw*, but the low order 8 bits of each value are used to assign to the consecutive bytes of *array*. Note that this function may **not** be used to initialize arrays of character pointers; it's not really meant for "characters", but for decimal integers all having values within the range of "character" variables and thus stored as characters.
NOTE: Some C programs will sometimes assign negative values to character variables, since standard C character variables are *signed* 8 bit quantities. In BDS C, character variables always have *unsigned* values and negative values can only be meaningfully assigned to 16-bit **int** variables.

initptr(strtab, str1, str2, str3, …, NULL)
char *strtab[], *str1, *str2, *str3 , …

This function is provided for the purpose of space-efficient initialization of a string pointer table, in lieu of regular initializers. The first argument must be a pointer to an array of string pointer variables. Subsequent arguments should be literal strings, except the final argument that must be 0 (symbolic constant NULL) to signal the end of the list of strings.
Here is an example, to initialize a list of pointers to names of the months:

```
char *months[12];
...
initptr(months, "January", "February", "March",
                "April", "May", "June", "July",
                "August", "September", "October",
                "November", "December", NULL);
```

3.4 File I/O

3.4.1 Introduction to BDS C File I/O Functions

There are two general categories of file I/O functions in the BDS C library. The **raw** (low-level) functions are used to read and write data to and from disk in even sector-sized chunks. The **buffered** I/O functions allow the user to deal with data in more manageable increments, such as one byte at a time or one line of text at a time. The raw functions will be described first, and then the buffered functions.

3.4.2 Filenames

Whenever a function takes a filename as an argument, that filename must be either a literal string or any expression whose value points to a filename. Legal filenames may be upper or lower case, but there must be no white space within the text of the filename.

3.4.2.1 The Disk Designator Prefix

The filename may contain an optional leading disk designator of the form "**d**:" to specify a particular CP/M drive; the default is the currently-logged disk. The character **d** may be any single-letter drive designator from A to Z (corresponding to some existing logical device on your system).

3.4.2.2 The User Area Prefix

An optional user area designator of the form "#/" may also appear as prefix to the filename, where # is a decimal number ranging from 0 to 31. If omitted, the current user area is assumed by default. If both a drive designator and a user-area designator are given, then the user-area prefix **must be first**. For example, to open the file named "foobar.zot" in user area 7 on drive C, you'd say:

```
open("7/c:foobar.zot", mode);
```

If any unprintable or nonstandard characters (such as control-characters) are detected within a filename, the filename will be rejected and an error value will be returned by the offended function. This somewhat alleviates the problem caused by trying to open a file whose name contains non-printing characters, but the mechanism still isn't entirely foolproof. Be careful when constructing filenames.

3.4.3 Error Handling

3.4.3.1 The Errno/Errmsg Functions

Whenever an error occurs, the usual –1 (ERROR) value is returned by the troubled function. After this happens, but before any new file I/O errors are drawn, the *errno* function may be called to return a special error code number giving more detailed information about the error. If you pass the value returned by *errno* to the *errmsg* function, then *errmsg* will return a pointer to a string which describes in words exactly what kind of error occurred. Here is an example of the use of this mechanism, in this case to diagnose errors which occur during a *write* statement:

```
if (write(fd, buffer, nsects) != nsects)
{
        printf("Write error: %s n",errmsg(errno()));
        ...             /* try to recover somehow */
}
```

Note that the *write* function is the exception to the rule that a value of –1 (ERROR) is always returned on an error condition; *write* returns the number of sectors successfully written, which should be considered an error if not equal to the number of sectors specified by the **nsects** parameter.

3.4.3.2 Random-Record Overflow

The *oflow* function is provided to detect when an overflow has occurred in reading/writing a large file. This only happens if you try to read/write past the 65535th sector of a file. Note that this only applied to systems having the standard CP/M 2.2 8-megabyte file size limitation.

3.4.4 The Raw File I/O Functions

int open(filename, mode)
char *filename;

> Opens the specified file for input if *mode* is zero, output if *mode* is equal to 1, or both input and output if *mode* is equal to 2. Returns a file descriptor, or –1 on error. The file descriptor is for use with *read*, *write*, *seek*, *tell*, *fabort* and *close* calls.

int creat(filename)
char *filename;

Creates an empty file having the given name,
first deleting any existing file with that name.
The new file is automatically opened for both reading and writing,
and a file descriptor is returned for use with
read, *write*, *seek*, *tell*, *fabort*, and *close* calls.
A return value of –1 indicates an error.

int close(fd)

Closes the file specified by the file descriptor *fd*, and frees
up *fd* for use with another file.
Unless running under MP/M II, disk accesses
will only take place when a file that was opened for writing is closed;
if the file was only open for reading,
then the fd is freed up but no actual CP/M call is performed
to close the file.
Close should not be used for buffered I/O files. Instead, use *fclose*.
Returns –1 on error.
Note that all open files are automatically closed upon return to the
run-time package from the **main** function, or when the *exit*
function is invoked. To prevent an open file from being closed,
use the *fabort* function.

int read(fd, buf, nbl)
char *buf;

Reads *nbl* blocks (each 128 bytes in length)
into memory at *buf*
from the file having
descriptor *fd*.
The r/w pointer associated
with that file is positioned following the just-read data; each call to
read causes data to be read sequentially from where the last call
to *read* or *write* left
off. The *seek* function may be used to modify the r/w pointer.
Returns the number of blocks actually read, 0 for EOF, or –1 on error.
Note that if you ask for *n* blocks of data when there are only *x*
blocks actually
left in the file (where $0 < x < n$), then *x*
would be returned on that call, 0 on the next call (provided *seek*
isn't used), and then –1 on subsequent calls.

int write(fd, buf, nbl)
char *buf;

Writes *nbl* blocks from memory at *buf* to file *fd*. Each call to *write* causes data to be written to disk sequentially from the point at which the last call to *read* or *write* left off, unless *seek* is used to modify the r/w pointer.
Returns –1 on hard error, or the number of records successfully written. If the return value is non-negative but different from *nbl*, it probably means you ran out of disk space; this should be regarded as an error.

int seek(fd, offset, code)

Modifies the next read/write record (sector) pointer associated with file *fd*.
If *code* is zero, then *seek* sets the r/w pointer to *offset* records.
If *code* is equal to 1, then *seek* sets the r/w pointer to its current value **plus** *offset* (*offset* may be negative.)
If *code* is equal to 2, then *seek* sets the r/w pointer to the **end-of-file** record number **plus** *offset*. Note that *offset* must be negative in order for this type of seek to end up pointing to an existing record in the file. If *code* is 2 and *offset* is zero, the r/w pointer is made ready for appending to the file.
A return value of –1 indicates that some kind of BDOS error was returned during a seek relative to EOF (*code* equal to 2). The *errno* function will give more details about the kind of error that occurred.
*Seek*s should **not** be performed on files open for buffered I/O.

int hseek(fd, hoffset, offset, code)

This variation of the *seek* function is for use on systems supporting an extra-large (greater than 8 megabyte) file size. Since a 16-bit sector offset value only allows addressing up to 8 megabytes, the *hseek* function actually takes a 24-bit sector offset value, broken up into a high-order 8-bit portion (**hoffset**) and a low-order 16-bit word (**offset**). The two offset parameters are combined to form a single, signed 24-bit offset value. In all other aspects, this function works just like the *seek* function above. For example, to seek to the 8 megabyte mark in a file, you'd say:

```
hseek(fd, 1, 0, 0);  /* Seek to 65536th  sector */
```

To seek to the last written sector of a file, you'd say:

```
hseek(fd, -1, -1, 2);  /* One sector before EOF */
```

int tell(fd)

Returns the value of the r/w pointer associated with file *fd*. This number indicates the next sector to be written to or read from the file, starting from 0.

int htell(fd)

This function returns only the high-order byte of the 24-bit random record position value of a file. This is only useful on systems supporting a larger than 8 megabyte file size. To obtain the low-order 16 bits of the random record position value, use the conventional *tell* function as above.

int unlink(filename)
char *filename;

Deletes the specified file from the file system.
Use with caution!

int rename(old, new)
char *old, *new;

Renames a file in the obvious manner.
The specified file **must not be open** while *rename* is being used on it, nor should a file having the new name already exist. This function simply performs the low-level BDOS rename operation, and is subject to all the potential disasters of that low-level call. If there is any possibility that a file with the new name already exists, then the *unlink* function should be used before *rename* to insure the consistency of the file system.
Returns (ERROR) –1 on error.

int fabort(fd)

Frees up the file descriptor *fd* without bothering to close the associated file. If the file was only open for reading, this will have no effect on the file. If the file was opened for writing, though, then any changes made to the currently open extent since it was last opened will be ignored, but changes made in other extents will **probably** remain in effect. Don't *fabort* a file open for write, unless you're willing to lose some of the data written to it.

unsigned cfsize(fd)

Computes the **exact** file size (in sectors) of the given open file, without affecting the r/w pointer associated with the file. Note that the size returned here **will** reflect data written to new extents before they are closed, unlike raw BDOS function 35.
This function is NOT for use with files larger than 8 megabytes. To obtain the size of such a file, use *hseek* to seek to the EOF, then use *htell* and *tell* to get the high-order byte and low-order word, respectively, of the 24-bit record size of the file.

int oflow(fd)

Returns true (non-zero) if an overflow has occurred into the high order (third) byte of the random-record field of the FCB associated with the given open file.

int errno()

Returns the code number for the last error condition detected after a file I/O operation. See below for a list of the error messages associated with the codes.

char *errmsg(errnum)

Given an error code returned by *errno*, this function returns a pointer to an ASCII string describing the given error condition in English. Here is a summary of all possible error numbers and their associated messages:

Error-code	Text
0	No error has occurred yet
1	Reading unwritten data
2	Disk out of data space
3	Can't close current extent
4	Seek to unwritten extent
5	Can't create new extent
6	Seek past end of disk
7	Bad file descriptor given
8	File not open for read
9	File not open for write
10	No file descriptor slots left
11	File not found
12	Bad mode given to *open*
13	Can't create file
14	Seek past 65535th record

int setfcb(fcbaddr, filename)
char fcbaddr[36];
char *filename;

Initializes a 36-byte CP/M file control block located at address *fcbaddr* with the null-terminated name pointed to by *filename*. Lower-case characters in the filename string are converted to upper case, and the appropriate number of ASCII blanks are generated to pad both the filename and extension fields of the fcb.
The next-record and extent-number fields of the fcb are zeroed.
If any strange character (of the kind not usually desirable in the name or extension fields of a file control block) are encountered within the filename string, then the offending character and remainder of the filename string will be ignored.

char *fcbaddr(fd)

Returns the address of the internal (usually invisible) file control block associated with the open file having descriptor *fd*.
–1 is returned if *fd* is not the file descriptor of an open file.

3.4.5 The Buffered File I/O Functions

In order to simplify the programming of object-oriented file input and output applications, the buffered I/O function library is provided. This set of routines, built upon the low-level I/O functions described above, conforms fairly well to the standard Kernighan and Ritchie I/O library.

A "File Pointer" (usually represented by the variable name *fp* in the function descriptions below) is the standard object used to identify a particular active buffered I/O file. The *fp* value for a file is assigned through a call to the *fopen* function, and should always be declared as a pointer to the symbolic type FILE. For example,

```
FILE *fp1, *fp2, *fp3; /* Declare three file pointers */
```

The file pointer returned by *fopen* will point to a buffer structure dynamically allocated during the *fopen* call. The technical structure of the I/O buffer is

```
struct _buf {
        int _fd;
        int _nleft;
        char *_nextp;
        char _buff[NSECTS * SECSIZ];
        char _flags;
};
```

The NSECTS symbol, defined in the STDIO.H header file, determines the number of sectors of in-memory buffering used by the buffered I/O functions. The BDS C distribution package comes with NSECTS set to 8, so that all buffered I/O is performed using 1K byte memory buffers. If you wish to alter this value, you must first change NSECTS in STDIO.H, then recompile the STDLIB?.C source files and create a new DEFF.CRL object library containing the modified buffered I/O functions.

Wherever a file pointer is called for in the parameter list to a buffered i/o function, it is permissible to use one of six special symbols in order to direct the i/o to or from a special device instead of a file. The recognized device symbols are:

```
stdin     --    Standard input stream (console input)
stdout    --    Standard output stream (console output)
stdlst    --    Standard list device (printer)
stdrdr    --    Standard reader device
stdpun    --    Standard punch device
stderr    --    Standard error device (console output)
```

IMPORTANT: All programs using buffered I/O must #include the standard header file, STDIO.H, at the beginning of the source file. Here are the functions:

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

Opens the specified file for buffered input or output,
initializes the associated buffer, and returns a file pointer to be used in all subsequent references to operations on the associated file.
Possible values of *mode* are as follows:

```
"r"     Text input, do CR-LF --> '\n' translations
"w"     Text output, new file, translate '\n' to CR-LF
"a"     Text output, append to existing file,
                        translate '\n' to CR-LF
"rb"    Binary input
"wb"    Binary output, create new file
"ab"    Binary output, append to existing data
```

Returns NULL (0) on error.

int fgetc(fp)
FILE *fp;

Returns the next byte from the buffered input file specified by file pointer *fp*.
This is the same function as *getc*.
The symbolic values *stdin* and *stdrdr* may be used instead of a file pointer argument with any buffered input function, to direct the input from the console or the reader:

```
fgetc(stdin)    is equivalent to "getchar()".
fgetc(stdrdr)   reads a char from the "reader" device.
```

A value of –1 is returned in physical EOF and on error conditions. If the file was opened in text mode, the logical EOF character (Control-Z, or 0x1A) is also mapped into –1 by *fgetc*.
Since the ERROR value of –1 conflicts with a –1 indicating routine EOF, the way to differentiate between a routine EOF and an error condition is to test the value of errno(), which returns NULL after an EOF has been encountered and non-zero after errors.

ungetc(c, fp)
char c;
FILE *fp;

Pushes the character *c* back onto the input stream *fp*.
The next call to *fgetc*
on the same stream will then return *c*. No more than one character should be pushed back at a time.

int getw(fp)
FILE *fp;

Returns next 16 bit word from input stream *fp*.
via two consecutive calls to *fgetc*.
–1 returned on error.

int fputc(c, fp)
char c;
FILE *fp;

Writes the byte *c* to the output stream *fp*.
If the file was opened in the default text mode, then newline ('\n') characters are automatically translated into a CR-LF combination in the output file.
The symbolic values *stdout*, *stdlst*, *stdpun* and *stderr* may be used in place of a file pointer with any buffered output routine, to direct the output character to the standard output, list device, punch device or standard error (console) device instead of to a file:

```
putc(c,stdout)  is equivalent to "putchar(c)".
putc(c,stdlst)  writes the character to the CP/M
                "list" device.
putc(c,stdput)  writes the character to the CP/M
                "punch" device.
putc(c,stderr)  writes the character to the
                standard error stream, which
                is always the console output device
                under CP/M. This may be used to
                guarantee that output goes to the
                console in applications where the
                directed I/O package (DIO) is being
                used and the standard output
                may be directed into a file.
```

Returns –1 (ERROR) on error.

int putw(w, fp)
FILE *fp;

Writes the 16 bit word *w* to buffered output stream *fp*, via two consecutive calls to *fputc*.
Returns –1 on error.

int fread(buf, size, count, fp)
char *buf;
unsigned size, count;
FILE *fp;

Efficiently reads *count* objects, each of size *size* bytes, from a buffered input file. Number of bytes to be read is exactly (size * count).
NOTE: CR-LF —> '\n' translation for text files is **not** performed.
Returns number of items of size **size** successfully read (up to **count**), or ERROR on error.

int fwrite(buf, size, count, fp)
char *buf;
unsigned size, count;
FILE *fp;

Efficiently writes *count* objects, each of size *size* bytes,
to a buffered output file. Number of bytes to be written is equal to (size * count).
NOTE: '\n' —> CR-LF translation for text files is **not** performed.
Returns number if items (up to *size*) successfully written (up to *count*), or
ERROR on error.

int fflush(fp)
FILE *fp;

Flushes output stream *fp*, i.e., makes sure that any
characters written to the output buffer since it last filled up
are written to the file on disk (provided the program isn't prematurely aborted
before the *exit* routine closes all files).
Fflush is for use with buffered *output* files; attempting to
invoke it on an input file will have no effect.
Note that an automatic *fflush* occurs whenever an output buffer fills
up, as well as when an output file is closed (via the *fclose* function).

int fclose(fp)
FILE *fp;

Closes the specified buffered I/O file. If the
file was opened for writing, then an automatic *fflush* is performed
to flush the output buffer before the file is closed.
When a text file opened for writing is closed,
then a Control-Z character is automatically appended
onto the end of the file.

int ferror(fp)
FILE *fp;

Returns TRUE if an error has occurred on the specified buffered I/O stream.

int feof(fp)
FILE *fp;

Returns TRUE if an end-of-file (EOF) has been encountered on the
specified buffered input stream.

int clearerr(fp)
FILE *fp;

Clears any error condition that may have been previously set for the
specified buffered I/O stream.

int fprintf(fp, format, arg1, arg2,…)
FILE *fp;
char *format;

Like *printf*, except that the formatted output is written to the output stream *fp*.
Returns –1 on error.

int fscanf(fp, format, arg1, arg2,…)
FILE *fp;
char *format;

Like *scanf*, except that the text input is scanned from the input stream *fp* instead of from the console.
Remember that arg1, arg2, etc., must be **pointers**!
Returns the number of items successfully assigned, or –1 if an error occurred in reading the file.

char *fgets(buf, maxlen, fp)
char *buf;
int maxlen;
FILE *fp;

Reads a line in from input stream specified by *fp* (up to *maxlen* characters), and places it in memory at the location pointed to by *str*.
NULL (zero) is returned on end-of-file, whether it is a physical end-of-file (attempting to read past the last sector of a file) or a control-Z (CPMEOF) character in the file.
Otherwise, a pointer to the string (the same as the parameter *str*) is returned.

int fputs(str, fp)
char *str;
FILE *fp;

Writes the null-terminated string from memory at *str* into the output stream specified by *fp*.
If a null (zero byte) is found in the string before a newline ('\n'), then there will be no line terminator at all appended to the line on output (allowing partial lines to be written.)

Chapter 4

Notes to APPENDIX A of "The C Programming Language"

4.1 Introduction

This chapter is a direct comparison between BDS C and the standard C definition outlined in Appendix A of the Kernighan and Ritchie The C Programming Language textbook. BDS C is designed to be a subset of UNIX C, and therefore most sections of the **C Reference Manual** apply to BDS C directly. The purpose of this appendix is to annotate those sections in which BDS C deviates from the definition appearing in the textbook.

After presenting a general summary of differences between the two implementations, I'll go into detail by referring to appropriate section numbers from the book and describing how BDS C *differs* from what is stated there. Any sections that are appropriate as they stand (with regard to BDS C) will not be listed.

Here is a short summary of BDS C's most significant deviations from UNIX C:

1. The entire source file is loaded into main memory at once, instead of being passed through a window. This limits the maximum length of a single source function to the size of available memory.

2. Compilation is accomplished directly into 8080 machine code, with no intermediate assembly language file produced.

3. BDS C is written in 8080 assembler language, **not** in C itself. If BDS C were written in itself, the compiler would be several times as large and run **nowhere** as fast as the present speed. Remember that we're dealing with 8080 code here, *not* PDP-11 code as in the original UNIX implementation.

4. The variable types **short int**, **long int**, **float** and **double** are not supported.

5. There are no explicitly declarable storage classes. **Static** and **register** variables do not exist; all variables are either *external* or *automatic*, depending on the context in which they are declared.

6. The complexity of declarations is restricted by certain rules.

7. Initializers are not supported.

8. String space storage allocation must be handled explicitly (there is no automatic allocation/garbage collection mechanism).

4.2 Notes to Appendix A

The following is a section-by-section annotation to the **C Reference Manual**[14] . For the sake of brevity, some of the items mentioned above will not be pointed out again; any references to floats, longs, statics, initializations, etc., found in the book should be ignored.

1. Introduction

BDS C is designed for 8080-based microcomputer systems equipped with the CP/M operating system, and generates 8080 binary machine code (in a special relocatable format) directly from given C source programs. Naturally, BDS C will also run on any processor that is upward compatible with the 8080, such as the Z-80 or 8085.

2.1 Comments

Comments **nest** by default; to make BDS C process comments the way Unix C does, the **–c** option must be given to CC during compilation.

2.2 Identifiers (names)

Upper and lower case letters are distinct (different) for variable, structure, union and array names, but *not* for **function** names[15] . Thus, function names should always be written in a single case (either upper **or** lower, but not mixed) to avoid confusion. For example, the statement

```
char foo,Foo,FoO;
```

declares three character variables with different names, but the two expressions

```
printf("This is a test");
```

and

```
prINTf("This is a test");
```

are equivalent.

2.3 Keywords

BDS C keywords:

14. Appendix A of The C Programming Language, the Kernighan & Ritchie textbook

15. Function names are stored internally as upper-case-only.

```
int             else
char            for
struct          do
union           while
unsigned        switch
goto            case
return          default
break           sizeof
continue        begin
if              end
register        void
short
```

Upper and lower case are not distinguished for keywords, e.g., **WHILE** is equivalent to **while**.

Identifiers with the same name as a keyword are not allowed, although keywords may be imbedded within identifiers (e.g. *charflag*).

On terminals which do not support the left and right curly-brace characters **{** and **}**, the keywords **begin** and **end** may be substituted instead. Note that you *cannot* have any identifiers in your programs named either "begin" or "end", since these are recognized as keywords by the compiler.

4. What's in a name?

There are only two storage classes, **external** and **automatic**, but they are *not* explicitly declarable. The context in which an identifier is declared always provides sufficient information to determine whether the identifier is external or automatic: declarations that appear outside the definition of any function are implicitly external, and all declarations of variables within a function definition are automatic.

Automatic variables have a lexical scope that extends from their point of declaration until the end of the current function definition. A single identifier may not normally appear in a declaration list more than once in any given function, which means that a local structure member or structure tag may *not* be given the same name as a local variable, and vice versa. See subsection 11.1 for a special case.

In BDS C, there is no concept of **blocks** within a function. Although a local variable may be declared at the start of a compound statement, it may not have the same name as a previously declared local automatic variable. In addition, its lexical scope extends **past** the end of the compound statement and all the way to the end of the function.

I strongly suggest that all automatic variable declarations be confined to the beginning of function definitions, and that the practice of declaring variables at the head of other compound statements be avoided.

If several files share a common set of external variables, then all external variable declarations must be identically ordered within each of the files involved[16]. The external variable mechanism in BDS C is handled much like the unnamed COMMON facility of FORTRAN. For example: if

16. The recommended procedure for a case such as this is to prepare a single file (using your text editor) containing all common external variable declarations. The file should have extension **.H** (for "header"), and be specified at the start of each source file via use of the **#include** preprocessor directive.

your **main** source file declares the external variables **a**,**b**,**c**,**d** and **e**, in that order, while another file uses only **a**, **b** and **c**, then the second file need not declare **d** and **e**. On the other hand, if the second file used **d** and **e** but not **a**, **b** or **c**, then *all* of the variables must be declared so that **d** and **e** (from the second file) do not overlap with **a** and **b** (from the first file) and cause big trouble. As an added inconvenience, *all* external variables used in a *program* (set of dependent source files) must be declared within the source file containing the "main" function, regardless of whether or not that source file uses them all.

To summarize: keep all external declarations common to several source files of a program in ".H" files, and use **#include** within each source file of the program to read in the **same** ".H" file(s) in the **same** order. This will insure that each source file sees the same external data declared in exactly the same manner.

6.1 Characters and integers

Sign extension is never performed by BDS C. Characters are interpreted as 8-bit *unsigned* quantities in the range 0-255.

A CHAR VARIABLE CAN NEVER HAVE A NEGATIVE VALUE IN BDS C.

Be careful when, for example, you test the return value of functions such as *getc*, which return –1 on error but "characters" normally. Actually, the return value is an **int** always, with the high byte guaranteed to be zero when there's no error. If you assign the return value of *getc* to a character variable, then a value of –1 will turn into 255 as stored in the 8-bit character cell, and testing a character for equality with –1 will never return true. Be careful in these kinds of situations.

Most arithmetic on characters is accomplished by converting the character to a 16-bit quantity having a zero high-order byte. In some non-arithmetic operations, such as assignment expressions, BDS C will optimize code generation by dealing with **char** values on a byte-only basis. To take advantage of this, declare any variables you trust to remain within the 0-255 range as **char** variables.

7. Expressions

Division-by-zero and mod-by-zero both result in a value of zero. No error of any kind is generated in these cases.

7.1 Primary Expressions

The order of evaluation of the parameters in a function call is **reversed**. I.e., the last parameter is evaluated first and pushed on the stack, then the next-to-last is evaluated and pushed on the stack, etc…this is done so that the parameters appear in ascending order to the function being called, for the benefit of functions taking a variable number of parameters.

7.2 Unary Operators

The operators

```
(type-name) expression
sizeof (type-name)
```

are not implemented. The **sizeof** operator may be used in the form

```
sizeof expression
```

provided that *expression* is **not an array**. To take the **sizeof** an array, the array must be placed all by itself into a structure, allowing the **sizeof** the structure to then be taken. Another possibility is to take the **sizeof** a single element in the array, then multiply that by the number of elements in the array to yield the size of the overall array.

The **sizeof** operator may **not** appear within expressions used as dimensions for array declarations.

7.5 Shift operators

The operation **»** is always *logical* (0-fill).

7.11, 7.12 Logical AND and OR operators

The two operators **&&** and **||** have *equal* precedence in BDS C, making parenthesization necessary in certain cases where it wouldn't be under Unix C. Any expressions involving complex combinations of **&&** and **||** are basically confusing anyway, and should be parenthesized just on general principles.

8. Declarations

Declarations have the form:

```
declaration:
        type-designator declaration-list ;
```

There are no "storage class" specifiers.

8.1 Storage class specifiers

Not implemented.

8.2 Type specifiers

The type-specifiers are

```
type-designator:
        char
        int
        unsigned
        register
        struct-or-union-designator
```

The type **register** willbe assumed synonymous with **int**, unless it is used as a modifier (e.g. **register unsigned** foo;), in which case it will be ignored completely.

The keyword **void** is treated as synonymous with **int**, and may be used to document the fact that a function does not return a value. There are no other "adjectives" allowed;

```
unsigned int foo;
```

must be written as

```
unsigned foo;
```

8.3 Declarators

Initializers are not allowed. Thus, the syntax for declarator lists is:

```
declarator-list:
        declarator
        declarator , declarator-list
```

8.4 Meaning of declarators

UNIX C allows arbitrarily complex typing combinations, making possible declarations such as

```
struct foo *( *( *bar[3][3][3]) () ) ();
```

which declares bar to be a 3x3x3 array of pointer to functions returning pointers to functions returning pointers to structures of type foo. BDS C would not allow that particular declaration.

Here is an informal summary of the declaration syntax BDS C **will** accept:

First, let a **simple-type** be defined by

```
simple-type:
        char
        int
        unsigned
        struct
        union
```

and a **scalar-type** by

```
scalar-type:
        simple-type
        pointer-to-scalar-type
        pointer-to-function
```

The final kind of scalar type, the **pointer-to-function**, is a variable which may have the address of a function assigned to it and then be used (with the proper syntax) to call the function. Because of the way BDS C handles these guys internally, pointers to pointer-to-function variables will not work correctly, although pointers to functions returning any other scalar type (except **struct**, **union**, and pointer-to-function) are OK.

So far, scalar-types cover declarations such as

```
int x,y;
char *x;
unsigned *fraz;
char **argv;
struct foobar *zot, bar;
int *( *ihtfp)();
```

The last of the above examples declares ihtfp to be a pointer to a function which returns a pointer to integer.

Building on the scalar-type idea, we define an **array** to be a one or two dimensional collection of scalar-typed objects (including pointer-to-function variables). Now we can have constructs such as

```
char *x[5][10];
int **foo[10];
struct steph bar[20][8];
union joyce *ohboy[747];
int * (foobar[10] ) ();
```

The last of the above examples declares foobar to be an array made up of ten pointers to functions returning integers.

Next, we allow functions to return any scalar type except pointer-to-function, **struct** or **union** (but not excluding *pointers* to structures and unions.)

Some more examples:

```
char *bar();
```

declares bar to be a function returning a pointer to character;

```
char *( *bar)();
```

declares bar to be a *pointer* to a function returning a pointer to characters;

```
char *( *bar[3][2]) ();
```

declares bar to be a 3 by 2 array of individual pointers to functions returning pointers to characters;

```
struct foo zot();
```

attempts to declare zot to be a function returning a structure of type foo. Since functions cannot return structures, this would cause unpredictable results.

```
struct foo *zot();
```

is OK. Now zot is declared as returning a *pointer* to a structure of type foo.

Note that explicit pointers-to-arrays **cannot** be declared. In other words, a declaration such as

```
char (*foo) [5];
```

would *not* succeed in declaring foo to be a pointer to an array. The preceding declaration ends up having exactly the same effect as

```
char *foo[5];
```

Any formal function parameter declared as an array is handled internally as a "pointer-to-array", causing an automatic indirection to be performed whenever the appropriate array identifier is used in an expression. This makes passing arrays to functions as easy as pi. For an extensive example of this mechanism, check out the *Othello* program included with some versions of the BDS C package (but always available from the C User's Group).

8.5 Structure and union declarations

"Bit fields" are not implemented. Thus we have

```
struct-or-union-designator:
        struct-or-union { struct-decl-list }
        struct-or-union identifier { struct-decl-list }
        struct-or-union identifier

struct-or-union:
        struct
        union

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

struct-declaration:
        type-designator declarator-list ;

declarator-list:
        declarator
        declarator, declarator-list
```

Names of members and tags in structure definitions must not be identical to any other local identifier names. The only time more than one structure or union per function can use a given identifier as a member is when all instances have the identical type and offset; see subsection 11.1.

8.6 Initializers

No initializers allowed. The library functions *initb*, *initw* and *initptr* have been provided to facilitate the initialization of certain types of arrays.

All external variables are now automatically initialized to zero unless the CLINK **–z** option is given during linkage.

8.7, 8.8 Type names

Not applicable to BDS C. **typedef** is not implemented.

9.2 Blocks

There are no "blocks" in BDS C. Variables cannot be declared as local to a block; declarations appearing **anywhere** in a function remain in effect until the end of the function.

9.6 For statement

The **for** statement is not completely equivalent to the **while** statement as illustrated in K&R, for this reason: should a **continue** statement be encountered while performing the *statement* portion of the **for** loop, control would pass to *expression-3*. In the **while** version, though, a **continue** would cause control to pass to the test portion of the loop directly, never executing *expression-3* during that particular iteration. The representation given in section 9.9, on the other hand, is correct since the increment is *implied* (to occur at **contin:**) rather than written explicitly.

This is merely an inconsistency in documentation; both the UNIX C compiler (as far as I can tell) and the BDS C compiler handle the **for** case correctly.

9.7 Switch statement

There may be no more than 200 **case** statements per switch construct.

Note that multiple cases each count as one, so the statement

```
case 'a': case 'b': case 'c': printf("a or b or c");
```

counts for three cases.

9.12 Labeled statement

A label directly following a **case** or **default** is not allowed. The label should be written first, and then be followed by the **case** or **default** keyword. For example,

```
case 'x': mumble: zap = frotz;
```

is incorrect, and should be changed to

```
mumble: case 'x': zap = frotz;
```

10. External definitions

Type specifiers must be given explicitly in all cases except function definitions (where the default is **int**.)

11.1 Lexical scope

Members and tags within structures and unions should not be given names that are identical to other types of declared identifiers. BDS C does not allow any single identifier to be used for

more than one thing at a time (except when a local identifier temporarily shadows a similarly named external identifier). This means that you cannot write declarations such as:

```
struct foo {            /* define struct of type "foo" */
        int a;
        char b;
} foo[10];      /* define array named "foo" made up
                   of structures of type "foo"  */
```

which are basically confusing and shouldn't be used anyway, even if UNIX C **does** allow them.

The one exception to this rule involves structure members. The compiler will tolerate the same identifier being used as a *member* within the definition of different structures, as long as 1) the *type* and 2) the *storage offset* (from the base of the structure) are identical for both instances. The following sequence, for example, uses the identifier "cptr" in this allowable manner:

```
struct foo {
        int a;
        char b;
        char *cptr;     /* type: char *, offset: 3  */
};

struct bar {
        unsigned aa;
        char xyz;
        char *cptr;     /* type: char *, offset: 3  */
};
```

11.2 Scope of externals

There is no **extern** keyword; all external variables must be declared *in exactly the same order* within each file that uses any subset of them. Also, *all* external variables used in a program must be declared within the source file that contains the "main" function.

Here is how externals are normally handled: location 0015h of the run-time package (usually memory location 0115h at run-time) contains a pointer to the base of the external variable area. All external variables are accessed by indexing off this pointer.[17] The external data area for the *entire program* is assumed by CLINK to be equal to the space needed by all external data defined in the "main" source file. Because no information is recorded within CRL files about external storage or external names (other than the total number of bytes involved and, optionally, the explicit starting address of the externals), it is up to the user to make sure that each source file contains an identical list of external declarations. Although the names need not necessarily be identical for each corresponding external variable in separate files, the types and storage requirements should certainly correspond to avoid overlap and mix-up.

It would not be far off the mark to consider BDS C external variables as just one big FORTRAN-like COMMON block.

> Reminder: if you use the library functions *alloc* and *free*, you must include the header file STDIO.H in your program, since there are several external data objects

17. The **–e** *xxxx* option to CC may be used to locate the external variable area at absolute location *xxxx*, thereby considerably speeding up and shortening the code produced by the compiler. Even so, all the declaration constraints must still be observed.

required by *alloc* and *free* declared in STDIO.H, and omission of these declarations within any source file having external variables would cause an undesirable data overlap.

12.1 Token replacement

All forms of the **#define** preprocessor directive are supported, including parameterized defines. Note that *recursive* (mutually referential) parameterized **#define** operations are not detected, and if attempted will cause a string overflow.

12.2 File Inclusion

If double-quotes are used to delimit the filename (e.g. **#include "**filename**"**), and no explicit drive or user-area designator appear preceding the filename, then the file is presumed to reside in the current directory only and compilation will abort if the file isn't there. If angle brackets (**#include** <filename>) are used, then only the default disk drive/user area (as described in chapter 1) is searched.

Note that **#include** directives are processed on-the-fly as the source file is being read in from disk, whereas conditional compilation directives are only processed on a later pass after included files have already been loaded. Therefore, the compiler will attempt to process an **#include** directive placed within a conditional compilation block even when the condition evaluates as false. As long as the files named in all **#include** directives are found, things will still work correctly because the appropriate code will simply be ignored later when the conditionals are processed…but, if the file named by **any #include** directive cannot be found, CC will print an error and abort the compilation.

Although file inclusion may be nested to any reasonable depth, error reporting recognizes only one level of nesting. Try experimenting with the "**-p**" option of CC, varying the level of inclusion nesting, to see exactly what happens.

12.3 Conditional Compilation

All standard conditional compilation directives are now supported, but the expression taken by the **#if <expr>** directive is limited to the following syntax:

```
<expr>  :=      <expr2>                 or
                <expr2> && <expr>       or
                <expr2> || <expr>

<expr2> :=      <decimal-constant>      or
                !<expr2>                or
                (<expr>)
```

The <decimal-constant> may be symbolic (yielding a plain decimal constant after **#define** substitution is complete), but is always treated as a logical value by the **#if** processor. I.e., a value of 0 is false, and any other value is true.

Nesting of conditional compilation directives is now fully supported.

12.4 Line Control

Not implemented.

15. Constant expressions

BDS C will simplify constant expressions at compile-time **only** when the constant expressions appear **immediately** after one of the following keywords: left square brackets, the **case** keyword, assignment operators, commas, left parentheses, and the **return** keyword. Any constant expression that doesn't follow one of the aforementioned keywords is guaranteed to *not* be simplified at compile-time.

The standard procedure for insuring the compile-time evaluation of constant expressions, **especially when contained within larger expressions involving elements other than constants**, is to place the constant expressions within parentheses. Thus, statements such as

```
x = x + y + 15*10;
```

will not be simplified (i.e., will cause the compiler to generate code to multiply 15 and 10) and, in general, will produce longer and slower code than the better form of:

```
x = x + y + (15*10);
```

All multiplicative operations on constants and constant expressions are performed as *unsigned* operations.

18.1 Expressions

The unary operators are:

```
*  &  -  !  ~  ++  --  sizeof
```

The binary operators **&&** and || have *equal* precedence. If the two operators are mixed at an equal nesting level, evaluation proceeds left to right. As in any expression involving **&&** or ||, a zero valued expression preceding an **&&** operator forces a value of zero for the entire expression and terminates evaluation, and a non-zero valued expression preceding an || operator forces a value of 1 (or true) and similarly halts the evaluation of further terms at the same nesting level. A sequence such as

```
a = 5; b = 0;
if (b && a || a)
        puts("true\n");
else
        puts("false\n");
```

prints "false" under BDS C, but might print "true" under other C compilers. To avoid system dependency in cases like this, explicit parentheses should be used to force order of evaluation. The second line of the sequence above, when changed to

```
if ( (b && a) || a)
```

would execute consistently on all systems.

The **sizeof** operator cannot correctly evaluate the size of an array, nor can it be used within an array declaration as a constant expression. See section 7.2 for additional restrictions on the use of the **sizeof** operator.

18.2 Declarations

The *complete* syntax for declarations is

```
declaration:
        type-designator declarator-list ;

type-designator:
        char
        int
        register  (same as int)
        unsigned
        struct-or-union-designator

declarator-list:
        declarator
        declarator , declarator-list

declarator:
        identifier
        ( declarator )
        * declarator
        declarator ()
        declarator [ constant expression ]

struct-or-union-designator:
        struct { declarator-list }
        struct identifier { declarator-list }
        struct identifier
        union { declarator-list }
        union identifier { declarator-list }
        union identifier
```

18.4 External definitions

```
data-definition:
        type-designator declarator-list ;
```

18.5 Preprocessor

The following preprocessor directives are now supported:

```
#define identifier token-string
#include "filename"
#include <filename>
#if expression
#ifdef identifier
#ifndef identifier
#else
#endif
#undef identifier
```

#Defines may appear anywhere in the source file, their scope extending until the end of the file, or until the identifier is re-**#define**d or **#undef**ed.

The

```
#if <expr>
```

directive is supported, but legal expression elements are limited to constants (including symbolic constants) and a small set of operators. The **#if** directive allows user to write system-dependent conditional expressions without having to resort to using **#ifdef**/**#ifndef** and/or play games with commenting and uncommenting **#define** directives. See section 12.3 above for the complete syntax.

The **#include** directive should *not* appear inside any conditional compilation directives. This is because the **#include** directives are all processed on-the-fly by the compiler as an input file is read in from disk, and conditional compilation processing doesn't take place until after the entire file has been read in. Thus, an **#include** directive will always cause the compiler to try and read the named file, even if the directive is placed within a false conditional compilation block. This may be considered a design flaw, but there is no way to process all conditional directives on-the-fly and still read the source file in at a reasonable speed from standard 8" single-density CP/M disks.

When using conditional compilation, note that each and every **#else** directive *must* be followed (eventually) by a matching **#endif** directive.

File inclusion may nest to any depth[18], but both the **–p** CC option and error reporting for both CC and CC2 become easier to deal with if file inclusion is limited to a single level.

18. Mutually inclusive files, though, will certainly cause an overflow.

Chapter 5

The RED Screen Editor

by
Edward K. Ream

Edward K. Ream has modifed his RED screen editor to interact with the new BDS C v1.6 error diagnostic mechanism. When the CC option "-w" is specified, or if the RED output option has been chosen through use of the new BDS C CCONFIG program, then a special error file called PROGERRS.$$$ containing compilation error messages is written by the compiler. If RED is then invoked, it will see PROGERRS.$$$ and use it as a guide to pinpoint and diagnose errors in the erroneous source file. See section 5.1.5 of this appendix for more detailed information on these new features of RED.

5.1 How To Install RED

This section tells you how to get RED up and running on your system, assuming you don't already have a functioning copy of RED available. Before you read on, though, make sure you read the "read.me" file, which you will find on one of the distribution disks. This file contains important information—additional details and clarifications as well as tips and warnings that are unique to the BDS C implementation of RED.

5.1.1 Run the Configuration Program

If you have a version of RED that is ready to run on your computer, you can safely skip the rest of this section. Also, if you haven't yet read the READ.ME file, you should certainly do so right now. Otherwise, you might well waste lots of time following inappropriate directions.

The next thing to do is to run the configuration program, RCONFIG. The executable version of RCONFIG is on the file RCONFIG.COM. The purpose of RCONFIG is to create two source files that describe your terminal and how you want RED to work.

While running RCONFIG you can use the normal CP/M line editing functions to correct mistakes. In other words, use control-h (also known as backspace) to erase one character and use control-x to erase an entire line. You can leave RCONFIG at any time by hitting control-c. By

the way, if you see that you have answered any question incorrectly, don't worry. RCONFIG always lets you revise your answer to any question later on.

5.1.1.1 Setting Defaults

RCONFIG first asks some questions about how you want to use RED. The first question asks whether the prompt line should tell what column the cursor is on. This column number changes any time the cursor moves right or left, so some flicker might show up on some screens. I recommend that you answer "yes" (or "y") to this question unless you find, after using RED a while, that updating the column numbers creates a problem.

The second question asks whether line wrapping will be enabled or disabled when RED starts up. What do I mean by line wrapping you ask? When line wrapping is ON, a new line is created automatically whenever a character is inserted with the cursor at the end of the screen (as long as no characters appear to the right of the cursor). When line wrapping is OFF, no new line is created and RED will not let you insert anything once the cursor bumps up against the right edge of the screen.

The answer you give at this point just sets the way RED works by default initially. You can always use the wrap or nowrap commands from within RED to change how RED works. I recommend that you turn line wrapping on by default unless you find, after experimenting with RED a while, that you want line wrapping to be off most of the time.

The next set of questions asks you how you want RED's modes to work. What mode or modes do you wish RED to be in by default? When exactly do you want RED to change from one mode to another? It turns out that people have very strong and persistent opinions on this subject.

RED can shift <u>automatically</u> from one mode to another in three different situations: 1) when a command finishes, 2) when the cursor moves from one line to another and 3) when a new line is created. RCONFIG asks you about each of these three cases. Different answers will result in different "styles" of using RED. The next two paragraphs illustrate some possibilities.

Say you want RED always to be in overtype mode unless you explicitly indicate otherwise. You should then have RED enter overtype mode in all three situations. Insert and edit modes will not intrude on you work, but will be available if you decide you want them. On the other hand, if you prefer insert mode to overtype mode, you can have RED switch to insert mode in all cases.

I myself use RED as follows: RED switches to edit mode after commands and when the cursor moves from one line to another, but RED switches to insert mode when a new line is created. Thus, RED is always in edit mode unless I am inserting text, in which case RED will be in insert mode. This is a more complicated style of using RED, but it works very well for me.

RCONFIG asks the following three questions. In each case, you answer 'e' for edit, 'o' for overtype or 'i' for insert. 1) What mode do you want RED to be in after commands finish? By the way, this question also determines which mode RED starts in. 2) What mode do you want RED to be when the cursor is moved up or down one line? 3) What mode do you want RED to be when a new line is created?

Please be clear that your answers to these three questions do not lock you in to a particular style of using RED—they only set defaults which may be overridden easily while RED is running.

Thus, there is no need to agonize over them; just make an educated guess about what you might prefer. After playing with RED you can always reset the defaults by rerunning RCONFIG and recompiling RED.

5.1.1.2 Selecting Control Keys

RCONFIG then asks how you want to set up RED's control keys. In other words, you will indicate which control keys on your keyboard will do what functions. RCONFIG asks a series of questions—one for each function to be performed. Answer each question by typing either a decimal number or a carriage return. If you type a carriage return, RCONFIG will use a default value indicated in parentheses. Otherwise, RCONFIG will use the key whose ascii code is the same as the decimal number you typed. Control keys have decimal values between 1 and 32. DEL, with decimal value of 127 can also be used. Avoid values greater than 127 or less than zero. Answering zero to any of these questions makes that function unavailable.

For example, some people like to have the carriage return key split the current line instead of just creating a new line. I find that style of operation to be a nuisance, but Word Star acts this way and some people prefer it. To make RED work this way just assign the split function key to be carriage return. You should then pick some other key to be the insert down function key or suppress that function altogether by assigning zero to it.

5.1.1.3 Describing Your Terminal

Next, RCONFIG finds out what your terminal can do—RCONFIG asks a series of questions about what built-in functions your terminal has. Answer each question with a yes or a no. You may use the letters 'y' or 'n' for yes or no.

Next, RCONFIG determines just how your terminal works. You will supply the character sequence for each of your terminal's built-in functions. For example, let us consider the goto x,y function, which is the only function that your terminal <u>must</u> have.

As a simple example, suppose that the way to move the cursor to ROW y and COLUMN x is send the escape character (27) to your terminal, followed by an equal sign, followed by 32 plus x, followed by 32 plus y. You would type the following in response to RCONFIG's questions:

```
Enter byte 1: 27
Enter byte 2: '='
Enter byte 3: x+32
Enter byte 4: y+32
Enter byte 5: (return)
```

Note that you type a carriage return to stop entering bytes.

Be **sure** that you enter the row and column numbers in the order that your terminal expects. Many terminals use a sequence in which the row and column numbers are reversed from the sequence shown above. If you do make a mistake in the goto x y sequence the screen will not look at all like it should when you run RED in step 4.

ASCII terminals furnish a more complex example. Such terminals require that the x and y coordinates be given in "ASCII" notation. For example, to move to column 5 and row 21 something like the following would have to be sent to the screen:

```
ESC 'C' '0' '0' '5' 'R' '0' '2' '1'
```

The point is that the digits '0' '5' '2' and '1' must be ASCII, not binary. Here is the way to generate such sequences:

```
Enter byte  1:  27
Enter byte  2:  'C'
Enter byte  3:  '0'+(x/100)
Enter byte  4:  '0'+(x/10)
Enter byte  5:  '0'+(x%10)
Enter byte  6:  'R'
Enter byte  7:  '0'+(y/100)
Enter byte  8:  '0'+(y/10)
Enter byte  9:  '0'+(y%10)
Enter byte 10:  (return)
```

As you can probably tell, you are in effect generating the ASCII digits using the arithmetic operators of the C language. You may use any legal C expression containing only constants or the variables x and y.

Finally, RCONFIG asks you if you are ready to create two files, RED1.H and RED6.C. If you answer yes, RCONFIG creates new versions of both files, erasing previous versions of the files if they exist. If you answer no, RCONFIG exits without changing the files or making any other changes.

5.1.2 Compile and link RED

Now that RCONFIG has created the files RED6.C and RED1.H, all of RED's source files are ready. It's time to create RED! All source files must be compiled and the resulting object files must be linked together to produce an executable version of RED. Compile RED using RED.SUB and link RED using RLINK.SUB.

WARNING: check the submit files before you use them to make sure you will have enough room on your disks for any temporary files that might be needed Change the submit files if required so that temporary files will be written to scratch disks having enough free space on them.

WARNING for BDS C users: be aware that RED is just on the verge of having too many functions for the CLINK linker to handle. I recommend using the L2 linker if at all possible. Not only can L2 handle a large number of functions but L2 produces shorter code. Note that L2 uses different command-line options than CLINK. Thus, you will have to modify the RLINK.SUB file to work with L2.

If you get an error during this step you may have made a mistake when you typed in the sequences of bytes for your terminal's built-in functions. Such an error will show up when compiling RED6.C. Rerun RCONFIG or modify RED6.C directly. Something is wrong with the files on your disk if you get any other error. Create a new working disk from your master disk and start again with step 2.

5.1.3 Test and use RED

You are now ready to run RED. It should clear the screen, draw the prompt line at the top of the screen and tell you what version you are using. If that doesn't happen, you probably made a mistake during step 2. Go on to step 5 for help.

If the screen looks reasonable, you are ready to start learning about RED. Run RED while reading the next chapter. Happy editing!

5.1.4 (Optional) Run STEST

You do not have to do this step, but it should help if RED doesn't draw the screen properly. Create a working version of STEST by compiling STEST.C and linking the resulting object file with the object file created from RED6.C. Remember that you must recreate STEST every time you change RED6.C.

Compile STEST.C using STEST.SUB and link STEST.CRL with RED6.CDL using SLINK.SUB.

Now run STEST. It prints test patterns on your screen and tells you what those test patterns should look like. If and when a test pattern doesn't look like it should, STEST tells you what part of RED6.C is suspect.

Armed with this information, go back to step 2 and rerun RCONFIG. Next, recompile just the one file RED6.C using RED1.SUB and recreate and run STEST. When that works properly, link RED as in step 3 and test RED again.

5.1.5 Additional Features for RED Under BDS C v1.6

The additions to RED that the user will notice (above and beyond original RED capabilities as outlined in the .DOC files) are as follows:

1. If the file PROGERRS.$$$ (generated by CC.COM under the appropriate conditions) exists when RED is entered, it will print a message on the prompt line. Hit any character to continue.

2. If the file PROGERRS.$$$ exists and no file was specified on the command line, RED will automatically load the file named on the first line of the PROGERRS.$$$ file. Otherwise, the file named on the command line is loaded.

3. Pressing > in edit mode (or ESC > in other modes) displays the error message of the next line of the error file on the command line. Hit any key to move to the offending line.

4. Pressing < in edit mode (or ESC < in other modes) displays the previous error message and move to that line after any key is pressed.

5. RED knows enough to ignore error lines in PROGERRS.$$$ that do not pertain to the file being edited. However, you can fool RED if you change the current files name using the "name" command.

6. RED knows enough to adjust line numbers properly for inserting, deleting, moving and copying lines.

7. The "cc" command has been added. This will automatically exit RED and will invoke CC <filename> where <filename> is the name shown on the command line. At present there are no provisions for additional arguments to CC.

8. With ERR_CMND NOT defined, there are exactly 255 functions in the link. This makes it possible to link red with CLINK.COM. If you add any more functions, or define ERR_CMND, then RED will contain more than 255 functions and you can only use L2 to link it.

9. If enabled (which it is NOT at present to get under 255 functions, as stated above) the "errors" command will list the first 20 or so (depending on screen size) lines of the PROGERRS.$$$ file.

5.2 Reference Manual

This section tells you how to use RED—it describes RED's commands and functions, tells how to use them and explains what to do about warning messages.

Each section discusses a particular activity or task that you do while creating, changing or saving a document. The table of contents at the beginning of this book will help you locate the correct section quickly. Consult the index at the end of this chapter to find complete information about a particular command, function key, control key, mode or error message. Also, you will find a summary of RED's operations on the back cover of this manual.

Starting RED

It is time to begin using RED! When RED starts up it does the following:

1. RED clears the screen and prints a welcoming message. This sign-on message tells you what version of RED you are using and how to print help messages. Help messages are simply reminders of what you can do with RED.

2. RED draws the prompt line at the top of the screen. For now, just notice this top line; we'll discuss the information on it in a moment.

3. RED puts the cursor just below the prompt line. The cursor is a distinctive character on the screen. (On most video terminals the cursor is shown as a box or underline which blinks.)

4. RED draws the end-of-file marker on the third line of the screen. This line looks like:

```
------------ End of File. ------------
```

Initially, most of the screen is blank because RED's buffer, or internal memory doesn't contain any information. You can think of the screen as a window into part of this buffer. As you make additions, corrections and deletions to the buffer, those changes appear automatically on the screen. The purpose of the end-of-file marker is to make absolutely clear what the buffer does and does not contain.

Let's look again at the prompt line. At the far left, you will see that the it says,

```
line: 1     column: 0
```

These two fields indicate which line in the buffer and column on the screen that the cursor is on. The next field says,

```
..no file..
```

indicating that the buffer does not contain any information from a file.

Finally, the prompt line tells you what mode RED is in—either edit mode, insert mode or overtype modes. In most respects, RED works exactly the same regardless of mode; that makes RED simple to use. However, some details of how RED works change depending on mode; that makes RED powerful. We'll see later that not only can you make RED change modes easily, but you can have RED change modes automatically if you so desire. This feature is very important—it allows you to make RED work exactly the way you think it should. We'll discuss modes shortly in complete detail. For the moment, just notice what mode RED is now in.

At your option, you may have RED automatically load a text file into RED's internal memory (the buffer) when RED initially starts up. For example, if you had invoked RED as follows:

```
A>red document.txt
```

then RED would have loaded the file document.txt into the buffer already. In that case the screen would not be blank but instead would show you the first several lines in the file and the prompt line would "document.txt" instead of "..no file.."

Using Function and Control Keys

The term function key refers to a key on your keyboard that does one and only one action or function. Just about everything you do with RED involves using function keys—they are used to change modes, to insert or delete lines and characters, to move the cursor, to split and join lines and to start commands. There is also a "repeat" function key that repeats the previous function. All function keys can be used in insert, overtype and edit modes and all function keys do the same thing, regardless of the current mode.

RED needs to be able to distinguish function keys from what you are typing into the buffer. Thus, function keys must be assigned to control keys on your keyboard. A control key is typed by holding down the key marked CTRL on your keyboard while typing another key. For example, you type the "control c key" (abbreviated control-c) by typing the letter c while holding down the CTRL key.

A decision was made when RED was created (that is, when the CONFIG program was run) which control key is assigned to each function key. For each function key, there is a default assignment of a particular control key. This is the assignment that is assumed in this chapter. Throughout this chapter, the name of each function key is followed in parenthesis by the control key assigned to it by default. For example, this chapter refers to the split function key as split (control-s). So in order to press the split function key you must actually press the control-s key on your keyboard. Clear?

Changing Modes

I said earlier that RED has three modes: edit mode, insert mode and overtype mode and that all function keys act the same regardless of mode. Thus, the only difference between the three modes is what happens when you type a non-control character. You use three different keys to switch RED between modes—the enter iNsert (control-n), enter overType (control-t) and enter edit (control-e) function keys.

Besides these three keys which explicitly change from one mode to another, RED can change from one mode to another automatically in three situations:

1) after every command
2) after inserting new lines and
3) whenever the cursor moves up or down one line.

What RED does initially in these three cases was chosen back when RED was configured (see Chapter 1), so I can't be specific about what your copy of RED will do. For the moment, just be aware of what does happen in each case.

You can change how RED switches modes using three sets of commands: def0edit, def0ins, def0over, def1edit, def1ins, def1over, def2edit, def2ins, and def2over. (We haven't discussed commands yet, so if you are reading this for the first time just realize that how RED switches between modes isn't carved in stone.)

For example, to make RED into an "overtype mode editor" just issue the def0over, def1over and def2over commands. You'll never see insert or edit modes again unless you switch to them explicitly. As another example, I prefer to use a hybrid combination of edit mode and insert mode—I configure RED so it acts as if I had issued def0edit, def1ins and def2edit commands. Try it. You may like it.

Inserting Characters With Insert and Overtype Modes

In this section we'll look at insert and overtype modes, leaving edit mode for much later. Let's discuss insert mode first, so if RED is not already in insert mode press the enter insert (control-n) function key. Notice that the prompt line indicates that the mode has changed.

In insert mode, any plain (i.e., non-control) character you type is inserted into the buffer without replacing any other information. Characters to the right of the cursor "move over" to allow room for the new character. To jump the gun a bit, you can make the cursor move left without erasing anything by hitting the left (control-l) function key. Try the following: insert a few characters, move the cursor left once or twice and insert some more characters.

Overtype mode works just like insert mode except that a character directly under of the cursor is replaced by what you type, instead of moving to the right. In other words, in overtype mode you "type over" whatever is already be on the line. Compare overtype mode to insert mode: enter overtype mode, type some characters, move the cursor to the left and type some more characters.

Inserting New Lines

You can't edit much if you are confined to a single line. You end one line and begin another using the insert down (carriage return or control-m) function key. Try it. The insert up (line feed or control-j) function key is a companion key to the insert down key. The insert up key inserts a blank line above the current line.

The insert up (line feed) and insert down (carriage return) function keys may also cause RED to shift automatically to a different mode. Which mode RED shifts to after hitting these keys may be changed at any time using the def1edit, def1ins and def1over commands. For example, the def1edit command causes RED to shift automatically to edit mode whenever the insert down or insert up function key is pressed.

Notice that RED will split the line automatically if the cursor reaches the end of screen while you are inserting characters. This feature is called line wrapping. Try it out. Notice also that line wrapping never happens if there are characters to the right of the cursor.

Play around with RED right now. See what happens when you insert new lines. Does RED switch modes? Don't worry about typos; in the next several sections we'll see how to deal with them.

Moving The Cursor

In order to change your text, you must position the cursor near the text to be changed. This section tells you how to do that.

The right (control-r) and left (control-l) function keys move the cursor right or left one column. However, these keys always leave the cursor on the same line. For example, nothing happens if you hit the left key when the cursor is at the leftmost column of the screen.

The up (control-u) and down (control-d) function keys move the cursor up and down one line respectively. The cursor will not move above the first line or below the last line of the file.

The up (control-u) and down (control-d) function keys may also cause RED to shift to a different mode. Which mode RED will shift to may be changed at any time using the def2edit, def2ins and def2over commands. For example, the def2ins command causes RED to shift to insert mode whenever the up (control-u) or down (control-d) function key is pressed.

The page up (control-q) and page down (control-p) function keys move the cursor up or down one page of the file. You need not wait for the screen to be completely redrawn before hitting another character.

The scroll up (control-w) and scroll down (control-o) function keys scroll the cursor up or down. Hitting any key interrupts the scrolling.

The word forward (control-f) and word backward (control-b) function keys move the cursor forward or backward one word. A word is any sequence of characters separated by end-of-line, blank or tabs.

Deleting Characters and Lines

A large part of my writing involves deleting characters and lines: two words forward and one word (taken) back—two sentences written and one erased. RED lets you do this without any fuss.

To delete a single character you must first position the cursor either directly over the character or just to its right. The delete left (control-h or backspace) function key deletes the character to the left of the cursor. Nothing happens if the cursor is up against the left edge of the screen. The delete under (del) function key deletes the character directly under the cursor.

Use the delete line (control-z) function key to delete the entire line on which the cursor rests. The screen is redrawn with the line squeezed out.

Undoing Mistakes

Sometimes RED lets us work faster than our thoughts—or maybe our fingers have a mind of their own. In any case, there is occasionally a need for undoing the "improvements" that have just been visited upon a line.

The undo (control-x) function key restores a line to what it was when the cursor last moved to the line. In other words, the undo function undoes whatever editing or inserting you have done on the current line. Several words of warning: you can not use the undo (control-x) function key to restore a line that has been erased with the erase line (control-z) function key. Also, you can not use the undo (control-x) function key to undo a change once you have moved the cursor to another line.

Splitting and Joining Lines

Being able to split a line into two pieces or make one line from two is often very handy. For instance, to edit a line longer than will fit on the screen, you would first split the line, then make your corrections and finally glue the line back together again.

The split (control-s) function key splits the current line into two pieces. Everything to the left of the cursor stays right where it is. All other characters are moved to a new blank line created below the original line. The split (control-s) function key acts just like the insert down (carriage return) function key if the cursor is positioned at the right end of the line.

The glue (control-g) function key combines two lines into one. This key appends the current line to the line above it and then deletes the lower line. The new line is allowed to be longer than the width of the screen.

Inserting Control Characters

In rare cases, it is desirable to insert control characters into the buffer. This requires a special function key. The verbatim (control-v) function key enters the next key pressed into the buffer, no matter what it is. For example, to insert a control-s into a buffer, type control-v control-s. After you press the verbatim (control-v) function key, but before you press the second key, the prompt line says 'verbatim'.

Repeating the Previous Function

The repeat (control-a) function key repeats the last function key, edit mode function or escape sequence. For example, typing control-p control-a is the same as typing control-p twice. As we will see, using the repeat key can sometimes save you typing.

The repeat key "amplifies" the effect of several functions as shown in this table:

```
original        amplified
function      function

begin line      home
end line        end page
home            page up
end page        page down
```

For example, typing ESC b ^a ^a is the same as typing ESC b ESC h ^q because the first ^a amplifies the begin line function (^b) into the home function (ESC h) and the second ^a amplifies the home function into the page up function (^q).

Using Commands

Up until now, we have been talking about functions, i.e., operations that can are done by pressing a single function key. However, functions are not appropriate in all situations—they might require additional information or they might potentially alter too much work to be safe.

Commands are RED's way of performing complex or dangerous operations quickly and safely. You start each command with the enter command (control-c) function key. Try this key out now. Notice that the cursor moves to the prompt line. All commands end with a carriage return, and if you type nothing but a carriage return the command is terminated. You can also exit from a command by hitting either the enter edit (control-e), the enter insert (control-n) or the enter overtype (control-o) function key. OK, exit the command in one of the ways just mentioned.

If you make a mistake while entering a command, just hit control-h (also known as backspace) to erase single characters. You may use either upper or lower case for commands.

Depending on how RED was configured, RED may shift to a different mode after each command. At any time, you may change which mode RED will shift to by using the def0edit, def0ins and def0over commands. For example, the def0over command causes RED to shift to overtype mode after each command.

The following several sections discuss RED's various commands in detail.

Creating, Saving and Loading Files

After you have finished working on your document you must save it on a file. This is a two-step process: you must name the file and you must actually save your work to that file.

Use the name command to name your file. Just type "name" (you don't type the double quotes) followed by the name you want your file to have, followed (as always) by a carriage return. Notice that the prompt line changes to reflect the new file name.

Aside: Your file name can have no more than eight letters or digits, followed optionally by a period and no more than three more letters. The question mark (?) and asterisk or star (*) are not allowed in file names. Examples:

```
legal           illegal
names         names

abc             ???.abc
foo.bar         foo.*
letter.doc      letter.doc1
12345678.doc    123456789.doc
xy.z            x.y.z
```

The last step in creating a new file is writing your work to the file. If you don't do this your work will be lost, but don't worry, RED reminds you if you haven't done so when you try to leave. To save your work, use the save command. While the save command is in progress, the message "—saving—" appears on the prompt line. The save command doesn't take any arguments; your work is saved to the file named on the prompt line. If you issue the save command when the prompt line indicates "..no file.." RED complains saying, "file not named". Hit any key to clear this message and continue.

If RED says "file exists" instead of "—saving—" it means that a file already exists on the disk with the name shown on the prompt line. You now have two choices: you can pick another name for your file and do the save command over again or you can use the resave command to replace what is already on that file with your present work. (Watch out: the resave commands destroys the previous contents of the file.) If you use the resave command and the file does not already exist, RED gives you the "file not found" message. As with the save command, the resave command never takes any arguments.

As mentioned earlier, you can load an already existing file (say memo) at the same time you start RED by typing 'red memo'. If RED finds the file on the disk, RED loads that file and updates the prompt line to indicate the name of the file. This is the file name used by the save and resave commands. (Of course, you can use the name command at any time to change this name.) If the file is not found the prompt line says "file not found." As always, hit a carriage return to clear this warning message.

If you did not give a file name when you started RED, or if you got the "file not found" message, you can use the load command to load a file into the buffer. The load command takes one

argument—the name of the file to be loaded. As you would expect, the load command changes the file name on the prompt line so that the save and resave commands will update the file you just loaded. Purely as a convenience, RED treats the red command just like the load command. Examples:

```
load abc.doc
red memo
save
resave
```

Unlike some other editors, RED's load command does not create a file if it does not exist, so you haven't created any unwanted file if you don't get the name right. Neither does the load command change the file name on the prompt line if the file does not exist. This feature makes the save and resave commands safe to use in almost all circumstances.

The load command replaces whatever is in the buffer by the contents of the file being loaded. For your protection, the load command asks "Buffer not saved, proceed?" if loading the file might destroy unsaved work. If you answer 'y' the load operation begins and whatever is in the buffer is lost. Otherwise, the load command terminates and you have an opportunity to save your work.

Leaving RED

There are two ways to leave RED. The first is the exit command, which takes no arguments. For your protection, RED asks "Buffer not saved, proceed?" if you issue this command before you have saved your work. Type 'y' to exit anyway or type anything else to cancel the command.

The quit command may or may not be available with your version of RED. If it is available, the quit command works like the exit command (it takes no arguments), except that RED saves information on your disk so RED can reload the file you were working on quickly and automatically. When RED is next restarted, it looks for this information to resume editing right where you left off.

The quit command is nice to have if you do a lot of work with a single file because it saves 99% of the time it takes to load a file with the load command. However, the quit command does have some drawbacks. First, the saved information (the work file) takes up space on the disk when RED is not being used. Second, if the work file is erased, some of your work may be lost. Third, if you interrupt RED by hitting your computer's reset key, the work file will not have the proper file status line. The next time you start RED, RED will complain and you will have to erase the work file by hand.

Searching for Patterns

As your file becomes longer and longer, it becomes harder and harder to find the parts of it that you want to change. Instead of searching for words or phrases yourself, RED can do the searching for you.

In order to do searching, you must specify patterns which tell RED what to look for. A pattern is simply any string of characters ended by a carriage return. Most letters in patterns just stand for themselves. Examples:

- The pattern 'abc' matches the three letters 'a' 'b' and 'c'.
- The pattern '12-4' matches the four characters '1' '2' '-' and '4'.

There are three characters which have special meanings within patterns and make patterns more powerful. A question mark in a pattern matches any character at all. Examples:

- The pattern '?bc' matches any 'bc' that is not the first character on a line.
- The pattern 'a?c' matches an 'a' and 'c' with exactly one character between them.
- The pattern '???' matches any three characters on the same line.

A leading caret ('^'), i.e., a caret that appears at the start of a pattern, matches the start of a line. Examples:

- The pattern '^abc' matches any line that starts with 'abc'.
- The pattern '^??abc' matches any line with 'abc' starting in column 3.

A caret that does not start a pattern loses its special meaning. Examples:

- The pattern '^?^a' matches any line with a '^' in column 2 followed by an 'a'.
- The pattern '?^' matches any '^' which is not in column 1.

A trailing dollar sign, i.e., a dollar sign that appears at the end of a pattern, matches the end of a line. Examples:

- The pattern 'abc$' matches any line that ends with 'abc'.
- The pattern 'abc??$' matches any line with exactly two characters after 'abc'.

A dollar sign that does not conclude a pattern loses its special meaning. Example:

- The pattern '^?$?' matches any line with '$' in column 2 followed by some other character.

A leading caret and trailing dollar sign may be used in the same pattern. Examples:

- The pattern '^abc$' matches any line that contains only 'abc'.
- The pattern '^?$' matches any line with exactly one character.

The find command puts the cursor at the start of the pattern when the pattern is found. You invoke the find command as you would expect: type the enter command (control-c) function key followed by find <CR>. The prompt line will now ask you for a search mask. This means that you should enter a pattern to search for. Type in the pattern and end it with a carriage return.

The find command will now search from the place where the cursor is, looking for the pattern. If the find command reaches the end of the buffer without finding a match the search "wraps

around the buffer," i.e., the search continues from the start of the buffer to the line where the search originally commenced. If the find command eventually matches the pattern, RED will place the cursor at the start of the pattern. If no match is found, RED will simply put the cursor back where it was before the find command was invoked.

The findr command works just like the find command except that it searches backwards through the buffer for a pattern.

The search command searches for a pattern just like the find command, but the search may be continued after a pattern is found. When a match is found, the prompt line will say, "next, exit?". If you hit an 'n', the search will continue. The search will end if you hit any other key.

The change command searches for a pattern in a manner similar to the search command, but when a match is found a substitution is made. When you invoke the change command, you will be asked for a search mask, just as with the search command. Next, you will be asked for a change mask. Whenever the pattern specified by the search mask is found, the pattern specified by the change mask is substituted.

Carets and dollar signs have no special significance in a change mask. Question marks in a change mask are replaced by the character that matched the corresponding question mark in the search mask.

For example, suppose the search mask is 'a?b?' and the change mask is 'A??C'. If the characters that match the search mask are 'a+b-' then 'a+b-' would be replaced by 'A+-C' because the two question marks in the change mask would be replaced by '+' and '-' respectively.

The change command does not make all changes in the buffer at once. When a match the substitution is made on the screen and the prompt line asks,

```
"yes, no, all, exit?"
```

The substitution is undone if you reply 'n' or 'e' and the change command terminates if you say 'e'. If you reply 'y', the change is made and the searching continues. If you say 'a', the change is made and searching continues. However, when you say 'a', all changes are made without further prompting and no further changes are shown on the screen. Only the line number field on the prompt line shows that changes are being made.

The find, findr, search and change commands may be stopped at any time by hitting any control character. This is especially important when using the 'a' option of the change command.

The find and findr commands start searching from the current line, but you can change that by invoking these commands with a line number. Examples:

```
command       search starts at
find             current line
find 1           line 1
findr 9999       end of buffer
```

The change and search commands look through the entire buffer for the pattern unless you specify a portion of the buffer to search. Examples:

```
command      what is searched
search          every line
search 1 9999   every line
change 70 90    lines 70--90
search 50       lines 50--end
change 90 9999   lines 90--end
```

Moving Blocks of Lines

One of the most common editing operations is cutting and pasting. RED has four commands that make this easy.

The move command moves a block of lines from one place in the buffer to another. The move command takes three arguments, the first line to move, the last line to move, and the line after which the lines are to be moved. Only one line is moved if only two line numbers are given. Examples:

```
                line(s)     where
command      moved       moved
move 1 2 3      1--2        after line 1
move 1 2 0      1--2        before line 1
move 2 10       2           after line 10
```

The copy command works just like the move command except that a copy of the lines is moved so that the original lines stay where they were. Examples:

```
                line(s)     where
command      copied      copied
copy 1 2 3      1--3        after line 3
copy 1 2 0      1--2        before line 1
copy 1 8        1           after line 8
```

The extract command copies a block of lines to a file without erasing the block from the buffer. Take care with this command: the file is erased if it already exists. Another caution: integers are legal file names, so make sure you include the file name. Examples:

```
                        file-       lines
command               name        written

extract abc             abc         whole file
extract 1 2              1          line 2
extract abc 1 2         abc         lines 1--2
extract f 1 9999         f          whole file
```

The inject command is the companion to the extract command. The inject command adds a file to the buffer. It does not replace the buffer as does the load command. Examples:

```
command              where injected
inject abc              after current line
inject abc 0            before line 1
inject abc 9999         at end of file
inject abc 50           after line 50
```

You can use the extract and inject commands to cut and paste between different files. Extract a block of lines from the first file into a temporary file, load the second file and then inject the lines from the temporary file into the second file.

Setting Tab Stops

Tab stops effect how tabs are shown on the screen and printed on the printer. RED sets tab stops every 8 columns to begin with, but this can be changed with the tabs command. After this command the screen is redrawn so you can see the results of the new tab setting. Examples:

```
command       width of tabs
tabs             8
tabs 8           8
tabs 4           4
```

Enabling and Disabling Line Wrapping

Initially, line wrapping is enabled. Lines are split whenever:

a) the cursor is the last character of the line and
b) the cursor is at the right edge of the screen and
c) a character is inserted.

The nowrap command disables line wrapping. When line wrapping is disabled, no further insertions are allowed in a line when the cursor reaches the right margin of the screen. If you want to enable line wrapping again after using the nowrap command, use the wrap command.

Listing the Buffer

The list command prints the buffer on your printer. Lines are formatted just as they are on the screen, but the length of the print line, not the width of screen, determines where long lines are truncated. You can interrupt the listing at any time by hitting any control key. Examples:

```
command       what is listed
list             the current line
list 15          line 15
list 1 9999      the entire buffer
list 400 500     lines 400--500
```

Deleting Multiple lines

The clear command erases the whole buffer, while the delete command deletes one or more lines. The clear command will caution you if erasing the buffer might cause work to be lost, but the delete command does not, so be careful. Examples:

```
command       what is deleted
clear            the whole file
delete 1 9999    the whole file
delete           the current line
delete 25        line 25
```

Choosing How RED Switches Modes

As mentioned before, RED will automatically switch from one mode to another in three situations:

1) after every command
2) after inserting new lines and
3) whenever the cursor moves up or down one line.

You can choose exactly what RED will do in each case. This section tells how.

The def0edit, def0ins and def0over commands determine which mode (edit, insert or overtype) RED will be in after each command. For example, after the def0edit command is given, RED will switch to edit mode after each command.

The def1edit, def1ins and def1over commands determine which mode RED will be in after the insert up (line feed) or insert down (carriage return) function keys are pressed. For example, after the def1ins command is given, RED will switch to insert mode whenever a new blank line is created.

The def2edit, def2ins and def2over commands determine which mode RED will be in after the up (control-u) or down (control-d) function keys are pressed. For example, after the def2over command is given, RED will switch to overtype mode whenever the up or down function keys are pressed.

Edit Mode Functions And Escape Sequences

Edit mode lets you avoid typing so many control keys. In edit mode, typing regular (i.e., non-control) keys makes RED act as though function keys were pressed. Some people find using normal characters in this way confusing, which is why this section has been left until now. Other people, myself for instance, think that edit mode is a great convenience.

Escape sequences are an added frill; they are a way of executing edit mode functions without switching to edit mode. Escape sequences consist of the escape (ESC) function key followed by an edit mode function. For example, the 'h' edit mode function homes the cursor to the top left corner of the screen. If RED were in insert mode I could home the cursor using the ESC h escape sequence without having to switch RED to edit mode first. By the way, edit mode commands and escape sequences may be typed either in upper case or in lower case.

Here is a list of the edit mode functions. For simplicity's sake the functions are listed in alphabetical order. Again, each function may be used outside of edit mode by using an escape sequence.

The space bar moves the cursor right one column. Nothing happens if the cursor is up against the right edge of the screen. In other words, the space bar works exactly the same as the right (control-r) function key.

The '+' key moves the cursor down a half a page. The '-' key moves the cursor up a half page.

The b key puts the cursor at the beginning (left hand edge) of the line. This key is amplified by the repeat key into the home function. For example, typing b ^a in edit mode is the same as typing b p. Similarly, typing ESC b ^a in insert mode is the same as typing ESC b ESC p.

The d key causes the cursor to move down rapidly. Type any key to stop the scrolling.

The e key moves the cursor to the right end of the line. This key is amplified by the repeat key into the end page function. For example, typing e ^a in edit mode does the same thing as typing e z. Similarly, typing ESC e ^a is the same as typing ESC e ESC z.

The g key moves the cursor to another line. After you type the g the cursor will move to the prompt line. The prompt line will show 'goto:' Now type a line number followed by a carriage return. By the way, there is a g command that works the same way.

The h key homes the cursor to the top left corner of the screen. This key is amplified by the repeat key into the page up function. Thus, typing h ^a in edit mode is the same as typing h q. Similarly, typing ESC h ^a works the same as typing ESC h ESC q.

The k key deletes all characters from the cursor up to but not including the word that starts with a "search character". Everything from the cursor to the end of the line is deleted if no word starts with the search character. After you hit the k the prompt line displays 'kill'. Now type the search character. If you wish to cancel the k command before specifying the search character, press any control character. The k command will be stopped and no deletion will be made. If too much text is deleted, use the undo key and try again. Example: Typing k <space> deletes the following word.

The m key moves the cursor to the start of the line in the middle of the screen.

The p key moves the cursor down one page. You don't have to wait for the screen to be completely redrawn before you hit another key. Thus, hitting several p keys is a very fast way to move short distances. The q key is the companion to the p key. It moves the cursor up one page.

The s key moves the cursor to the next word that starts with a search character. If no word starts with the search character the cursor is moved to the end of the current line. After you hit the s key, the prompt line displays 'search'. Now type the search character. Example: Typing s <space> moves the cursor right one word.

The u key moves the cursor up rapidly. You stop the scrolling by typing any key.

The x key replaces the character under the cursor. After you type the x command the prompt line displays 'eXchange'. Now type the new character. If you hit a control character no change is made and the x command is canceled.

The z key moves the cursor to the start of the last line on the screen. This key is amplified by the repeat key into the screen down function. For example, typing z ^a in edit mode is the same as typing z p. Similarly, typing ESC z ^a is the same as typing ESC z ESC p.

What To Do About Error Messages

RED will print a message on the prompt line should anything go amiss. You clear the message by hitting any key. Usually the message will be a reminder about how to enter a command. For example, if you forget how to use the move command, just do the move command anyway. RED will say,

```
usage: move <block> <n>
```

This may jog your memory enough so you won't have to look up the command in this chapter.

The only serious error you might see is:

```
write error: disk or directory full??
```

This error usually means that RED could not complete a save or resave command because there was not enough room on the disk. This error is not too serious; you should be able to recover from this error if you do a save or resave to another disk. But you should never remove the disk invoked RED! (That disk contains the work file, which is what you are trying to save.)

More rarely you can see this error when you are making insertions into the buffer. When this happens, try to save the buffer to a new file on another disk. Once again, do not remove the disk from which you invoked RED! This may or may not work, depending on whether another disk is available. Thus, you may lose the work you have done since your last save or resave. Obviously this is not a pleasant occurrence. You can avoid this problem by making sure that your disk has enough room to hold your work and by frequently saving your work.

Chapter 6

CDB: A Debugger for BDS C

Version 1.60
1 October 1986

David Kirkland
3766 Purdue
Houston, Texas 77005
(713) 660-9151 (home)
(713) 229-1101 (office)

Copyright (c) 1982-1986 by David Kirkland

6.1 An Explanation of CDB Components

CDB is an interactive symbolic debugger for programs written for the BD Software C Compiler. CDB enables a user to set breakpoints in a program, to trace the flow of program execution, and symbolically to display and set variables. It thus provides the developer of an application program with what I hope is a useful environment for program development and testing.

The debugging package consists of three executable files. The first of these three, L2.COM, is a linker for object code files in the C relocatable (.CRL) format. L2 prepares a .COM file to be loaded and executed under the control of the other parts of the debugging package and also prepares a symbol table for the package's use.

The second element of the package, CDB.COM, is used by the program developer (the "user") to invoke the debugger. CDB interprets the command-line arguments entered by the user, prepares various in-memory data tables, and invokes CDB2.OVL.

CDB2.OVL (CDB2 for short) is the third and final element of the CDB package. CDB2 resides in high memory immediately below the CP/M BDOS. It loads the program to be debugged (the "target program") at the base of the TPA (the CP/M "transient program area," normally at 0100 hex) and remains co-resident in memory with the target program throughout the debugging session. Once CDB2 has loaded the target program, it passes control to the main routine in the target, and execution begins. Whenever the target program (i) enters a function, (ii) returns from a function, or (iii) encounters the beginning of the compiled code for a C statement, the target passes control to CDB2, which either returns control to the target or stops target execution and prompts the user for a debugger command.

In this document, square brackets [] are used to signify optional elements.

6.2 Constructing the Debugger

Because of various changes that might need to be made to the code of the several components of the debugger, the package is distributed as source code. This part of the documentation describes the steps that must be taken to transform the source code into the three executable files L2.COM, CDB.COM, and CDB2.OVL.

6.2.1 Constructing L2

Because L2 needs no customization, there is no need for each user to prepare his own version. The standard version of L2 supplied with BDS C obtains C.CCC and DEFF*.CRL from the currently logged disk during a linkage operation; to change this, modify the **#define** for the DEF_DRIVE macro as described in L2.C. If you make changes (or correct bugs) in L2, the procedure for creating L2.COM is described in the appendix entitled "The L2 Linker for BDS C".

6.2.2 Constructing CDB2

The distribution disk contains a version of CDB.COM and CDB2.OVL set up for a system with a BDOS at or above D000. Almost all systems with over 60K of RAM should be able to use this version as is. However, this version leaves only 31K for the target program and symbol tables; if your system has its BDOS substantially above D000, you may wish to customize the debugger to give you more memory for the target program; and if your system has its BDOS below D000, you must customize to get a working debugger.

6.2.2.1 The CDBCONFG Utility

To simplify the customization process, a configuration program called CDBCONFG is now provided. If CDBCONFG.COM is not on the distribution diskette, you must first compile and link CDBCONFG.C using cc and either clink or l2 as follows:

```
cc cdbconfg.c
clink cdbconfg  (or)   l2 cdbconfg
```

Once you have a CDBCONFG.COM, all you need do is type

```
cdbconfg [bdos]
submit makecdb
```

CDBCONFG creates a submit file called MAKECDB.SUB in the current user number on the currently logged disk that, when submitted, will compile and link CDB.COM and CDB2.OVL. CDBCONFG also edits the CDB.H file to set the #define for CDB2ADDR to the appropriate address for the target system.

If the bdos option is given to CDBCONFG (as a hex number with no leading 0x), CDB is configured for a system with a BDOS beginning at the specified address. If no parameter is given, CDBCONFG uses the current BDOS location of the system on which CDBCONFG is being run.

6.2.2.2 CDB System Description

CDB2 sits in high memory, above the target program and its stack but below both CP/M's BDOS and CDB2's own stack. The code that makes up CDB2 is a little less than 0x4900 bytes long; the externals are about 0x0980 bytes. I have decided, a bit arbitrarily but after some analysis, that the CDB2 stack (which starts immediately below the BDOS) should be allocated about 0x0480 bytes. I hope this is cautious, but because it is possible to create complex expressions that must be recursively parsed to dump symbolically variable contents, I think discretion is wise. Adding the numbers up, we get a total of 0x5700 bytes for the code, globals, and stack for CDB2; thus, CDB2 must start 0x5700 bytes below the start of the BDOS. Because my BDOS starts at 0xE406, my CDB2 sits at 0x8d00 (and I will use this value in the examples that follow).

Once you have decided where to put CDB2, you must edit CDB.H and change the #define for CDB2ADDR to the value you have determined. Below, I will use "CDB2ADDR" to refer to this value.

Constructing CDB

After changing CDB2ADDR in CDB.H, you are ready to compile the two source files for CDB:

```
cc cdb.c -e3c00
cc build.c -e3c00
l2 cdb build
```

Constructing CDB2

To compile the source files for CDB2, we need to know the address of the CDB2 externals. Since the externals are placed right after the CDB2 code, we merely add 0x4900 (the size of the code, given above) to CBD2ADDR. In my case, the result is 0xd600; thus, to compile CDB2 for my system, I must specify "-ed600" as an option.

CDB2 is composed of seven C source files; to compile them, you must enter:

```
cc cdb2.c -exxxx
cc atbreak.c -exxxx
cc break.c -exxxx
cc command.c -exxxx
cc print.c -exxxx
cc parse.c -exxxx
cc util.c -exxxx
```

Once the C files are compiled, you need to assemble the one assembler source file, DASM.CSM (see the CASM Appendix for details on the .CSM assembly language format). The sequence for creating DASM.CRL from DASM.CSM is as follows:

```
casm dasm
asm dasm
cload dasm
```

To save you this step (especially if you don't have a compiled version of CASM.COM handy) the distribution disk contains a pre-assembled DASM.CRL.

The final file to be created is an empty file called NULL.SYM, which L2 will try to use to determine the location of all the functions used in the root segment for which CDB2.OVL will be the overlay segment. Because there is no such root segment, there are no functions, either; but L2 requires a root name if the –ovl option is used, so we create an empty file to please the linker by issuing

```
save 0 null.sym
```

Now that all the .CRL files are ready, we are ready to link them. The proper command is

```
l2 cdb2 dasm atbreak command break
            print parse util -ovl null yyyy -wa
```

where yyyy should be replaced with the value computed for CDB2ADDR, in hex.

Before configuring the debugger, you may want to edit CDB.H and change the **#define** for CDB2_DRIVE; this specifies the drive from which CDB2.OVL will be loaded if the user does not override the default with the **–d** option to CDB. You may specify either a drive letter (with or without the colon), such as "A" or "B:", a user number prefix, such as "10/", or a drive letter with a user number prefix, such as "0/A". As distributed, the default is no drive designator, which will cause CDB2 to be loaded from the currently logged drive and user area.

One note: It is vital that CDB.COM, CDB2.OVL, and the target .COM file all be linked using the same C.CCC (the run-time package) and DEFF?.CRL libraries. This need arises because CDB2.OVL uses the runtime package of CDB.COM when CDB2 initializes itself, and then uses the runtimes from the target once the target is swapped in. If there is a mismatch, at some point in the startup process the debugger will fail miserably.

Changing the restart number

As distributed, the debugger package uses the RST 6 (restart 6) instruction to generate breakpoints. Whenever the RST 6 instruction is encountered, control is transferred to location 0x0030. In some systems, this area of memory (or the RST 6 instruction itself) may be reserved for other use. If so, it is necessary to assign some other restart number to the breakpoint function. Any restart number from one to seven (inclusive) may be used; restart zero is not allowed. To change the restart number, changes must be made to L2.C, CDB.H, and DASM.CSM.

In both L2.C and CDB.H, the **#define** for RST_NUM should be changed to the restart slot the user has assigned to the debugger. In DASM.CSM, the "EQU" for RstNum should be changed to the same value. Note that the value should be specified as a number from 1 to 7.

Finally, when the target program is compiled (with the **–k** option) it is necessary to specify the new restart number. Type **–kn**, where **n** is the new restart number, instead of the usual **–k**.

6.3 How to Invoke the Debugger

In order to use the debugger, the user must first compile and link the target program, and then invoke the debugger itself. This part describes that process. As an aid to the understanding of parts III and IV of this document, part VII below is an example of a debugging session.

6.3.1 Compilation: The –K Option of CC

As documented in the BDS C User's Guide, the **–k** option is used to cause the compiler to (i) generate a symbol table with the extension .CDB and (ii) generate restart instructions in the compiled code. The user issues the cc command as with any other compile, and adds the **–k** option. For example:

```
cc target.c -k
```

6.3.2 Linkage: The –D and –S Options of L2

To link the target program, the user must use the L2 provided with the package instead of CLINK. The following L2 options apply to use with CDB:

–d Create an output module that is compatible with CDB. This option causes L2 to put a restart instruction at the beginning of most functions. Unless overriden by the **–s** option, a restart is placed at the beginning of every function.
except those functions from DEFF*.CRL that are referenced only by functions that are themselves from DEFF*.CRL.

–s CRL files after the **–s** will be treated as "system" library files. A function in a system library file that is referenced only by a function from a system library file will not have an initial restart added by L2, and the debugger will not trace execution into such a function. If the **–s** option is not specified, no files will be treated as system libraries.

–i Specifies that the command line entered is "Incomplete"; this option causes L2 to prompt the user to enter more command line arguments. Each line entered by the user may end with another "-i", in which case the user will again be prompted for more options. Note: the "-i" option must be the LAST option on the command line to work.

–n Just like the CLINK "-n" option. This option makes the resulting COM file preserve the CP/M CCP at run-time, instead of overwriting the CCP with the run-time stack. Programs linked with this option return to the CCP command level without performing a (time-consuming) warm boot.

For example:

```
l2 target -d
```

Invoking CDB

To invoke the debugger, the user enters the CDB command. The command line is of the form:

```
cdb target-name [-l [local_cdbs]] [-g [global_cdbs]]
    [-d [user/][drive[:]] ] [% [target operands]]
```

The **–l** (letter ell) and **–g** options allow the user to specify the .CDB files from which CDB will read symbol tables containing information about the variables used in the target program. target-name.CDB is used if **–l** or **–g** is not specified; although this default is normally adequate, if the target source code is contained in more than one file, the user must provide the names of the .CDB files produced from each of the source files if he wishes to access symbols defined in these files. Often, all the globals are defined in a header (.H) file which is included in each source file; in such a case, there is no need to use the **–g** option, only the **–l** option. With either of these options, if the user enters a zero instead of the file name CDB will not load any symbol files for the specified type of symbol (either local or global). If the user enters no argument at all for either option, CDB will prompt the user to enter file names, one per line, for the symbol files. A null line terminates the prompt.

The "%" operand allows the user to specify arguments to the target program. If the "%" is followed by any operands, these additional operands will be passed directly to the target program; if nothing follows the "%", the user will be prompted for a command line. (Note to hackers: CDB does not pass the arguments that follow the "%" by accessing the "argv" passed to CDB; rather, CDB changes the arguments as they appear in memory at 0x0080, and lets the target program, via C.CCC, parse this command line.)

The **–d** option specifies the drive or user number or both from which CDB2.OVL will be loaded; the default as the package is supplied is the CP/M default drive, but the user can modify this default.

A standard invocation of CDB is:

```
cdb target
```

6.3.3 Summary

To sum this section up, the standard procedure for debugging a program named target.c is as follows:

```
cc target.c -k
l2 target -d
cdb target
```

For a more complex example, assume that FOO.C contains the source for the "MAIN" routine and other functions, and that BAR.C and LIB.C contain source for other needed functions. Both FOO.C and BAR.C contain the same declarations for global variables (both source files **#include** the header file GLOBALS.H), while LIB.C contains the user's library functions that do not access the global variables. Finally, assume that the user has certain other (already debugged) functions in STDLIB.CRL. To compile this mess, the user enters

```
cc foo.c -k
```

cc bar.c –k cc lib.c –k]

To compile this, the user enters

```
cc foo.c -k
cc bar.c -k
cc lib.c -k
```

To link it all together to obtain FOO.COM, the user types

```
l2 foo bar -l lib -s stdlib
```

The **–s** operand tells L2 not to generate function traces into routines included in FOO.COM that were called only by routines in STDLIB.CRL. To invoke the debugger, the user enters

```
cdb foo -l foo bar lib
```

The **–l** operand tells CDB that the files FOO.CDB, BAR.CDB, and LIB.CDB contain symbol table information put out by CC, and that all local symbol information on these files should be loaded. Global symbol information from FOO.CDB is loaded.

6.4 Debugging Commands: How to Use the Debugger

This part of the document will discuss various CDB commands, grouped by function.

When the debugger is invoked, it displays the location of CDB2 (i.e., CDB2ADDR), the amount of space taken up by the local and global symbol tables, and the top of the target stack (i.e., the highest byte not taken up by CDB2 or its tables). The debugger then passes control to the target program, which, after executing the initialization code from C.CCC, invokes the "MAIN" function of the target program. Because a breakpoint is set at the entry to MAIN, control is then passed back to the user, who is prompted for a command.

6.4.1 Breakpoints

CDB normally allows the target program to execute one statement after another without interruption. There are two ways the user can stop target execution; the breakpoint and the keyboard interrupt. By setting breakpoints the user tells CDB to stop immediately before the target executes a given C statement; by generating a keyboard interrupt, the user tells CDB to stop target execution before executing any more C statements. To generate a keyboard interrupt,

the user merely types any character; when CDB sees this character, it will stop execution (note, however, that if the target program is waiting for input the character types by the user will go to the target and NOT cause an interrupt).

To set a breakpoint, the user enters the "break" command:

```
b[reak] [function_name] [statement_number [count] ]
```

(recall that bracketed characters can be omitted; thus, the "break" command can be entered by typing "b", "br", "bre", etc., and both function_name and statement_number can be omitted). If function_name is omitted, the breakpoint is set at the specified statement number of the current function (that is, the function which is currently being debugged; this function name is shown by cdb when target execution is stopped, and can be listed by the "list" command). statement_number tells cdb exactly where in the specified function to set the breakpoint. Statements are numbered by line, with the first line of a function (that is, the function definition definition line on which the open parenthesis is found) being numbered line 1. If multiple statements appear on one line, a decimal notation is used. The first statement in line n is numbered n.0, the next n.1, etc. For example, consider the line

```
a = 5; putchar('x'); while (*s) s++;
```

If this line were the fifth line in a function, then "a = 5;" is numbered 5.0; "putchar('x');" is 5.1; "while (*s)" is 5.2, and "s++;" is 5.3). Whenever no decimal is given, ".0" is assumed. Thus, a statement number can be defined as

```
sn := line_number[.statement_number_within_line]
```

To complicate matters a bit, sometimes CC rearranges the source code or generates its own statements. When this happens, it becomes difficult for the user to set a breakpoint at the desired statement. The most important cases in which CC generates these "hidden statements" are: (i) in the looping constructs ("while", "for", "do"), the compiler generates branch instructions from the bottom of the loop back to the head of the loop; (ii) in the "for" statement, CC moves the "increment" portion of the statement (i.e., the last of the three statements imbedded in the "for" statement) to the end of the loop; thus, this statement is not numbered with the rest of the "for" statement, but with the statement number following the last line of the loop.

Aside from the numbering listed above, there are two special statement numbers, 0 and –1. Statement number 0 is the entrance to a function, and is encountered before any of the code of the function is executed. Statement number –1 is the return from a function, wherever the return happens to be, and is encountered **after** the return is executed (and thus the return value of the function is available for display). Breakpoints can be set at statement numbers 0 and –1 just as any other statements.

The count operand allows the user to defer the recognition of a breakpoint. The breakpoint set by the "break" command does not actually cause cdb to stop executing the target program until the breakpoint has be encountered count times. The default is 1, which causes a stop the first time the statement is encountered. Note that count cannot be entered unless a statement number is given.

Up to forty breakpoints can be set at one time.

The "reset" command is used to remove a breakpoint. The syntax is

```
r[eset] [function_name] [statement_number]
```

and the defaults are the same as for the "break" command. It is, of course, an error to try to "reset" a non-breakpoint. The "clear" command can be used to reset ALL breakpoints; the syntax is

```
clear
```

(no brackets are given; the "clear" command must be typed in full).

The "list breakpoints" command can be used to give a listing of all breakpoints currently set.

6.4.2 Executing code

There are several commands that are used actually to execute the compiled C code. The first of these, the "go" command, simply starts execution (from wherever it was last stopped) and continues until a breakpoint is encountered or the user types a keyboard interrupt. The command has no operands.

To see which statements are executed by the target program, the user can use the "trace" command. The command

```
t[race] [number_of_statements]
```

causes the debugger to execute number_of_statements statements, each time printing the function name and statement number of the statement before execution. Execution ends after number_of_statements have been executed, when a breakpoint is encountered, or at a keyboard interrupt. The default for number_of_statements is 1.

The "untrace" (also know as "walk") is similar to the "trace" command, except that the function names and statement numbers are not displayed as each statement is executed. In other words,

```
u[ntrace] [number_of_statements]
```

causes the debugger to execute number_of_statement statements. As with trace, execution ends after number_of_statements are executed, when a breakpoint is encountered, or at a keyboard interrupt; the default for number_of_statements is 1.

The final command causing target execution is the "run" statement, which cannot be abbreviated. This statement causes cdb to pass control to the target, and deactivates the debugger altogether; once "run" is entered, there is no way to get back to the debugger.

6.4.3 Dumping variables

The "dump" command is used to dump the contents of memory. The syntax of the command is

```
d[ump] [&]expression [multiple] [format]
```

Synonyms for "dump" are "p[rint]" and "," (a comma).

The "dump" command dumps memory starting at the address specified by expression. Although the full definition of an expression is given below, the two most common forms of an expression are a single variable name (such as "i", "foo", or "filename") and an integer in either hexadecimal or decimal notation (such as 0x0100, 43000, or 12). If a variable name or other symbolic expression is given for expression, cdb will dump the variable in the format corresponding to the declaration of that variable; if the variable is a structure, cdb will symbolically dump each element of the structure. However, the user can specify another format to use, and often does so specify when expression is not a symbolic expression but an integer address. The allowable formats are

```
c       character
p       pointer
i or w  integer/word
s       string (null terminated array of char)
```

and "w" is the default if no format is specified for a non-symbolic expression.

The multiple option specifies how much memory is dumped. The "dump" command dumps multiple occurrences of the specified format; thus

```
dump 0x0100 10 c
```

would dump ten characters, from 0x0100 to 0x010A, while

```
dump 0x0100 10
```

would dump ten words (twenty bytes), from 0x0100 to 0x0114, since "w" is the default format.

The syntax for an expression is as follows:

```
expression  :=  *expression
                primary

primary     :=  integer
                identifier
                (expression)
                primary[expression]
                primary.identifier
                primary->identifier
```

This basically means that any C expression that does not contain a logical or arithmetic operator is a cdb expression; the expressions can be fairly complex, as in

```
table[table[1,i],j].name[10]
```

To stop an excessively long "dump" command, type any character.

Normally, C scope rules are used for symbolic references. This means that when the debugger has stopped at a breakpoint in routine "foo", a reference to a variable "bar" refers to the variable local to routine "foo" named "bar" if such a variable exists; if no such local variable exists, the

reference is to the global symbol "bar". This scope rule makes it impossible for a C function with a local variable of the same name as a global variable to access the global variable. cdb allows the user to override the standard scope rule and to specify the global variable by prefixing the variable name with a backslash ("\"). In the example above, to access the global variable "bar" from within the function "foo", the user could type:

```
dump \foo
```

One final use for the "dump" command is finding the address, but not the value, of a symbol. To do this, the expression is prefixed with "&", an ampersand, the C "address of" operator. For example, to determine the address of a variable named table, enter

```
dump &table
```

Complex symbolic expressions can also be used, such as

```
dump &table[i,j]
```

6.4.4 Setting variables

The "set" command is used to store data into memory. The command

```
s[et] expression value [c]
```

will store value into the memory location referred to by expression. Normally, a 16-bit value is stored; however, if (i) expression is a symbolic expression that refers to a char variable, or (ii) value is within single quotes, such as '#', or (iii) the "c" option is given, then only an 8-bit value is stored.

6.4.5 The list command — various items of information

The "list" command is used to access various items of information.

```
l[ist]                  List the current function
                        and statement number
l[ist] a[rguments]      List arguments to current
                        function
l[ist] b[reakpoints]    List all breakpoints
l[ist] g[lobals]        List all global variables
l[ist] l[ocals]         List local variables for
                        current function
l[ist] m[ap]            List linker map of target
                        program
l[ist] t[raceback]      List function trace from
                        MAIN to current function
```

To stop the "list globals" or "list locals" listing of variables, the user can type any character (except carriage return). To stop the listing of a large array and skip forward to the next variable, type carriage return.

The quit command

To end the debug session and return to CP/M, the "quit" command is used. This command cannot be abbreviated.

6.5 Alphabetical Listing of Debugger Commands

A statement number is defined as

```
sn := line_number[.statement_number_within_line]
```

An expression is defined as

```
expression  :=  *expression
                primary

primary     :=  integer
                identifier
                (expression)
                primary[expression]
                primary.identifier
                primary->identifier
```

b[reak] [function_name] [statement_number [count]]

Set a breakpoint. Defaults:

```
function_name:          current function
statement_number:       0
count:                  1
```

clear

Remove all breakpoints.

d[ump] [&]expression [multiple] [format]

Dump "multiple" items in "format" format. Defaults:

```
multiple:       1
format:         i or format associated with symbol
Synonyms:       p[rint] and , (comma).

The allowable formats are:

c       character
p       pointer
i or w  integer/word
s       string
```

g[o]

Begin execution.

l[ist]

List the current function and statement

l[ist] a[rguments] List arguments to current function

l[ist] b[reakpoints] List all breakpoints

l[ist] g[lobals] List all global variables

l[ist] l[ocals] List local variables for current function

l[ist] m[ap] List linker map of target program

l[ist] t[raceback] List function trace

quit Return to CP/M.

r[eset] [function_name] [statement_number]
Remove a breakpoint. Defaults:

```
function_name:          current function
statement_number:       0
```

run Begin execution, disengage debugger.

s[et] expression value [c]
Store data into memory. Normally, a 16-bit value is stored; however, if expression is a symbolic expression that refers to a char variable, value is within single quotes (such as '#') or the "c" option is given, then only an 8-bit value is stored.

t[race] [number_of_statements]
Trace execution, listing statements executed. Default: one statement.

u[ntrace] [number_of_statements]
Execute number_of_statement statements. Default: one statement. Synonym: w[alk]

6.6 An Example — A CDB Debugging Session

This section contains a transcript of a debugging session to demonstrate the use of CDB. The target program, which is contained in the file TARGET.C, is as follows:

```
/*
 * /.C    David Kirkland, 20 October 1982
 *
 * This is a short submit program.  It is designed to be used
 * when the user wants to batch a few commands, but it's too
 * much trouble to edit a SUB file to do the work.  It can be
 * used in two forms:
 *
```

```
 *  B>/ command line 1; command line 2; ... command line n
 *
 *  or
 *
 *          B>/
 *          }command 1
 *          }command 2
 *            .
 *            .
 *          }command n
 *          }
 *
 *  In the first form, the / command is entered with arguments.
 *  group of characters delimited by a semicolon (or the end of
 *  the line) is treated as a separate command.
 *
 *  In the second form, / is entered without arguments.
 *  / then prompts with a "}", and the user enters commands, one
 *  per line.  A null line terminates command entry.
 *  (To enter a null line, enter a singe ^ on the line.)
 *
 *  In either form, control characters can be entered either
 *  directly or via a sequence beginning with a "^" and followed
 *  by a letter or one of the characters: [ \ ] ^ _
 *
 */


#include <stdio.h>

#define OPEN            15          /* BDOS function codes        */
#define CLOSE           16
#define DELETE          19
#define CREATE          22
#define SET_DMA         26
#define RAND_WRITE      34
#define COMPUTE_SIZE    35

struct fcb {                        /* define fcb format          */
        char drivecode;
        char fname[8];
        char ftype[3];
        char extent;
        char pad[2];
        char rc;
        int  blk[8];
        char cr;
        int  rand_rec;
        char overflow;
        };

#define CPMEOF 0x1a
#define MAXBLK 256
#define SUBNAME "A:$$$.SUB"
```

```
struct  fcb ffcb;
                                        /* the way a record from the $$$.SUB  */
struct  subrec {                        /* file looks:                        */
        char reclen;                    /*    number of characters in command */
        char aline[127];                /*    command line                    */
        } ;

struct  subrec out[128];

storeline(block,line) int block; char *line; {

        /* storeline takes the line pointed to by "line" and
          * converts it to $$$.SUB representation and stores
          * it in out[block].
          * This routine handles control characters (the ^
          * escape sequence).
          *
          */

        char *p;
        struct subrec *b;
        int i, len;

        b = out[block];

        /* copy line into out.aline, processing control chars */
        for (p = b->aline; *p = *line; p++, line++)
                if (*line=='^')
                        if ('@' <= toupper(*++line) &&
                                toupper(*line) <= '_')
                                *p = 0x1f&*line;
                        else if (*p = *line)
                                break;

        /* set up length byte */
        b->reclen = len = strlen(b->aline);
        if (len>127) {
                printf("Line %d is too long (%d > %d)\n",block,len,127);
                bdos(DELETE,ffcb);
                exit();
                }

        /* pad block with CPMEOFs (not needed?) */
        for (i=len+2;i<128;i++)
                *++p = CPMEOF;
}

main (argc, argv) int argc;  char *argv[]; {
        char  *p,                       /* points to ; that ended
                                                current command         */
              *b,                       /* current character in
                                                command                 */
              done;                     /* loop control                    */
        char  line[256], *gets();
```

```
        int   block;                    /* index into out array */

        block = 0;

        if (argc<2)                     /* prompt user format          */
                while (1) {
                        putchar('}');
                        if (!*gets(line))
                                break;
                        storeline(block++, line);
                        }
        else {
                /* scan command line in low memory */
                b = p = 0x80;
                for (done=0; !done; p = b) {
                        /* skip leading whitespace */
                        while (isspace(*++b)) p = b;
                        while (*b && *b!=';') b++;
                        done = !*b;
                        *b = 0;
                        storeline(block++, p+1);
                        }
                }

        setfcb(ffcb,SUBNAME);
        if (255==bdos(OPEN,ffcb) && 255==bdos(CREATE,ffcb)) {
                printf("Can't create %s\n",SUBNAME);
                exit();
                }

        /* find end of $$$.SUB so submits can nest */
        bdos(COMPUTE_SIZE,ffcb);

        /* write blocks in REVERSE order for CCP */
        for(--block; block >= 0; block--) {
                bdos(SET_DMA, out[block]);
                bdos(RAND_WRITE, ffcb);
                ffcb.rand_rec++;
                }

        /* all done! */
        if (255==bdos(CLOSE,ffcb))
                printf("Could not close %s\n",SUBNAME);
}
```

The debugging session follows. Text typed by the user is in **boldface**. Note that specific addresses will vary from the numbers appearing here, as this session was recorded using an eariler release of the compiler package.

----- Start of Session -----

B>

```
B>cc target.c -k
BD Software C Compiler v1.xx  (part I)
     35K Unused
BD Software C Compiler v1.xx (part II)
     32K to spare
B>l2 target -d
L2 Linker ver. xxx
Loading  TARGET.CRL
Scanning DEFF.CRL
Scanning DEFF2.CRL

Link statistics:
   Number of functions: 17
   Code ends at: 0x133B
   Externals begin at: 0x133B
   Externals end at: 0x535F
   End of current TPA: 0xE406
   Jump table bytes saved: 0x5D
   Link space remaining: 26K

B>cdb target
c debugger ver 1.21
top of target stack is 8C94, cdb2 is at 9000
globals use 0160 bytes, locals use 00D9 bytes

break at MAIN 0 [0A54]
>list map

STORELIN   08A1          MAIN     0A51     TOUPPER
                                                   0CDA     STRLEN 0D11
PRINTF     0D51          ISSPACE
                                 0D79     ISLOWER
                                                   0DAF     _SPR2    0DDE
PUTS       113E          _USPR    116A     ISDIGIT 120C      _GV2     123B
BDOS       1298          EXIT     12AC     GETS    12B2      PUTCHAR
                                                                      12E6
SETFCB     1318
>list args
argc       [8C90] = 0001 =     1  '..'
argv       [8C92] = 0863
>break storeline
>l breaks
MAIN  -1
STORELIN 0
>go
}dir a:

break at STORELIN 0 [08A4]
>list args
block      [8B81] = 0000 =                 0  '..'
line       [8B83] = 8B8A
>d *line string
8B8A (len 6): "dir a:"
>trace 5
```

```
   trace:          STORELIN 15 [08AF]
   trace:          STORELIN 18 [08CE]
   trace:          STORELIN 18.1 [08DC]
   trace:          STORELIN 19 [08F1]

break at STORELIN 27 [096D]
>break 28
>go

break at STORELIN 28 [09A6]
>dump *b
a struct subrec
reclen          [135F] = 06 = '.'
aline           a 127 element array of char
1360 [  0] 64 69 72 20 61 3a 00 00  00 00 00 00 00 00 00 00 'dir a:.........'
1370 [ 16] 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 '................'
1380 [ 32] 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 '................'
1390 [ 48] 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 '................'
13A0 [ 64] 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 '................'
13B0 [ 80] 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 '................'
13C0 [ 96] 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 '................'
13D0 [112] 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00    '...............'
>d len
[8B7B] = 0006 =       6  '..'
>,b->reclen
[135F] = 06 = '.'
>b setfcb
>go
}

break at SETFCB 0 [131B]
>list args
first argument address is 8B81
[1] = 133B = 4923, [2] = 0C97 = 3223, [3] =01FE = 510
[4] = 22Ca = 8906, [5] = 0030 = 48, [6] = 7269 = 29289
>t

break at MAIN 33 [0BA1]
>dump ffcb
a struct fcb
drivecod   [133B] = 01 = '.'
fname      a 8 element array of char
133C [  0] 24 24 24 20 20 20 20 20 '$$$     '
ftype      a 3 element array of char
1344 [  0] 53 55 42  'SUB'
extent     [1347] = 00 = '.'
pad        a 2 element array  of char
1348 [  0] 00 00  '..'
rc         [134A] = 00 = '.'
blk        a 8 element array of int
134B [  0] 0000 0000 0000 0000 =      0     0     0     0  '........'
1353 [  4] 0000 0000 0000 0000 =      0     0     0     0  '........'
cr         [135B] = 00 = '.'
rand_rec   [135C] = 0000 =       0  '..'
```

```
overflow   [135E] = 00 = '.'
>t 5
   trace:  BDOS 0 [129B]
   trace:  BDOS returning 00FF = 255 = '..'
   trace:  BDOS 0 [129B]
   trace:  BDOS returning 0001 = 1 = '..'

break at MAIN 39 [0BE6]
>t

break at BDOS 0 [129B]
>t

BDOS returning 00FF = 255 = '..'
>t

break at MAIN 42 [0BF6]
>go

MAIN returning FF02 = -254 = '..'
>quit

B>
```

----- End of Session -----

Chapter 7

Tutorials and Tips

This chapter contains tutorial material to introduce the BDS C file I/O library (both buffered and low-level), teach some console I/O interface procedures, and provide some additional operational notes for the compiler.

7.1 BDS C File I/O Tutorial

7.1.1 Introduction

The library functions provided with BDS C for performing file input/output fall into two major categories: the **raw** or **low-level** I/O functions, and the **buffered** I/O functions.

The raw functions, typically coded in assembly language for best performance, are an extended interface to the low-level CP/M BDOS calls that actually perform all file I/O. The quantity of data transferred during raw I/O calls is always a multiple of one full CP/M logical sector (128 bytes).

The buffered functions, written in C, provide a byte-oriented, sequential file I/O system geared especially for filter-type applications. They allow the user to read and write data in whatever sized quantities are most convenient, while invisible mechanisms handle all sector buffering and actual disk transfers. Thus the buffered I/O functions are usually more convenient to deal with than the raw functions, but they generate considerable overhead in terms of speed of execution and consumption of memory space for code and buffer areas.

Since the raw I/O functions form the building blocks from which the buffered functions are constructed, I'll present the raw I/O in detail first and then go on to the buffered functions.

7.1.2 The Raw File I/O Functions

All raw I/O functions are characterized by their use of *file descriptors* to identify the files which are being operated on. A file descriptor, or **fd**, is a small integer value that is assigned to a file when that file is opened or created, and remains associated with the file until it is closed. An fd is obtained by calling either the *open* or the *creat* function. The usage of these functions is:

```
fd = open(filename,mode);
fd = creat(filename);
```

"Filename" is either a literal string or any expression that evaluates to a pointer to characters. *Open* is used to open an already existing file (usually, a file that has some data in it) for reading, writing or both. *creat* is used to create a new file and open it for reading and writing. In both cases, the fd is returned by the call when successful. If some kind of error occurs and the specified file cannot be opened or created, a value of ERROR (-1) is returned instead and the *errno* function may be called to find out exactly why the file could not be opened.

All other raw functions require an fd to specify the file to be operated on (except *unlink* and *rename*, which take filename pointers). Two very important raw I/O functions, *read* and *write*, transfer data to and from disk in multiples of 128-byte logical sectors. Their typical usage is:

```
i = read(fd, buffer, nsects);
j = write(fd2, buffer2, nsects2);
```

The first call tries to read *nsects* sectors of data, from the file whose *fd* is specified, into memory at location *buffer*. The second call tries to write *nsects2* sectors of data, from memory at location *buffer2*, to the disk file whose fd is *fd2*. Unless an error occurs (as when an illegal fd is given or an attempt is made to read past the end of a file), *read* and *write* should cause an immediate disk operation to take place. This is one of the main differences between raw and buffered I/O: raw functions always cause immediate file I/O activity[19], provided the requested operation is possible, while buffered functions only access the disk when a buffer either fills up (during writes) or becomes exhausted (during reads).

There is an invisible "r/w pointer" associated with each file opened for raw I/O. This pointer keeps track of the next sequential sector to be read from or written to the file. Immediately after a file is opened, the r/w pointer is initialized to 0 (the first sector of the file). It is automatically incremented, following *read* and *write* calls, by the number of successfully transferred sectors. So, by default, each data transfer picks up from where the previous one left off. The value of a file's r/w pointer is returned by the *tell* function, and may be modified by using the *seek* function.

To illustrate the use of raw I/O in a program, let's build a simple utility to make a copy of a file. The command format for this utility (which we'll call "copy") shall be:

```
A>copy filename newname <cr>
```

"Copy" will take the file named by "filename" and create a copy of it named "newname". Since this is to be a classy utility, we want full error diagnostics in case something goes wrong (such as running out of disk space, not being able to find the master file, etc.) This includes checking to make sure that the correct number of parameters were typed on the command line. It is sometimes convenient to summarize a program in a half-C/half-English pseudo code form, something like a flowchart but not as boxy. Here is such a summary of the copy program:

19. On most CP/M systems, raw file I/O calls cause the disk drive hardware to go immediately into action. Some systems perform BIOS sector buffering, though, and may not need to go to the physical disk for each and every raw I/O call.

```
copy(file1,file2)
{
    if (exactly 2 args weren't given)
                                                complain and abort
    if (can't open file1)
                                                complain and abort
    if (can't create file2)
                                                complain and abort
    while (not end of file1)
    {
                                                Read a chunk from file1 and write it
                                                if (any error has occurred)
                                                    complain and abort
    }
    close all files;
}
```

And here is the actual C program to perform the copy operation:

```
#include <stdio.h>  /* The standard header file     */
#define BUFSECTS 64 /* Buffer up to 64 sectors in memory    */

int fd1, fd2;       /* File descriptors for the two files   */
char buffer[BUFSECTS * SECSIZ];     /* The transfer buffer  */

main(argc,argv)
int argc;       /* Arg count    */
char **argv;        /* Arg vector   */
{
    int oksects;    /* A temporary variable */

    if (argc != 3)  /* make sure exactly 2 args were given  */
                                                perror("Usage: A>copy file1 file2 <c

                                                    /* try to open 1st file; abort o
    if ((fd1 = open(argv[1],0)) == ERROR)
                                                perror("Can't open: %x\n",argv[1]);

                                                    /* create 2nd file, abort on err
    if ((fd2 = creat(argv[2])) == ERROR)
                                                perror("Can't create: %s\n",argv[2])

                                                    /* Now we're ready to move the d
    while (oksects = read(fd1, buffer, BUFSECTS)) {
                                                if (oksects == ERROR)
                                                    perror("Error reading: %s\n",arg
                                                if (write(fd2, buffer, oksects) != o
                                                    perror("Error; probably out of d
    }
                                                    /* Copy is complete. Now close t
    close(fd1);
    if (close(fd2) == ERROR)
                                                perror("Error closing %s\n",argv[2])
    printf("Copy complete\n");
}

perror(format,arg)  /* print error message and abort    */
{
    printf(format, arg);    /* print message   */
    fabort(fd2);        /* abort file operations */
    exit();         /* return to CP/M  */
}
```

Now let's take a look at the program. First come the declarations: we need a file descriptor for each file involved in the copying process, and a large array to buffer up the data as chunks of disk files are shuffled through memory. The size of the buffer is computed as the sector size (SECSIZ, defined in STDIO.H) multiplied by the number of sectors of buffering desired (BUFSECTS, defined at the top of the program).

In the *main* function, we first make sure that the correct number of parameters were typed on the command line. Since the "argc" parameter is provided free by the run-time package to every main program, and is always equal to the number of parameters given PLUS ONE, we test to make sure it is equal to three (i.e, that two parameters were given). If argc is not equal to three, we call *perror* to lodge a complaint and abort the program. *Perror* interprets its arguments as if they were the first two parameters to a *printf* call, performs the required *printf* call, aborts operations on the output file[20], and exits back to command level.

If we make it past the argc test, it is time to try opening files. The next statement opens the master file for reading, assigns the file descriptor returned by *open* to the variable "fd1", and causes the program to be aborted if *open* returned an error. This can all be done at one time thanks to the power of the C expression evaluator; if you aren't used to seeing this much happen in one statement, take a moment to follow the parenthesization carefully. First the call to *open* is performed, then the return value from *open* is assigned to the variable "fd1", and then a test is done to see if that value was ERROR. If the value was **not** equal to ERROR, then the file had opened correctly and control will pass on to the next **if** statement; otherwise, the appropriate call to *perror* diagnoses the problem and terminates the program. Creation of the output file follows a similar pattern, again with *perror* getting called if the attempted file creation returns an ERROR value.

Having made it through all the preliminaries, it is time to start copying some data (finally!). Each time through the **while** loop, we read as much data as we can get (up to BUFSECTS sectors) into memory from the master file. The *read* function returns the number of sectors successfully read; this may range from 0 (indicating an end-of-file condition) up to the number of sectors requested (in this case, BUFSECTS), with a value of ERROR being returned on disaster (when the disk drive door pops open or something). Whatever this value may be, it is assigned to "oksects" for later examination. In the special case when it is equal to zero, indicating EOF, the **while** loop will be exited. Otherwise, we enter the loop and attempt to write out the data that was just read in. First, though, we want to make sure no gross error has occurred; so, a check is performed to see if ERROR was returned by the *read* call. If so, it's Abortsville. Having safely circumnavigated Abortsville, we call *write* to dump the data into the output file. If we don't succeed in writing exactly the number of sectors we wanted to write, it's back to Abortsville with an appropriate error message (most write errors are caused by running out of disk space.) If the *write* succeeds, we go back to the top of the loop and try to read some more data. This process continues until all of the data has been read and written, at which point the *read* function returns zero and control falls out of the **while** loop.

The last thing to do, once the **while** loop has been left, is to mop up by closing the files; just to be complete, we check to make sure the output file has closed correctly. And that's it.

---

20. This has no effect if called before the file has been opened, as in the case where the wrong number of parameters have been given and the "argc != 3" test succeeds.

7.1.3 The Buffered File I/O Functions

The raw file I/O functions presented in the last section are most useful when large amounts of data, preferably in even sector-sized chunks, need to be manipulated. The preceding file-copy program is a typical application. Raw file I/O requires you to always think in terms of **sectors**—while this poses no particular problem in, say, the file-copy example, it does add quite a bit of complexity to shuffling bits and pieces of randomly-sized data.

Consider, for example, the unit known as the **text line**: a line's worth of ASCII data may vary in size anywhere from 1 byte (in the case of a null string, represented by the terminating null only) up to somewhere around 130 bytes or maybe even more. Some convenient method of reading and writing these text lines to and from disk files would be a very useful thing for text processing applications. Ideally we'd like to call a single function, passing it some kind of file descriptor along with a text line pointer, and have the function write the line of text to the file sequentially following the last line written. Also, to prevent a time-consuming disk access every time a line is written, it would be nice to have our function **buffer up** a number of lines and write them all to disk at once when the **buffer** fills up. Analogously there would have to be a function to read a text-line from a file into memory; here, also, it would greatly improve performance if an invisible buffer were managed by the text-line grabbing function so that disk activity is kept to a minimum. The functions just described are, in fact, *fputs* and *fgets* from the standard library. These are two examples of *buffered* I/O functions.

The spotlight in the world of buffered I/O is a structure named, appropriately, an *I/O buffer*. Within this structure is a large buffer array to store the data being transferred, and several assorted pointers and descriptors to keep track of "what's happening" in the data array portion of the buffer. These include a file descriptor to identify the file for raw I/O operations, a pointer into the data array to tell where the next byte shall be read from or written to, a counter to tell how many bytes of either data or space (depending on whether you're reading or writing) are left before it becomes necessary to reload or dump the buffer, and finally a set of bits that remember things like whether the buffer is being used for input or output (so that the right things happen when the file is closed). Buffered I/O functions use pointers to these I/O buffers as identification for the file being operated on, just as the raw file I/O functions use file descriptors.

There are six functions that perform all actual buffered I/O for single bytes of data (characters). The other buffered I/O functions (such as *fputs* and *fgets*) do their jobs in terms of these six "backbone" functions.

For reading files, there are the functions *fopen*, *fgetc*, and *fclose*. *Fopen* is called to open an existing disk file, identify it by a file pointer variable, and initialize the buffer for receiving data from the file. i(Fgetc) grabs a single byte (character) from the buffer, making sure to refill the data array from the disk file whenever the array is found to be empty, and returns a special EOF value (-1) when the physical end-of-file is reached. *Fclose* closes the file associated with an I/O buffer and frees the buffer for use with another file.

For writing files, the functions *fopen* and *fclose* mentioned above are used, plus the functions *fputc* and *fflush*. *Fopen* creates a new file and prepares an associated I/O buffer structure for receiving output. The data is written to the buffer via calls to *fputc*, one byte at a time; whenever an *fputc* call causes a buffer to fill up, then the buffer is dumped to disk and reset to accept another batch of data. When all the data has been written to a file, *fclose* wraps things up by

closing the associated file. For output files, *fclose* automatically calls *fflush* first to dump out ("flush") the contents of the not-yet-full I/O buffer to the disk file before the file is closed.

The functions that actually read and write data directly to a file are *fgetc* and *fputc*; functions such as *fgets*, *fputs*, *fprintf*, etc. do their reading and writing in terms of *getc* and *putc*.

Careful examination of the STDIO.H header file will reveal that the number of 128-byte sectors used for buffering is 8, by default, and that this value may be changed by the user for optimal performance on different systems. If, for example, you're using BDS C on a CP/M system having a 1024-byte physical sector disk format, then the 1024 bytes of buffering performed by the buffered I/O functions is probably unnecessary, and changing the buffering from 8 sectors to 1 sector would save quite a bit of memory without causing any significant loss in execution speed. On CP/M systems running 8" standard 128-byte physical sectors, though, the default 1K buffering scheme really speeds things up.

Let's look at a simple first example. The following program prints a given text file out on the console, generating line numbers along the left margin:

```
/*
    PNUM.C: Program to print out a text file with
                                                automatic generation of line numbers
*/

#include <stdio.h>

main(argc,argv)
char **argv;
{
    FILE *fp;       /* declare I/O pointer      */
    char linbuf[MAXLINE];   /* temporary line buffer    */
    int lineno;     /* line number variable    */

    if (argc != 2) {    /* make sure file was given */
                                                printf("Usage: A>pnum filename <cr>
                                                exit();
     }

    if ((fp = fopen(argv[1],"r")) == NULL) {
                                                printf("Can't open %s\n",argv[1]);
                                                exit();
     }

    lineno = 1;     /* initialize line number   */

    while (fgets(linbuf, MAXLINE, fp))
                                                printf("%3d: %s",lineno++,linbuf);

    fclose(fp);
}
```

The declaration of *fp* provides the I/O buffer pointer for use with *fopen*, *fgets* and *fclose*.

After checking the argument count and opening the specified file for buffered input (and making sure the file exists), all the real work takes place in one simple **while** statement. First the *fgets* function reads a line of text from the file and places it into the *linbuf* character array. As long as the end of file isn't encountered, *fgets* will return a non-zero (true) value and the body of the **while** statement will be executed. The body consists of a single call to *printf*, in which the current

line number is printed out followed by a colon, space, and the current text line. After the value of *lineno* is used, it is incremented (by the ++ operator) in preparation for the next iteration. The reading and printing cycle continues until *fgets* returns zero; at that point the **while** loop is abandoned and *fclose* wraps things up.

For our final example we have the kind of program known as a **filter**. Generally, a filter reads an input file, performs some kind of transformation on it, and writes the result out into a new output file. The transformation might be quite complex (like a C compilation) or it might be as trivial as the conversion of an input text file to upper case. Since printing costs are pretty high these days, let's skip the C compiler example for the time being and take a look at a To-Upper-Case filter program:

```
/*
    UCASE.C: Program to convert an arbitrary input text
                                                 file to upper-case-only.
*/

#include <stdio.h>

main(argc,argv)
char **argv;
{
    FILE *ifp, *ofp;
    int c;

    if (argc != 3) {
                                                printf("Usage: A>ucase <file> <newfi
                                                exit();
     }
    if ((ifp = fopen(argv[1],"r")) == NULL) {
                                                printf("Can't open %s\n",argv[1]);
                                                exit();
     }
    if ((ofp = fopen(argv[2],"w")) == NULL) {
                                                printf("Can't create %s\n",argv[2]);
                                                exit();
     }

    while ((c = fgetc(ifp)) != EOF)
                                                if (fputc(toupper(c),ofp) == ERROR)
                                                    printf("Write error; disk probab
                                                    exit();
                                                }

    fclose(ifp);
    fclose(ofp);
}
```

This time there are two buffered I/O streams to be dealt with: the input file and the output file. The first task is to check whether the correct number of parameters were given on the command line. In this case, we expect two parameters: the name of an existing input file, and the name of the output file to be created. Then *fopen* is used to open and create the two files for buffered I/O. If that much succeeds, the main loop is entered and the fun begins.

On each iteration of the loop, a single byte is grabbed from the input file and compared with the end-of-file value EOF. Note that the EOF value returned by *fgetc* is –1, which can only be represented as a 16-bit value because **char** variables in BDS C cannot take on negative values.

This is why the variable "c" is declared as an **int** instead of a **char** in the above program; if it were declared as a **char**, then the sub-expression

```
(c = fgetc(ifp))
```

would result in a value having the type **char**, and thus could never possibly equal EOF (-1) as tested for in the program. When *fgetc* returned EOF in such a case, "c" would end up being equal to 255 (the **char** representation of the low order 8 bits of the value EOF). Thus, "c" is declared as an **int** so the EOF comparison can make sense.

This is awkward because "c" is used here for holding characters, and it would be nice to have it declared as a character variable. There's actually a way to do it, at the price of complete generality: if the EOF in the comparison were changed to 255, then "c" would have to be declared as a **char** and the program would work…**except** when an actual hex FF (decimal 255) byte is encountered in the input file! Now, while it is a pretty safe bet to assume there aren't any hex FF bytes in your average text file, there may be exceptions. Also, there's no law saying filters can only be written for text files. Consider a program to take a binary file and "unload" it, creating an Intel-format HEX file. Would we want it to halt when the first hex FF is encountered? No, the original method is clearly the most general.

After determining that the end-of-file has not been encountered, the body of the **while** statement is executed. Here we use *toupper* to convert the character obtained from *fgetc* to upper case, and then we use *fputc* to write the resulting byte out to the output file. To be neat, errors are checked for: the program terminates if *fputc* returns ERROR.

As soon as an end-of-file condition is detected, we use *fclose* to close the input and output files. Note that *fclose* automatically calls *fflush* for the output file, and *fflush* makes sure that the output file is terminated by a CP/M end-of-text-file (0x1A, or Control-Z) character.

For a large-scale example of buffered I/O usage, see CASM.C. Also, take some time to inspect the files STDIO.H, STDLIB1.C and STDLIB2.C, which contain the sources of all the buffered I/O functions. STDLIB1.C contains the general byte-oriented portion of the buffered I/O library, and STDLIB2.C contains the line-oriented and format-conversion functions.

7.2 BDS C Console I/O: Some Tricks, Clarifications and Examples

7.2.1 Introduction

In this document I will attempt to remove some of the mystery behind the CP/M console input/output mechanism, and show how to take best advantage of that mechanism from BDS C programs.

The accent here will be on how to use the *bios* and *bdos* library functions for performing console input and output directly via CP/M's BIOS and BDOS, respectively. One reason for going directly to CP/M's BIOS for console I/O, instead of using the *getchar*/*putchar* functions supplied in the standard library, is to avoid the frustrating unsolicited interception of certain ASCII

characters by both the CP/M BDOS **and** the *getchar*/*putchar* functions (which use BDOS calls to perform their tasks). Some suitable applications are telecommunication programs, games, or any programs requiring more direct control over the console than the standard *getchar* and *putchar* functions provide.

7.2.2 Elementary Console Interfacing

Let's take a look at what really happens during console I/O, and how to control it…

The lowest (simplest) level of console-controlling software is in the BIOS (Basic Input/Output System) section of CP/M. There are three subroutines in the BIOS that deal with reading and writing raw characters to the console: **CONST** (check CONsole STatus), **CONIN** (wait for a character to be typed on the CONsole, then read it IN), and **CONOUT** (send the CONsole an OUTput character to be typed). The way to locate these subroutines from the assembly language level is rather complicated, so the BDS C library contains the *bios* function to make it easy to access the BIOS subroutines from C programs.

BIOS vectors 2, 3 and 4 are used to communicate directly with the console device. The expression **bios(2)** specifies a call to the CONST subroutine in the bios, which returns a non-zero ("true") value when a character is available at the console, or zero otherwise. To actually read the character after **bios(2)** indicates one is ready, or to wait until a character is ready and then read it, use **bios(3)** to call the CONIN subroutine and return a character from the console. To directly write a character **c** to the console, say **bios(4,c)** to call CONOUT. Note, though, that the BIOS subroutines are not aware that C programs represent a carriage-return/linefeed combination by a single "newline" character ('\n')…the call **bios(4,'\n')** will cause only a single linefeed character (ASCII decimal value 10) to be printed on the console **without a leading carriage-return**. When using direct console I/O you must send both a carriage-return ('\r') **and** a newline ('\n') to the CONOUT subroutine in order to go to the beginning of a new line on the console output.

Such a sequence would appear as follows:

```
bios(4,'\r');   /* send carriage-return to CONOUT */
bios(4,'\n');   /* send linefeed to CONOUT */
```

Making sure that all console I/O is eventually performed by way of the three BIOS subroutines is the **only** way to approach portability of programs between different CP/M systems when total control is required over the console device[21].

7.2.3 The BDOS and How It Complicates Things

The next higher interface level (above the BIOS) on which console I/O may be performed is the BDOS (Basic Disk Operating System). Just as there are the three basic BIOS subroutines for interfacing with the console, there are three similar but "higher level" BDOS operations for performing similar tasks. These BDOS functions, each of which has its own code number distinct from its BIOS counterpart, are: **Console Input** to get a single character from the console

21. Even so there's no way to know what kind of terminal is being used by another system— so "truly portable" software either makes some assumptions about the kind of display terminal being used (whether or not it is cursor addressable, **how** to address the cursor, etc.) or includes provisions for self-modification to fit whatever type of terminal the end-user happens to have connected to the system.

(BDOS function 1), **Console Output** to write a single character to the console (BDOS function 2), and **Get Console Status** to determine if there is a character available from the console input (BDOS function 11). There is also BDOS function 6, named "direct console I/O", provided as a direct link to the BIOS console I/O functions. This functions is "yet another way to get not quite complete control over console I/O", and has only one slight advantage over using BIOS calls (which I'll describe later).

Whenever the standard C library functions *getchar* and *putchar* are called, they perform their tasks in terms of BDOS calls…which in turn perform **their** operations through BIOS calls, leading to some nasty confusion. The BDOS operations do all kinds of things for you that you may not even be fully aware of. For instance, if the BDOS detects that a control-S character is present on the console input during a console **output** call, then everything will stop dead until another character is typed on the console input, before control is returned from the original output call. This may be fine if you want the ability to stop and start a long printout without having to code that feature into your C programs, but it causes big trouble if you need to see **every** character typed on the console, including control-S. A little bit of thought as to how the BDOS does its stuff reveals some interesting facts: since the BDOS must be able to detect control-S on the console input, it must read the console whenever it sees that a character has been typed. If the character is not among those requiring special processing, such as control-S, then it must be saved somewhere internal to the BDOS so that the next "Console Input" call returns the character as if nothing happened. Also, the BDOS must make sure that any subsequent calls made by the user to "Get Console Status" (before any are made to "Console Input") indicate that a character is available. This leads to a condition in which a BDOS call might say that a character is available, but the corresponding BIOS call would NOT, since, physically, the character has already been gobbled up by the BDOS during a prior interaction with the BIOS.

If this all sounds confusing, bear in mind that it took me several long months of playing with CP/M and early versions of the compiler before I was able to comprehend what goes on in there. The library versions of *getchar* and *putchar* were designed for an environment where the user does **not** need absolute direct control over the console. Since the BDOS already does some nice things (like control-S processing), I threw in some additional features: automatic conversion of the '\n' character to a CR-LF combination on output, automatic program termination when control-C is detected on input or output (so that programs having long or infinite unwanted printouts may be stopped without resetting the machine, even when no console input operations are performed), automatic conversion of the carriage-return character to a '\n' on input, etc.

For BDS C v1.6, the new functions *iobreak* and *cmode* were added to provide some flexibility to the *getchar* and *putchar* functions. Specifically, the *iobreak* function allows the selection (under program control) of whether or not Control-C is detected during *getchar* and *putchar* calls. Calling *iobreak(0)* causes Control-C's to be ignored (except when typed as the first character of a line under line-buffered input mode, due to the way the BDOS operates). Calling *iobreak(1)* (or not calling *iobreak* at all, as *iobreak(1)* is the default on start-up) causes any Control-C's typed on the console during either character input or character output to terminate program execution and return to CP/M command level.

I promised some examples earlier, so let's get to them. First of all, here is a very rudimentary set of functions to perform the three basic console operations in terms of the *bios* function, with no special conversions or interceptions **at all**…i.e., nothing like the '\n' —> CR-LF translations:

```
/*
    Ultra-raw console I/O functions:
*/

getchar()     /* get a char from the console */
{
    return bios(3);
}

kbhit()       /* return true (non-zero) if a char is ready */
{
    return bios(2);
}

putchar(c)      /* write the char c to the console */
char c;
{
    bios(4,c);
}
```

These ultra-raw functions do nothing more than provide direct access to the BIOS console subroutines. To use them instead of the standard versions provided in DEFF2.CRL (which, incidentally, are written in assembly language and available in source form within DEFF2A.CSM), simply create a C source file containing them (or any variation you please), compile the file, and link your programs with the resulting CRL file.

Now let's consider some more sophisticated games that can be played with customized versions of the console I/O functions. For starters, let's design a set of direct console I/O functions that perform newline conversions just like the library versions described earlier, abort execution on control-C, but **ignore** control-S/control-Q protocol and throw away any characters typed during output **except** control-C, which should cause a return to command level. What we need here are the skeletal functions given above, plus some extra code to handle the following conditions: a) conversion of single '\n' characters into two characters, CR and LF, on output; b) conversion of CR to newline ('\n') and control-Z to –1 on input; c) automatic echoing of input to the console output; and d) re-booting on control-C during both input **and output**. Here are the beasts:

```
/*
    Vanilla console I/O functions without going through BDOS:
    Note that 'kbhit' would be the same as the preceding
    raw version)
*/

#define CTRL_C 0x03     /* control-C */
#define CPMEOF 0x1a     /* End of File signal (control-Z) */

getchar()       /* get a character, hairy version  */
{
    char c;
    if ((c = bios(3)) == CTRL_C) exit();  /* abort on ^C */
    if (c == CPMEOF) return -1;      /* turn Ctl-Z to -1 */
    if (c == '\r') {            /* if CR typed, then  */
        putchar('\r');      /* echo a CR, and set */
        c = '\n';           /* up to echo a LF also */
    }               /* and return a '\n'    */
    putchar(c);         /* echo the char  */
    return c;           /* and return it  */
}
```

```
putchar(c)      /* output a character, hairy version  */
char c;
{
    bios(4,c);          /* first output the given char */
    if (c == '\n')      /* if it is a newline, */
    bios(4,'\r');       /*      then output a CR also */
    if (kbhit() && bios(3) == CTRL_C) /* if Ctl-C typed, */
        exit();                /* then reboot  */
    }           /* else ignore the input */
```

Now, if you want to add control-S processing and a push-back feature (the two are actually quite related, since you must be able to push back anything except control-S that might be detected during output), you could add some external "state" to the latest set of functions and keep track of what you see at the console input. Once this is done, though, what you'd have is much the same functionality as the original standard library versions of *getchar* and *putchar* (which use the BDOS), and you might as well just use those.

So far, everything I've talked about has been in terms of the BIOS, and applies equally to all CP/M systems. Unfortunately, there is one console operation often needed when writing real-time interactive operations that is not supported by the BIOS, and thus there is no portable way to implement it under CP/M. What's missing is a way to ask the BIOS if the console terminal is **ready to accept** a character for output. An example of the trouble this omission causes is visible in the sample utility CMODEM.C. There, the program must be able to read input from the keyboard at any instant, and cannot afford to become tied up waiting for the terminal when the amount of data being sent to it has caused it to refuse more characters and thereby to lock up the program until a character can be sent. Given that the only "kosher" way to send a character to the console is through the CONOUT BIOS call, and that such a call might at any time tie up the program for longer than is tolerable, the only recourse is to bypass CONOUT completely and construct a customized output routine in C that can be more sophisticated. This is done in CMODEM.C, at the expense of non-portability for the object code; each user must individually configure his HARDWARE.H header file to define the unique port numbers, bit positions and polarities of the I/O hardware controlling his console and modem devices. It would have been much easier if the BIOS contained just one more itty bitty subroutine to test console output status and modem output status, but life is rough sometimes.

The last several examples have all used the *bios* function for direct interface to the BIOS console subroutines. Note that BDOS function number 6 was provided in CP/M 2.x as an alternative to direct BIOS access, and it can indeed be used in most cases instead of *bios* calls. I know of both one advantage and one disadvantage to using BDOS function 6, though. The disadvantage is that you cannot send a hexadecimal FF byte to the console output using BDOS function 6. The advantage, on the other hand, involves an incompatibility problem between different implementations of CP/M. On some systems, the *bios* function provided with BDS C will not work correctly because the **jmp** instruction at the start of the CP/M base page does not point directly to the warm-boot entry in the BIOS jump-vector table. The *bios* function assumes that this is the case, and computes the address of the base of the BIOS vector table on this assumption. The significance of all this is that C programs written using the *bios* function, and distributed in binary form to other systems which do not conform to the *bios* function's assumptions about **jmp** instruction targets, will not work correctly on those systems.

Oh well…I hope this has helped to demystify some of the obscure behavior of the CP/M console I/O interface. For the low-down on how the library versions of *getchar*, *putchar*, etc. really work, see their source listings in DEFF2A.CSM.

7.3 Some Mistakes Commonly Made By Beginning C Programmers and Other Things Deserving Clarification

There are several aspects of the C language that tend to cause a great deal of brow-beating when encountered for the first time. In this section I will try to summarize those sensitive "features" of C that are constantly being brought to my attention by confused users in their phone calls and letters.

7.3.1 '=' versus "=="

The = operator is used for *assignment* only, while the == operator is used for testing a relational condition of equality. The two operators have nothing in common except the character used to represent them, and can cause very frustrating debugging sessions when confused.

A common construct in C is to have an assignment operation imbedded within a larger expression, perhaps involving conditionals. This can lead to statements such as:

```
if ((c = getchar()) == '\n')
        printf("You typed a newline!\n");
```

Here, the beginning C user might interpret the = operation as a conditional test instead of the assignment expression it actually is. Note also that the precedence of the == operator is **higher** than that of the = operator. This fact makes it essential that the assignment operation be explicitly parenthesized in an expression such as the one above. If the statement were mistakenly written as:

```
if (c = getchar() == '\n')
        ...
```

then the compiler would treat that expression exactly the same as if it were written:

```
if (c = (getchar() == '\n'))
        ...
```

Now consider the following code fragment:

```
if (!(c = getnext())) {
        printf("All done\n");
        break;
}
```

The **if** expression in this statement assigns the return value from the *getnext* function to the variable *c*, then asks whether or not that return value is zero…if it is zero, it prints "All done!" and breaks out of whatever control structure encloses the fragment. Of course, if a tired programmer looks at this very quickly, it might seem as if *c* were being compared to the return value of *getnext*…you get the idea.

7.3.2 Character Constants within Literal Strings

Often it is necessary to imbed non-standard characters inside literal strings. All ASCII characters and most useful control characters (e.g. newline, carriage-return, formfeed, etc.) are easy enough to represent in a string, but the more obscure control characters (and **all** 8-bit characters having the high-order bit set) must be represented in the following form: a backslash (the '\' character) followed by the **octal** form of the value of the character. While C allows the representation of hexadecimal values by a special prefix notation (e.g., 0x1f) in general expressions, note that this notation is **not** allowed for single-quoted character constants or within double-quoted literal strings. **Anytime** the backslash-prefix notation is used the digits are presumed to be octal, and therefore the first non-octal digit encountered will not be considered part of the value. As an example of the confusion this can cause, consider the following statement:

```
printf("This is a test. Here is a bell: \08\n");
```

What actually is printed for output? If you think a bell (or beep) will sound, look again…the digit '8' is *not legal in octal*, so the compiler considers the sequence '\0' a complete octal constant (having the value zero), and leaves the '8' alone to print out on the console. The result of the above statement would be:

```
This is a test. Here is a bell: 8
```

and an invisible null would be "printed" immediately before the '8'. The **correct** way to get the desired effect is:

```
printf("This is a test. Here is a bell: \10\n");
```

7.3.3 The Precedence of Assignment Operators

Because there are so many binary operators in C, it is easy to confuse the relative precedence of the different operators and get very incorrect results when explicit parenthesization is lacking. By far the most common example involves assignment operators used in conjunction with other binary expression operators. For example, the correct way to assign the return value of function *getc* to the variable *c*, and then compare that value to the symbol CPMEOF, is as follows:

```
if ((c = getc(fp)) == EOF)
        puts("Found EOF\n");
else
        puts("No EOF yet...\n");
```

When the first line is mistakenly written as follows:

```
if (c = getc(fp) == EOF)
```

the effect is **entirely** different; because the precedence of the == operator is **higher** than that of the = operator, the comparison for equality between the return value of *getc* and the symbol EOF will be performed **before** the assignment to *c*, and thus *c* will end up with a logical value of either 0 or 1 depending on the result of the comparison. This, obviously, is not the desired effect. A rule of thumb in these kinds of cases is: if an assignment expression is placed within a larger

expression involving other binary operators, **isolate the assignment expression in parentheses** or it will probably not do what you want it to.

7.3.4 Array Subscripting

Arrays of length *n* in C have elements numbered from 0 to *n-1*. If you declare an array of length *n* and attempt to reference an element with a subscript of value *n*, you'll actually be referencing data past the end of that array. This happens most often when a user is thinking in terms of the BASIC language, where arrays of length *x* may have both an element number 0 **and** and element number *x*. Note that in C, the most common **for**-loop construct neatly iterates through *n* items numbered 0 through *n-1* as follows:

```
for (i = 0; i < n; i++)
        ...
```

and such loops are ideal for iterating through an array. If you *really* need to have an array numbered 1 through *n* for *n* items, then you must declare the array to have one more item than required, leaving the 0-th element unused.

7.3.5 How NOT To Use a Pointer

When a pointer variable is declared in a program, either externally or within a function, it is **not** given a value automatically. A pointer is simply a 16-bit variable that is typically used to hold the address of some other piece of data (to *point* to it), and must be initialized before being used, just like any variable. The particular mistake I see most often involves assigning a value indirectly through an uninitialized pointer; e.g, the declaration

```
char *foo;
```

would be later followed by a statement such as

```
*foo = 'a';
```

before *foo* is ever assigned any specific value, and unpredictable things would begin to happen. What the assignment statement above says is "place the character 'a' into memory at the location whose address is specified by the value of variable *foo*." If *foo* has never been initialized to anything, then the 'a' character gets stored in some totally random location in memory. The correct procedure here would have been to declare a buffer area, assign *foo* the address of that area, and **then** begin assigning data indirectly through *foo*. For example, the following sequence places the character 'a' at location buffer[0]:

```
char buffer[50], *foo;
foo = &buffer[0];
...
*foo = 'a';
```

7.3.6 Functions Shouldn't Return Pointers to Their Automatic Data

As soon as a function returns to its caller, storage that was local to that function (i.e., where all declared local variables were stored) is **de-allocated** and made ready for use by the next called function. A common mistake is to have some function (call it *foo*) create a piece of text in a local

buffer and return a pointer to that text… Immediately upon return from *foo* the text appears intact, but later on in the course of the program (as the space in which the string resides is allocated for other functions' local data frames), the string turns into garbage. There are two viable solutions to this kind of problem: a) Have *foo* take a parameter telling it where to put the string result (in which case the caller must provide a working buffer for *foo*), or b) Make the destination string area external. Each method has its own advantages; passing a destination area on each call allows many such returned strings to be saved separately in different areas of memory, while an external destination area shortens the calling sequence by requiring one less parameter to be passed. But whatever you do, do **not** expect any data that was locally allocated by a called function to remain valid after that function has returned!!

7.3.7 Understanding Formal Parameters

What is a "formal parameter", anyway? A formal parameter is one of the arguments (if any) that a function expects to have passed to it whenever called. All formal parameters are specified at the beginning of a function's definition in a parenthesized list immediately following the function name. The *declarations* of a function's formal parameters must be made immediately after the parenthesized list, before the first open-curly brace that marks the beginning of the function body. Any formal parameters not explicitly declared are assumed to be simple **int** values. If a formal parameter is accidentally declared within the actual function body (inside the curly-braces), the compiler will correctly diagnose a "redeclaration" error… since after the formal declarations are passed and the compiler begins processing the function body without having seen a declaration for a formal parameter, that formal parameter will have been automatically declared as an **int**.

Whenever a function call takes place, *copies* of the values of any formal parameters are passed to the function. All such values are 16 bits in length with BDS C version 1. This means that structures, arrays, or any data type not inherently 16 bits in size cannot be directly passed to a function; *pointers* to such data types, though, can. Now…what happens when an array name is passed to a function? There is a special magic mechanism for passing pointers to arrays that can be confusing, because it is not intuitively obvious from the declaration syntax that a pointer is actually being passed. For example, consider the following function:

```
int arraysum(array)
int array[3];
{
        return array[0] + array[1] + array[2];
}
```

While *arraysum* may appear to take an array of 3 elements as a formal parameter, in reality only a *pointer* to that array is passed. The declaration looks as if an entire array were being passed, but if you change any element in the array here you'll be changing that element for the calling program also. There is only one copy of the array in existence.

Another tricky point about formal array parameters is that you can actually treat the array name as a simple pointer variable within the called function (i.e., assign to it the address of another array and voila! it then becomes the base of that other array…) while such things would not work (and indeed, cause unpredictable results) when the array is an *actual* (non-formal-parameter) array. The Kernighan & Ritchie book contains an entire chapter on the "duality" of

pointers and arrays; in this mechanism are the most powerful **and** the most confusing aspects of C.

7.3.8 Dependence on Parameter Evaluation Order

Function calls should never be written such that varying the order of evaluation of the parameters in a single call could have an effect on the values of the parameters. An example of such a badly written call is as follows:

```
x = 1;
foobar(x++, x++, x);
```

The three values passed to the function *foobar* in this example would end up being 2, 1, 1, **not** 1, 2, 3 as might be expected. Most C compilers evaluate function parameters in **reverse** order, including BDS C, so that they will end up on the stack in "forward" order and allow functions like *printf* to process a variable number of parameters in an efficient manner. Thus, function parameters should never have side effects which change the values of other parameters in the same list, or in fact even in the same expression.

The lesson here is to be careful not to rely on the *order of evaluation* when dealing with several parameters in a function call. If the order is critical and side effects cannot be avoided, then each parameter should be made into a separate statement with values assigned to temporary variables, so that the values can be placed in a function call later when all ordered computation is complete.

7.3.9 Function Calls MUST Have Parentheses

If the name of a function is used without an argument list, then the resulting expression evaluates to the **address** of the named function…no call is ever made to the function unless the name is followed by a parenthesized list of parameters, even if the list is null. For example, the following expression assigns the address of the end of the external data area to the variable *i*:

```
i = endext();
```

while the following expression assigns the address of the function *endext* to variable *i*, but only if *endext* has been previously declared:

```
i = endext;
```

Note that if *endext* has not been previously declared when the latter expression is encountered, then the compiler will correctly diagnose the "undeclared variable" *endext*. In the first example, though, *endext* is implicitly declared (in context) as a function returning an **int**.

7.4 Miscellaneous Notes

This section contains a collection of tips and clarifications about both the C language in general and some of the BDS C Compiler's quirks.

- The "Constant expression" evaluation mechanism, as described in section 4.15, indicates how BDS C simplifies certain expressions involving constant values at compilation time. Because constant expressions are often used for calculating the dimensions of arrays and structures, it was decided to have BDS C perform all constant expression simplification in **unsigned** arithmetic mode. Because of this, certain innocent-looking arithmetic expressions written in terms of constant values may yield unexpected results when the unary minus (negation) operator is used. For example, the statement

```
printf("%d\n",-12/5);
```

- causes the value "13104" to be printed as the value of "-12/5". This is because the division is performed in unsigned arithmetic mode; the "-12" is actually treated as a value of 65524, which when divided by 5 yields 13104.

- The keywords **begin** and **end** may be substituted for left and right curly-braces (**{** and **}**). This feature is provided so that users not having the curly-brace characters on their terminals can still use the compiler. Aesthetically, at least in this hacker's opinion, the curly-braces produce listings far more readable than **begin** and **end**, and should be used whenever possible.

- Error recovery during compiler operation may not appear especially intelligent in certain cases. If either CC or CC2 spews out a set of error messages clustered around the same line or set of lines, then only the *first* error message in the cluster should be believed. Chances are that after that error is fixed, the rest will go away.

- The line number given by CC2 in error reports is not always guaranteed to be accurate. CC does some rearranging of code once in a while; for instance, the increment portion of a **for** statement is physically moved down past the statement portion. Thus, if there is an error in the increment portion that CC is not equipped to detect, then CC2 *will* detect it…and report the line number erroneously. Try not to mess up the increment portion of **for** statements.

- Certain types of errors will cause the compiler to cease execution and immediately return to the operating system without scanning the rest of the source. This occurs when, for example, mismatched parentheses or a missing semicolon manage to confuse the compiler to the point where it cannot recover. Instead of guessing about where the proper punctuation *should* be, it aborts to let you fix the error quickly and try again.

- Note that the *argc* value passed to a C *main* function is, by convention, always positive, and equal to the number of arguments specified **plus one**.

- The first string in argv, argv[0], is **undefined** due to CP/M's not providing the name of the executing program to transient programs.

- Arguments on the command line are **character strings** in all cases, not values. To convert a numeric command line parameter into a value appropriate for assigning to a variable, something like the *atoi* function must be used.

- A problem with the "bdos" library function has come up that is rather tricky, since it is system-dependent: A program that runs correctly under a normal Digital Research CP/M system might **not** run under MP/M or SDOS (or who knows how many other systems) if the *bdos* function is used. A typical symptom of this problem is that upon character output, a character on the keyboard needs to be hit once in order to make each character of output appear.

  To understand the problem, we must first understand exactly how the CPU registers are supposed to be set after an operating system BDOS call. Normal CP/M behavior (which the library function *bdos* had always assumed) is for registers A and L to contain the low-order byte of the return value, and for registers B and H to contain the high order byte of a return value (which is zero if the return value is only one byte). The CP/M interface guide explicitly states that "A == L and B == H upon return in all cases", and I figured that just in case CP/M 1.4 or some other system didn't put the values in H and L from B and A, I'd have the *bdos* function copy register A into register L and copy register B into register H, to make **sure** the value is in HL (where the return value must always be placed by a C library function.)

  Not all systems actually follow this convention, though. Under MP/M, H and L always contain the correct value but B does not! So when B is copied into H, the wrong value results. Therefore, the way to make *bdos* work under both CP/M 2.2 and MP/M was to discontinue copying B and A into H and L, and just assume the value will always be correctly left in HL by the system. This was done for v1.45, so at least CP/M and MP/M are taken care of, but…

  Under SDOS (and perhaps other systems), register A is sometimes the **only** register to contain a meaningful return value. For example, upon return from a function 11 call (interrogate console status), the B, H and L registers were all found to contain garbage. So if no copying is done in this case, the return value never gets from A to L and the result is wrong; but if B is copied into H along with A getting copied into L, the result is still wrong because B contains garbage. Evidently the only way to get function 11 to work right under SDOS is to have the *bdos* function copy register A into L and **zero out** the H register before returning…but then many other system calls which return values in H wouldn't work anymore. And that is the problem: You can please **some** systems all the time, but not **all** systems all the time with only one standard *bdos* function.

  The way I left *bdos* for v1.5 is so that it works with CP/M and MP/M (i.e., no register copying is done at all…HL is assumed to contain the correct value). This, of course, won't work in all cases under SDOS and perhaps other systems…in those cases, you need to either use the *call* and *calla* functions to perform the BDOS call, or create your own assembly-coded version(s) of the *bdos* function (using CASM) to perform the correct register manipulation sequences for your system. Note that it may take more than one such function to cover all possible return value register configurations.

- A well-designed C program should always diagnose a command line error by displaying the command line syntax to the user and aborting. This is generally known as a "Usage" message; it reminds the user of what is expected on the command line and often saves everyone who uses the program a lot of time. If there are command line options, they

should be shown in square brackets. A good practice is to include detailed explanations of all the options along with the sample command line.

- Although external initializations are not supported by the compiler, some convenience functions have been provided to allow initialization of simple integer and character arrays. To set any contiguous set of words to integer values, use the function *initw*. For characters (single-byte integers in the range 0-255), **but not strings**, use *initb*. For example, to simulate the UNIX C construct of

```
int foobar[10] = { 3,0,-2,-5,3,6,9,-23,-14,0 };
```

- you can first declare foobar normally by saying

```
int foobar[10];
```

- and then, in the main function, insert the statement

```
initw(foobar,"3,0,-2,-5,3,6,9,-23,-14,0");
```

- The following tidbits should be kept in mind when striving for optimum efficiency:

  1. Comments are stripped off a source file dynamically as the file is being read in from disk; thus, there is no excuse (except maybe laziness) for not documenting a program adequately.

  2. The **switch** statement is most efficient when the switch variable (e.g. *xx* in "**switch** (*xx*)…") is declared as a **char**. Integer variables are often used to hold character values in text processing applications involving file I/O; assigning such a value to a character variable before large **switch** constructs could save memory and speed up execution.

  3. The **case**s in a **switch** statement are tested in the order of their appearance; thus, the most common cases (or the ones requiring fastest response time) should appear first.

  4. For the fastest execution speed possible, CC should be given the **–o** and **–e** *xxxx* options for compilation. For the shortest possible code length, only the **–e** *xxxx* option should be used with CC.

  5. Logical expressions in C evaluate to a numerical value of 0 (if false) or 1 (if true) whenever their value is actually needed, but may not evaluate to any value at all when used in flow-of-control tests. This means that you can take advantage of the numerical results of logical expressions in many situations. Consider the following code fragment, whose purpose is to set the variable *x* to 1 if a<b, or to 0 if a >= b:

```
if (a < b) x = 1;
        else  x = 0;
```

  6. The same operation can be written as

```
x = (a < b);
```

7. This takes advantage of how the subexpression "(a < b)" evaluates to the desired value automatically, and thus avoids the use of two separate assignment expressions, their associated control structure, and the considerable overhead that all entails.

8. A related opportunity for brevity comes up whenever any variable needs to be tested for equality or inequality with zero; since any expression may be considered logically "true" if it evaluates to a non-zero value, the "!= 0" portion of an expression such as "a != 0" is practically redundant. Statements such as

```
        if (a != 0) printf ("A is non-zero");
or      if (a == 0) printf ("A is zero");
```

9. may just as well be written as

```
        if (a)  printf ("A is non-zero");
 and    if (!a) printf ("A is zero");
```

10. Of course, such an abbreviation may not always be appropriate to a given situation. If the variable in question is used as a counter of some sort, and is expected to take on many different values, then saying "a != 0" might be clearer to the logic of the program. But in cases where the variable is used as a Boolean flag, or where a value of zero is considered special in some sense, then the shorter forms are clearer and may in fact lead to shorter object code in some cases.

Chapter 8

Auxiliary BDS C Package Programs

This chapter describes several of the larger utility programs included with the BDS C package: the CASM assembly language preprocessor, the L2 linker, and the CMODEM telecommunications package.

8.1 The CASM Assembly-language-to-CRL-Format Preprocessor For BDS C

This section describes the "CASM" assembly language preprocessor system, supplied to allow the combination of C functions with assembly language functions within final object (.COM) files.[22]

CASM is a preprocessor that takes, as input, an assembly language source file of type ".CSM" (mnemonic for C aSseMbly language) in a format very close to the standard assembly language accepted by the standard CP/M ASM.COM assembler, and writes out an ".ASM" file as output. This .ASM file must be assembled by ASM.COM, yielding a .HEX file, and this .HEX file is finally converted into a .CRL file by using the CLOAD.COM utility.[23]

CASM operates by automatically recognizing which assembly language instructions require relocation parameters, and inserting the appropriate pseudo-operations and extra opcodes into the ".ASM" output file so it properly assembles directly into CRL format. In addition, some rudimentary logic checks are performed: doubly-defined and/or undefined labels are detected and reported, and similarly-named labels in different functions are ALLOWED and converted into unique names so ASM won't complain.

The .CSM files prepared as input to CASM.COM must consist of individual assembly language functions delimited by the FUNCTION and ENFUNC pseudo-ops (described below). Each functions must conform to the calling convention and register allocation rules detailed in Chapter 2.

---

22. The means provided with pre-1.46 versions of BDS C for creating relocatable object modules (CRL files) from assembly language programs was the macro package CMAC.LIB that operated in conjunction with Digital Research's macro assembler (MAC.COM). That system was inadequate for two reasons: a) MAC.COM, if not already owned, cost about as much as the entire BDS C package to purchase, and b) the macros in CMAC.LIB were difficult to use. This CASM procedure replaces the CMAC.LIB method.

23. CASM restricts mnemonics in the source file to only 8080 operations recognized by ASM.COM. An alternative package named ZCASM.COM, contributed to the C User's Group and available to all members for a nominal media cost, accepts Z80 mnemonics and works in conjunction with the Microsoft M80 assembler.

8.1.1 Creating CASM.COM

CASM is supplied in source form only (as CASM.C) on the BDS C distribution disk. Before compiling CASM.C to make an executable version, customize the beginning of the file by setting the default library drive and/or user area definitions to conform to your system configuration. Instructions for compilation and linkage of CASM are given in the comments at the head of the file.

8.1.2 Command Line Options

–c Enables comment retention on both input and output. By default, CASM strips off all comments from the input file when reading it in, and does not put any comments into the assembly code added to form the final ASM file. If **–c** is specified, the original comments are preserved and CASM adds its own comments to new sections of code.

–f Flags old CMAC.LIB macro library operators, to help users convert old assembly language source files to the CSM format.

–o *name* Calls the output file *name*.ASM. Normally, the output file is named by tacking an .ASM extension onto the filename of the CSM input file.

The files making up the CASM package are as follows:

CASM.C Source file for CASM program.

CLOAD.C Replaces CP/M's LOAD.COM, to be used for converting the .HEX output of ASM.COM directly into .CRL format. This program properly handles out-of-sequence .HEX data that would draw an "INVERTED LOAD ADDRESS" error from the LOAD.COM utility provided with CP/M. Note that with the addition of CLOAD.C, it is no longer necessary to enter an explicit SAVE comment to CP/M at the conclusion of the CASM sequence, as was the case with most pre-v1.51 releases.

CASM.SUB Submit file for performing the entire conversion of a CSM file into CRL format. For a file named FOO.CSM, you would type:

```
submit casm foo
```

ASM.COM (or MAC.COM)
Standard CP/M utility, for assembling the output of CASM.

There are several pseudo-operations that CASM recognizes as special control commands within a .CSM file. Each pseudo-ops should be indented at least one character away from the left margin, or else CASM will think it is a label. Recognized pseudo-ops are as follows:

FUNCTION <name> Each function must begin with a *FUNCTION* pseudo-op, where <name> is the name that will be used for the function in the .CRL file directory. No other information should appear on this line. Note that there is no

need to specify a complete list of contained functions at the start of a .CSM file, as was the case with the old CMAC.LIB method of CRL file generation.

EXTERNAL <list> If a function calls other C or assembly-coded functions, an **EXTERNAL** pseudo-op naming these other functions must follow immediately after the **FUNCTION** op. One or more names may appear in the list, and the list may be spread over as many **EXTERNAL** lines as necessary. Only function names may appear in **EXTERNAL** lines; data names (such as "external" variables defined in C programs) cannot be placed in "external" statements.

ENDFUNC
ENDFUNCTION This op (both forms are equivalent) must appear after the end of the code for a particular function. The name of the function need not be given as an operand. The three pseudo-ops just listed are the ONLY pseudo-ops that need to appear among the assembly language instructions of a ".CSM" file, and at no time do the assembly instruction themselves need to be altered for relocation, as was the case with CMAC.LIB.

INCLUDE <filename>
INCLUDE "filename" Thisop causes the named file to be inserted at the current line of the output file. If the filename is enclosed in angle brackets (i.e., <filename>) then a default CP/M logical drive is presumed to contain the named file (the specific default for your system may be customized by changing the appropriate #define in CASM.C). If the name is enclosed in quotes, than the current drive is searched. Note that you'll usually want to include the file BDS.LIB at the start of your .CSM file, so that names of routines in the run-time package are recognized by CASM and not interpreted as undefined local forward references…since CASM is a one-pass preprocessor, that would cause it to generate undesired relocation parameters for instructions having run-time package routine names as operands. Note that the pseudo-op **MACLIB** is equivalent to **INCLUDE** and may be used instead.

The format for a ".CSM" file is as follows:

```
;
; Make sure to indent pseudo-ops!
; (Anything starting in column 1 is presumed to be a label)
;
            INCLUDE        bds.lib

            FUNCTION       function1
    [   EXTERNAL       needed_func1 [,needed_func2] [,...]   ]
            code for function1
            ENDFUNC

            FUNCTION       function2
    [   EXTERNAL       needed_func1 [,needed_func2] [,...]   ]
            code for function2
            ENDFUNC
            .
            .
            .
```

Additional notes and bugs:

1. If a label appears on an instruction, it *must* begin in column 1 of the line. If a label does not begin in column 1, CASM will not recognize it as a label and relocation will not be handled correctly.

2. Pseudo-ops must **not** begin in column one, or else they will be mistaken for labels.

3. Forward references to EQUated symbols in executable instructions are not allowed, although forward references to relocatable symbols are OK. The reason for this is that CASM is a one-pass preprocessor, and any time a previously unknown symbol is encountered in an instruction, CASM assumes that symbol is relocatable and generates a relocation parameter for the instruction.

4. When a relocatable value needs to be specified in a **dw** op, then it must be the *only* value given in that particular DW statement, or else relocation will not be properly handled. In other words, only one 16-bit relocatable item is allowed per **dw** statement.

5. Characters used in symbol names should be restricted to alphanumeric characters; the dollar sign ($) is also allowed, but might lead to a conflict with labels generated by CASM.

8.2 The L2 Linker

L2, and this documentation, was originally written by Scott W. Layson for Mark of the Unicorn, Inc. Thanks go to Scott and Mark of the Unicorn, Inc. for placing L2 in the public domain and thus allowing all BDS C users access to this extremely useful tool. L2 was then modified by David Kirkland, mainly for integration with the CDB debugging package. Scott's original documentation has been modified by Leor Zolman to reflect the CDB-related extensions.

L2 is an alternative to CLINK for linking BDS C programs. A program linked with CLINK will have a jump table at the beginning of each function; calls to other functions are made indirectly through the table. L2 eliminates these jump tables, and adjusts indirect calls through them to go directly to the target function. Besides making the code imperceptibly faster, this has two real advantages: one, it makes the code smaller by 4% to 10% (the latter has been observed in a program containing many small functions which do little besides call a few other functions), and it allows SID to display the name of the target function of a call, simplifying debugging.

L2 seems to be complete enough to replace CLINK entirely. Its biggest advantage is that it's written in C, so that if you need some feature it doesn't have, you can just hack it in. However, its user interface is somewhat different.

A typical command line is

```
l2 foo bar -l bletch grotz -wa
```

Given this command, L2 will load all the functions in FOO.CRL and BAR.CRL (the program files). Then it will scan the libraries BLETCH.CRL, GROTZ.CRL, DEFF.CRL, and DEFF2.CRL (in that order) for functions that have been referenced but not linked. If there remain unsatisfied references, L2 will display a list of the needed functions and prompt for the name of a CRL file to scan; it will repeat this process until all references are satisfied (just like CLINK). Then it will write the resulting code to FOO.COM, display the link statistics, and write a symbol table (with the link stats appended) to FOO.SYM.

Note the following differences in the option-specification mechanism between L2 and CLINK:

1. L2 option names have varying lengths, while CLINK options all have single-character names.

2. When L2 options require arguments, a space **must** separate an option name and its argument. With CLINK, the spaces are optional. For example, "-m fubar" may not be written "-mfubar".

3. L2's options may not be combined; "-l –w", for example, may not be abbreviated "-lw".

Here is a complete description of available command line options:

–f <funcs> Reserves enough table size for <funcs> functions. (<funcs> is in decimal.) The default is 200. If you often link programs with more than 200 functions, you may wish to change the default — it's in setup() in L2.C.

–l CRL file names before the first "-l" on the command line will be treated as program files; CRL files after the first "-l" are treated as libraries. Subsequent "-l"s have no effect.

–m <name> <name> becomes the top-level function. This is the function initially called when the .COM file is run; by default, of course, it is "main". Note that, unlike with CLINK, the top-level function need not be the first

function in the first CRL file; it can be anywhere. –m also works with –ovl (see below).

–ovl <name> <addr> An overlay segment will be built instead of a root segment; the overlay will be linked to run at base address <addr> (entered in hex). <name> is the name of the root segment for which the overlay is being built; <name>.SYM, a symbol table produced with either L2 or CLINK, will be read in *before* the CRL files, to allow overlay functions to call root functions. The name of the top-level function in the overlay — i.e., the one that gets invoked by a call to the overlay base address — is by default not "main", but rather <firstcrl>, the name of the first CRL file in the L2 command line. The overlay segment is written to <firstcrl>.OVL. (See example below.)

–org <addr> This option is used to produce a root segment with base address <addr>, e.g., for use in generating code for ROM-ing. <addr> is entered in hex, and is the starting address of the code, not of RAM; the default is, of course, 0x100. (To link a program for a nonstandard CP/M, you need a C.CCC, DEFF.CRL, and DEFF2.CRL which have been assembled for that address. If you are running L2 on a nonstandard CP/M, you should change the default origin in setup() to 0x4300.) If you are using this option to generate code for ROM, be sure to use the "-t" option also (see below).

–t <addr> Works just like the CLINK "-t" option: sets the stack pointer to the given address at the start of the run-time package. This option MUST ALWAYS BE USED when "-org" is used to generate code for ROM. IF "-t" is NOT used, then the first two instructions of the resulting COM file will be:

```
lhld origin-100h+6
sphl
```

(where "origin" is normally 0x100 or 0x4300) while using "-t" causes the first two instructions to be:

```
lxi sp,<addr>
nop
```

–w A SID-compatible symbol table is written to <firstcrl>.SYM, where <firstcrl> is the name of the first CRL file listed in the command line. This table is normally produced in address order, not alphabetical order like CLINK's; see below for how to change this.

–wa A variation on –w. The link statistics, which are always displayed on the console at the end of linking, are also appended to the .SYM file. If the resulting .SYM file is read into SID, SID will complain by issuing its typical verbose error message "?", but then will work correctly. The big advantage of putting the stats at the end of the .SYM file is that one can

subsequently look at that file to see exactly how long the code was and where the externals started.

–ws Another variation on –w. This one writes the symbol table to <firstcrl>.SYM and the link statistics to <firstcrl>.LNK.

Because L2 is so large, it cannot always link large programs in a single pass. If it runs out of memory during linking, it will switch automatically to (very slow) two-pass mode. (If it says "Module won't fit in memory at all", you probably have a very large program file. Split it up or make it a library. If this doesn't work, you don't have enough memory to use L2.)

L2 is built from the source files L2.C and CHARIO.C. A typical compilation is

```
cc l2.c -e5500
cc chario.c
        (followed by either)
clink l2 chario (or) l2 l2 chario
```

If you want a shorter version of L2.COM, see the definitions of symbolic constants SHORTL2 and OVERLAYS immediately after the initial comments at the start of L2.C. These constants may be changed to yield several shorter configurations of L2, depending on which features you want to sacrifice.

8.3 The CMODEM Telecommunications Program

CMODEM is a communications program for transmitting files and connecting to other systems or networks as an ASCII terminal. The file transmission modes allow sending/receiving either binary or text files in either MODEM7 or straight-ASCII modes.

> **Author's note:** "CMODEM" is a cross between the original BDS TELNET program, Ward Christensen's MODEM program, its MODEM7 derivative, and a C version called XMODEM. It is also the result of taking the TELEDIT package as distributed with previous versions of BDS C and removing the text editing capabilities, since the RED package is now included for that purpose. [24]

Installation

Both the STDIO.H and HARDWARE.H header files should be properly configured for the target computer configuration before CMODEM is compiled.

The modes selected from the menu are:

24. The author wishes to thank Nigel Harrison for his work on the TELEDIT package

T: Terminal mode – no text collection

CMODEM behaves like an ASCII terminal. Eight-bit characters are sent and received; no parity bits are checked, inserted or removed.

To return to the selection menu the SPECIAL character is typed. The SPECIAL character of <ctrl>shift uparrow was chosen because it is unlikely to be struck accidentally. To change the SPECIAL character, recompile it with the desired #define SPECIAL …

X: terminal mode with teXt collection

Same as terminal mode above, except that any text characters received on the communication link are saved in a text buffer. The tab, newline and formfeed characters are also placed into the buffer; any other characters are discarded. When the internal text buffer is within 1000 characters of being filled, the console bell (alarm) sounds on every 16th character. When this happens the user should find a convenient time to suspend communication with the remote station so that the accumulated text can be saved (flushed) onto disk, as described below, before the buffer fills up completely and data is lost.

G: toGgle echo mode (currently set to echo)

Should not be toggled if the user is communicating in full duplex mode and receiving an echo from the remote station, or the user is in half duplex mode. Use this option to talk to another person running CMODEM, typically in between file transfers to inform the person of the next file to be transmitted.

F: Flush text collection buffer to text collection file

Flushes the text collection buffer accumulated in text collection mode. Does not close the file.

U: select CP/M User area

For users who have user areas, others should ignore this command.

V: select CP/M logical driVe

Select any of the disk drives available. The drive selected becomes the currently logged disk.

D: print Directory for current drive and user area

The current directory may be selected by using the **U** and **V** commands.

S: Send a file, MODEM protocol

Prompts for the name of the file to send, then waits for the receiver to "synch up".

The receiver must be using this program or one which uses the same MODEM protocol.

Returns to menu after completion, successful or not.

R: Receive a file, MODEM protocol

Prompts for the name of the file to be received, then waits for the sender to begin transmission. The sender must be using CMODEM or a program that employs the same MODEM protocol.

Q: Quit

Quits and returns to command level. If a text file has been accumulated in X mode, the user is asked whether or not he wants it saved.

SPECIAL:

Sends the SPECIAL character to the communication line, should that ever be necessary. The SPECIAL character is defined at compilation time by a **#define** statement at the top of the CMODEM.C source file.

Chapter 9

Auxiliary BDS C Libraries

This chapter describes several utility function libraries provided with the BDS C package.

9.1 BDS C v1.5 Compatibility Library

In order to compile programs written under earlier releases of BDS C (i.e., programs using the old buffered I/O library) without requiring that the programs be modified to conform to the new I/O library, the following two files have been provided:

BDSCIO.H
DEFF15.CRL

BDSCIO.H is the old standard header file (replace by STDIO.H), and DEFF15.CRL is the v1.50a DEFF.CRL file containing the compiled object code for the C-coded portions of the v1.50a library (the DEFF2.CRL functions are functionally equivalent, so the old version has not been provided.)

To compile and link a v1.50a source program "TEST.C" with the v1.6 compiler package, place the two files named above into the current directory with the source file and use the following procedure (in this example, we're compiling a source file named test.c):

```
cc test.c
clink test -f deff15
```

9.2 A BCD Function Package For BDS C

Copyright 1983, 1986
By Robert Ward

9.2.1 Description of Files

The bcd floating point package consists of these files:

MCONFIG.H This is a "header" file which serves the same function for all c files. Any time the package is changed, this file must be examined and modified as described in the section on package configuration.

BCD1.CSM
BCD2.CSM These are the source files for the assembly portion of the package. The first is the single function FH. All the real work is done in FH. The second file is the interface routines which provide the different calls to FH. This file includes source for FPADD, FPSUB, etc., but each of these functions in turn calls FH. BCD2 also contains lcnst, an encapsulating function which "knows" several constants useful to the transcendental functions.

BMATH.C Source code for all functions and routines written in c. Simple functions (abs, neg, assign) are included here as are input and output formatting functions.

DEMO1.C
TSTINV.C
LMATH.C Source code for demonstration programs. A COM file is provided for DEMO1.

BCD.CRL The complete library of float functions in relocatable form, ready to be linked to an application program. This file was compiled with precision set at 14. You may make an alternate version with more or less precision by following the instructions in the "configuration" section.

MATH.DOC
MATH.TXT This documentation in machine readable form. The first is in a form compatible with the C Users' Group formatter NRO. The second is the formatted output from NRO. More information on NRO is available through the C Users' Group.

9.2.2 Data Representation

In this package numbers are represented by a normalized fraction of PREC digits (where PREC may be adjusted by the user) stored as packed binary-coded decimal, an exponent in excess 64 notation and a sign bit. Each number requires PREC/2 + 1 bytes of storage. The fraction will be stored in the low address PREC/2 bytes, the exponent as the high order seven bits of the high address byte. The sign is stored as the least significant bit of the high address byte.

The fraction is stored with the least significant two digits in the low address byte. Within each byte of the fraction the least significant digit is in the right nibble. Thus with six digits of precision the number 1.23456e2 would be stored thus:

```
          +-----------+
     n    |  5    6   |
          +-----------+
     n+1  |  3    4   |
          +-----------+
     n+2  |  1    2   |
          +-----------+
     n+3  |1000010  0 | <-- sign bit
          +-----------+
             ^
          exponent
```

Under BDS C, space for such numbers may be declared simply as an array of char:

```
char number[PREC/2 +1];
```

Because the fractions are maintained in normalized form, the smallest number that can be represented is 1.0 e-64, regardless of the precision available. The largest number is a normalized fraction of all nines raised to the 63th power of 10. For twelve digits of precision this is:

```
9.99999999999 e 63
```

Only positive zeros are allowed; zero always has the exponent –1 (7E). Thus you should think of zero as always being normalized to .0. Several numbers and their hex representation in RAM are given below.

```
         1.0 --   00 00 00 00 00 .... 10 80
         0.1 --   00 00 00 00 00 .... 10 7E
normal   0.0 --   00 00 00 00 00 .... 00 7E
         -.1 --   00 00 00 00 00 .... 10 7F
         -.01 -   00 00 00 00 00 .... 10 7D
```

9.2.3 Testing For Zero

Since the normalized fractions are never zero in the most significant position unless the entire number is zero, the quickest way to test for num for zero is:

```
if (!num[PREC/2-1]) printf("\nNum is zero");
```

9.2.4 Rounding and Accuracy

Accuracy and predictable rounding have been primary design objectives during the creation of this package. All single computations should produce results accurate to one ulp (a five in the digit just to the right of the least significant digit). Thus the result of a single computation is always within 5 * 10 ^ (x-PREC) of being exact (where x is the exponent of the result) unless an error condition has occurred. To avoid loss of accuracy in the rightmost significant digits during divide, add and subtract operations, results with at least PREC + 2 digits are computed and THEN rounded. Rounding is always delayed until all other steps in the computation have been completed. Multiply uses all PREC digits of both operands in forming its result, maintaining all resulting digits until the resulting fraction has been normalized and rounded. The primitive functions "round up", the kind of rounding most persons consider "natural".

In addition to being rounded by the arithmetic functions, results will be rounded by printf if the precision specified is less than that available in the bcd number. This particular rounding may be disabled by changing three lines of the function _spr. The lines appear near the labels doe and dof. There are more complete instructions there.

9.2.5 Error Handling

Overflow, underflow and divide by zero errors are detectable through status flags maintained by fpadd, fpdiv, fpmult and fpdiv. Testing for errors is the programmer's responsibility. The program demo1 illustrates an approach using errm(). The programmer may construct more sophisticated mechanisms using fstat and fpstr. Error status is cleared when the status is checked by fstat, but not if checked by fpstr.

None of the routines validate inputs; they assume they have been given valid bcd numbers. Fprint is particularly sensitive to invalid data.

Underflow and Overflow always result in the least and greatest representable value respectively. The substituted result will always match the exact result in sign. Divide by zero similarly produces the greatest representable value.

The location ERRF in BCD2.CSM is used to store error status. Thus, this location's value will change during execution even though it is within the code block. Fstat and related functions locate ERRF by "dead reckoning" — they are hardwired with an offset from the entry point to FH. Any changes to the first few instructions of FH must be accompanied by appropriate adjustments to the status functions in BCD1.CSM. BCD2.CSM generates an error message during assembly as a reminder. You needn't attempt to correct the error (the code generated is correct), just remember to make the necessary adjustments.

9.2.6 The Return Values

All arithmetic functions in this package take pointers as their input arguments and return a pointer to their result. This allows the programmer to treat each function as if its value were the value of its result, at least when combining it with other functions. This allows nesting of computations for an effect similar to polish notation. For example, 2 + 4 * a * c could be written thus:

```
fpadd(
  fpmult(
    fpmult(a,c,result),
    atof(temp,"4"),result),
  atof(temp,"2"),result);
```

This arrangement calls for a programming style very similar to that used in three address assembly languages. In particular you will often find long sequences of calculations referring to the same "scratch pad" variable, e.g. result in the sequence above.

9.2.7 Transportability

Certain practices in drafting code will enhance the ease with which programs may be transported to other compilers. The primary concern is to somehow "flag" all floating point data structures so that they may be easily redefined under the new compiler (we will assume you are porting up to a machine with full C).

The parameterized **#define** preprocessor directive allows a "poorman's typedef" which not only enhances portability, but also improves readability. We suggest you include these definitions:

```
#define FLOAT(name) char name[PREC/2 + 1]
```

Pointers need to be provided separately, e.g.:

```
#define FLTPT(name) char *name
```

With these defines, one would declare space for a number and for a pointer to a number like this:

```
FLOAT(number);
FLTPT(pointer);
```

To transport the resulting code to a compiler with built in floats, the declarations may be corrected by changing the defines to:

```
#define FLOAT(name) float name
#define FLTPT(pointer) float *pointer
```

If data types have been declared consistently, using this technique, then all that remains is to create a set of functions (under the new compiler) which look like those in this package. The new fpadd, for instance, would look like this:

```
float *fpadd(op, op2, ans)
float *op, *op2, *ans;
{
        *ans = *op + *op2;
        return ans;
}
```

9.2.8 Configuration

You should be warned that ERRF in the function FH is a magic location. If any code changes are made preceding the appearance of ERRF (changes which would affect its location relative to the function entry point) then the appropriate equates must be changed in BCD1.CSM. See the section on Error Handling for more information.

9.2.9 Changing Precision

This is a simple change, so long as your target precision is 4-20 digits. You need to change the PREC equate in BCD1.CSM, BCD2.CSM and the PREC define in MCONFIG.H. Then reassemble the .csm files, recompile math.c and build a new CRL file as detailed below. C programs using the math package will also need to be recompiled, so the change in MCONFIG.H can be reflected in their CRL image.

9.2.10 Rebuilding BCD.CRL

1. Compile BMATH using

   ```
   cc bmath
   ```

2. Process BCD1.CSM and BCD2.CSM following the instructions supplied with CASM.

3. You should now have new versions of BMATH.CRL, BCD1.CRL and BCD2.CRL, all that remains is to merge them into a single relocatable library. Begin by copying BMATH.CRL to BCD.CRL, thereby creating the base of your new library.

4. Using clib, open BCD, BCD1 and BCD2 and transfer all the files in BCD1 and BCD2, one at a time, to BCD.

5. List the contents of BCD. In particular, check that each of the functions you meant to add to it actually appears. Be certain to close BCD before exiting clib or your work will not be recorded on the disk.

9.2.11 Linking to the BCD Functions

We have deliberately avoided the use of external data in order to make the package's operation more transparent to the user. The demonstration programs are good illustrations of the coding requirements. These must be the first code lines of any program using the package:

```
#include "stdio.h"
#include "mconfig.h"
```

The math package has no other impact on the compilation process. When you link you must use this form (to force the loading of the bcd versions of printf and scanf):

```
clink program bcd <other files and options>
```

For example, demo1 is produced by:

```
cc demo1
clink demo1 bcd
```

9.2.12 BCD Package Function Summary:

int abs(num)
int num; Returns the absolute value of the integer num. Included because some of the bcd functions require it.

char *assign(dest, source)
char *dest, *source; Copies the floating point number *source to *dest.

char *atof(result, string)
char *result, *string; Accepts a wide variety of numeric representations in *string and converts them to a floating point number stored at *result. Returns pointer result. The representation at *string must be null terminated. Roughly equivalent to a call to scanf with a %f format string, but easier to use (and somewhat less expensive). MAXLINE in BMATH.C sets the upper length of the string atof will scan. MAXLINE may be made arbitrarily large, but larger values will normally just let bugs hide longer.

char *errm() Returns a pointer to an appropriate error message if any error flags have been set since the last call to errm. The error message is a null-terminated string of the following form:

```
<type> {<type>} ERROR
```

where <type> is one or more of the following:

```
DZ   --   Attempt to divide by zero
UF   --   Exponent underflow
OF   --   Exponent overflow
```

Errm checks error status flags through a fstat call, thus it also clears the flags, implying that if the program needs access to error status it should first do an fpstr call and then call errm to print warnings for the user.

char *exp(x,y,result)
char *x,*y,*result; Computes x^y for floating point x and y. Stores answer at *result, returns result. Neither x nor y may point at result. Uses the relation:

```
x^y = lginv(y*log(x))
```

See lninv for comments about accuracy.

char *fneg(num,out)
char *num,*out; Performs the operation

```
*out = *num * (-1)
```

by manipulating the sign bit of *out.

char *fpabs(num,result)

char *num,*result; Constructs the absolute value of *num at *result. Works correctly if num == result, allowing the call:

```
abs(num,num)
```

to change num to its absolute value. Returns a pointer to result.

char *fpadd(op1, op2, result)
char *op1, *op2, *result;
Expects floating point numbers in strings *op1 and *op2. Inserts the sum of *op1 and *op2 into the string space at result (result MUST point to available space). Returns the pointer result. Performs the operation:

```
*result = *op1 + *op2
```

On underflow sets *result to the smallest representable value with the same sign as the actual result. On overflow, sets result to the largest representable value with the same sign as the actual result. When the result has more than PREC digits of precision, it is rounded at the right as described in the section on rounding.
All arguments are copied to workspace within the function, allowing the same variable to appear as both operands or as an operand and the result. You are guaranteed that the operands will be copied before the value at result is changed. Thus

```
fpadd(op,op,op)
```

performs the implicit assignment *op = 2 * (*op). More useful, running totals may be computed using the form:

```
fpadd( next, subtotal, subtotal)
```

int fpcmp(op1, op2)
char *op1, *op2; Compares the absolute value of the floating point number *op2 to that of the floating point number *op1. Returns:

```
1 if op1 < op2
0 if op1 = op2
-1 if op1 > op
```

char *fpdiv(dividend, divisor, quotient)
char *dividend, *divisor, *quotient;
Performs the operation:

```
*quotient = *dividend / *divisor
```

Similar in behavior to fpadd (see). Returns the pointer to quotient as its value. Sets the divide by zero flag and returns the maximum representable value if *divisor==0. Responds to overflow and underflow conditions in same fashion as does fpadd. Always computes PREC + 2 digits internally. Allowing for a possible lead zero, this guarantees PREC + 1 meaningful digits, yielding enough information for predictable rounding to PREC digits.

char *fpmult(op1, op2, product)
char *op1, *op2, *product;

Performs the operation:

```
*product = *op1 * *op2
```

Similar in behavior to fpadd(see). Returns the pointer product as its value. Sets the overflow or underflow flags as appropriate. Maintains PREC + 2 digits internally, guaranteeing PREC meaningful digits after rounding.

int fpstr() Identical to fstat except it does not clear the current error flags. (See fstat for details.)

char *fpsub(op1, op2, result)
char *op1, *op2, *result;

Identical to fpadd except that the sign of FH's internal copy of op2 is changed before the operation proceeds. Performs:

```
*result = *op1 - *op2
```

Affects underflow and overflow flags as described for fpadd. Returns the pointer result.

char *frnd(num, pos, result)
char *num, *result;
int pos;

Rounds *num so that the result has pos digits of rounded fractional information. For example:

```
atof(op1,"123.555555555");
frnd(op1,4,op1);
```

will produce 123.555600000 in *op1.
If pos lies outside the precision of the representation, the function tries to behave intelligently. If pos is to the right of the rightmost significant digit, *result will equal *num. If pos is to the left of the leftmost significant digit, *result will be zero.
Performs correctly if result == num.
WARNING!! Do not confuse this with the function fprnd which is used internally by printf. For details on fprnd, see the code in BMATH.C.

int fscmp(num1, num2)
char *num1, *num2;

Compares the floating point numbers *num1 and *num2, returning an integer indicating the following relations:

```
1 if num1 < num2
0 if num1 = num2
-1 if num1 > num2
```

This is the signed version of fpcmp (see).

int fstat() Returns an integer whose lower byte contains the current error flags. Clears the flags in the process. The error assignments are:

```
bit 2:  exponent underflow
bit 1:  exponent overflow
bit 0:  divide by zero
```

int ftoi(num)
char *num;

Converts the floating point number *num to an integer and returns the resulting integer. On overflow, returns the least significant sixteen bits of the true value. Always truncates fractional portions (as opposed to rounding prior to conversion). Overflow errors are not detected.

char *ftrunc(num,result)
char *num,*result;

Constructs a number in result which represents only the whole number portion of num. Returns result.

char *itof(result, source)
char *result;
int source;

Converts the integer source to floating point representation and stores at *result. Returns result.

char **lcnst()

Does nothing more than return a pointer to a table of constants. To print pi to precision places, use this:

```
char **const;
const = lcnst();
printf("\n%g",const[17]);
```

The other constants and their relative positions:

```
const[0] = log(2)
for i=1 to 7
const[i] = log(1+(10^(-i)))
const[8] = 0
const[9] = 2/ln(10)
const[10]= 1
const[11]= 5
const[12]= 10
const[13]= 20
const[14]= ln(10)
const[15]= 2
const[16]= log(e)
const[17]= pi
```

char *lginv(x,result)
char *x,*result;

Constructs the common antilog of x in result (result = 10^x). With 14 digits of precision, accurate to 12 places. May never terminate if true result would be larger or smaller than the largest or smallest representable values.

char *ln(x, result)
char *x, *result;

Computesthe natural log of the absolute value of x. Stores value in *result, returns result. Uses the relation:

```
ln(x) = ln(10)*log(x)
```

With fourteen digits of precision, twelve are reliable.

char *lninv(x,result)
char *x,*result; Computes e^x where e is Napier's constant, 2.7182818… Returns a pointer to result, where the answer is stored. Both x and result are assumed to be floating point numbers. x must point to a different space than result.
WARNING!!! If e^x > MAX where MAX is the largest representable number then the routine may never finish.
This function uses the relation

```
e^x = lginv(log(e)*x)
```

With fourteen digits of precision, twelve are reliable.

char *log(x,result)
char *x,*result; Computes the common (base 10) logarithm of x. Puts value in *result, returns result. With fourteen digits of precision, 12 are reliable.

int mag(x)
char *x; Returns an integer corresponding to the magnitude (exponent) of the floating point number x.

int printf(format, arglist)
int sprintf(string,format,arglist)
int fprintf(stream,format,arglist)
Versions of the standard formatted output function which support f,F,e,E,g,G, formats in addition to those explained elsewhere. See the demonstration programs for examples. In general, **%f** formats bcd numbers as

```
ddd.pppp
```

where ddd is the whole portion of the number and there are precision digits p.
%e formats bcd numbers as

```
d.ppppe+xx
```

where there is always one digit d before the decimal point and precision digits p after it. The exponent will always be printed with a sign and two digits.
%g causes the shortest representation (e or f) to be used and trailing zeros are suppressed. The uppercase forms of each merely cause the e in the exponential form to be printed uppercase.

char *_restlg(x,result)
char *x,*result; This function computes the common log of x for restricted values of x such that 0 < x < 10. With fourteen digits of precision, 12 are reliable.

char *scale(num, k, result)
char *num, *result;
int k; Performs the operation:

```
*result = *num * 1.0ek
```

by manipulatingthe exponent portion of *num. Does NOT check for out of range errors. Performs correctly when called with num == result.

int scanf(format, arglist)
int sscanf(string,format,arglist)
int fscanf(stream,format,arglist)

Versions of the standard formatted input routine with %f,%g,and %e conversion specifications added. In these routines %f,%g, and %e are equivalent.
Returns the number of assignments made. Acceptable forms include:

```
n       (where n is an arbitrarily
        long string of digits)
+n
+n.n
+.n
.n
-n
-n.n
-.n
fex     (where 64>x>0 and f is one
        of the forms above)
fe-x
fe+x
fEx
fE+x
fE-x
```

Also acceptable are the above forms with leading blanks. There may NOT be any blanks embedded in the number.

char *zfl(num,pos)
char *num;
int pos;

Used internally by frnd to zero all digits right of the pos'th in the mantissa of *num. Does nothing if called with pos <0 or pos > PREC – 1. Returns num, but unlike other functions it modifies the input *num.

9.3 A Long Integer Package for BDS-C

Rob Shostak
August, 1982

9.3.1 Introduction

This package adds long (32-bit) signed integer capability to BDS C much in the same spirit as Bob Mathias's floating point package. Addition, subtraction, multiplication, division, and modulus routines are provided as well as comparison, assignment, and various kinds of conversion.

Each long integer is stored as an array of four characters. A long integer x is thus declared by:

```
char x[4];
```

The internal representation is two's complement form, with the sign (most significant) byte as the first byte of the array. For most purposes, however, you needn't be concerned with the internal representation.

Most of the routines that operate on longs take three arguments, the first of which points to where the result is to be stored, and the other two of which give the operands. For example, given longs x, y, and z (all declared as char[4]),

```
ladd(z,x,y)
```

computes the sum of x and y and stores it into z, which is returned as the value of the call. Note that the result argument may legitimately be the same as one (or both) of the operand arguments (for instance, ladd(x,x,x) does "the right thing").

The package is written partly in C and partly (for speed and compactness) in 8080 assembly language. To use it, simply link LONG.CRL into your program. A description is given below for each routine.

itol(l,i)
char l[4];
int i;

Stores the long representation of the 16-bit integer *i* into *l*, and returns *l*.

atol(l,s)
char l[4];
char *s;

Stores the long representation of the Ascii string *s* into *l*, and returns *l*.
The general form of *s* is a string of decimal digits, possibly preceded by a minus sign, and terminated by any non-digit.

ltoa(s,l)
char *s;
char l[4];

Stores the Ascii representation of long *l* into string *s*, and returns *s*. The representation consists of a null-terminated string of Ascii digits preceded by a minus sign if *l* is negative. *s* must be large enough to receive the conversion.

ladd(r,op1,op2)
char r[4];

Stores the sum of longs *op1* and *op2* into *r*, and returns *r*. *op1* or *op2* may be used for *r*.

lsub(r,op1,op2)
char r[4];
char op1[4],op2[4];

Similar to ladd, but computes *op1 – op2*.

lmul(r,op1,op2)
char r[4];
char op1[4],op2[4];

Similar to ladd, but computes *op1 * op2*.

ldiv(r, op1, op2)
char r[4];
char op1[4], op2[4];

Similar to ladd but computes the integer quotient *op1 / op2*. If *op2* is zero, zero is computed as the result.

lmod(r, op1, op2)
char r[4];
char op1[4], op2[4];

Similar to ladd but computes *op1* mod *op2*. If *op2* is zero, zero is computed as the result.

lcomp(op1,op2)
char op1[4], op2[4];

Compares longs *op1* and *op2*, and returns one of (the ordinary integers) 1, 0, –1, depending on whether (*op1 > op2*), (*op1 == op2*), or (*op1 < op2*), respectively.

lassign(dest,source)
char source[4],dest[4];

Assigns long *source* to long *dest*, and returns pointer to *dest*.

ltou(l)
char l[4];

Converts long *l* to unsigned (by truncation).

```
utol(l,u)
char l[4];
unsigned u;
```

Stores the long representation of unsigned *u* into *l* and returns *l*.

9.3.2 Implementation Details

Most of the work in the routines above is done by a single 8080 assembly-language function called *long*, the source for which is found in the file LONG.CSM (available from the C User's Group). The remainder of the package resides in LONG.C. Note that most of the primitives described above simply call *long*, passing it a function code (that tells it what operation is to be performed) together with the arguments to be manipulated.

The library object file DEFF2.CRL contains the workhorse function *long*, the source for which is in DEFF2D.CSM. The source file LONG.C needs to be compiled, yielding LONG.CRL. When linking programs that use the long integer package, the *long* library should be included on the linker command line.

Appendix A

Dynamic Overlays in C Programs

In order to allow C programs to be longer than physical memory without resorting to the *exec* and *execl* library functions (which may indeed get the job done, but resemble "chain" operations more than true segmentation tools), a set of capabilities has been built into the CLINK program to make program segmentation possible. The general idea is to have one copy of a **root segment** always remain in memory (at the base of the TPA) containing the C run-time package, the "main" C function, and any other functions that more than one overlay segment might need. The root segment controls the loading of overlay segments in higher memory, and each overlay segment, when loaded into memory somewhere above the root segment, can take advantage of run-time package entry points within the root segment as well as function entry points in any lower-level overlay segments (as well as the root segment).

Normally (i.e., when overlays are not being used), the run-time environment of an executing C program looks something like this:

```
-------------------------------------------------------------------------------
low memory:  base+100h:  C.CCC run-time utility package (csiz bytes)

              ram+csiz:  start of program code
                         ... (program code) ...
                xxxx-1:  end of program code

                  xxxx:  external variable area (y bytes long)
                         ... (external data) ...

                xxxx+y:  free memory,
                                available for
                                        storage
                                             allocation

                  ????:  as low as the machine stack ever gets
                                local data, function parameters,
         machine stack:         intermediate expression results,
                                etc. etc.
high memory:      bdos:  machine stack top (grows down)
-------------------------------------------------------------------------------
```

Memory Map 1

Note that *xxxx* is the first location following the program code and *y* is the amount of memory needed for external variables.

To incorporate overlays, it must first be decided just where the swapped-in overlay code is to reside in memory. One possibility is to locate the overlay swapping area between the end of

root segment code and the start of the external data area. Here is the modified memory map, accommodating this method of handling overlays:

```
-----------------------------------------------------------------------------
low memory:  base+100h:  C.CCC run-time package (csiz bytes)
              ram+csiz:  start of root segment code
                         ... (root segment code) ...
                zzzz-1:  end of root segment code

                  zzzz:  start of overlay area
                         ... (overlay area) ...
                xxxx-1:  end of overlay area

                  xxxx:  external variable area (y bytes long)
                         ... (external data) ...

                xxxx+y:  free memory,
                                available for
                                        storage
                                             allocation

                  ????:  as low as the machine stack ever gets
                                local data, function parameters,
         machine stack:         intermediate expression results,
                                etc. etc.
high memory:      bdos:  machine stack top (grows down)
-----------------------------------------------------------------------------
```

Memory Map 2

Note that *zzzz* is where overlay segments get swapped in, guaranteed that the longest segment doesn't reach *xxxx*.

It is also possible (but not as secure) to put the overlay area **after** the external data area. The memory map for this alternative configuration is as follows:

```
-----------------------------------------------------------------------------
low memory:  base+100h:  C.CCC run-time utility package (csiz bytes)
              ram+csiz:  start of root segment code
                         ... (root segment code) ...
                xxxx-1:  end of root segment code

                  xxxx:  external variable area (y bytes long)
                         ... (external data) ...
             xxxx+y-1:  end of external data area

                xxxx+y:  start of overlay area (ssss bytes long)
                         ... (overlay area) ...
         xxxx+y+ssss-1:  end of overlay area

           xxxx+y+ssss:  <unused memory>

                  ????:  as low as the machine stack ever gets
                                local data, function parameters,
         machine stack:         intermediate expression results,
                                etc. etc.
high memory:      bdos:  machine stack top (grows down)
-----------------------------------------------------------------------------
```

Memory Map 3

Note that the storage allocation functions (*alloc* and *sbrk*) always start obtaining memory from the area immediately following the end of the externals. If you plan to use the storage allocation

functions (*alloc*, *free*, *sbrk*, *rsvstk*) in your program under this scheme, remember to initially call the *sbrk* function with argument *ssss*, the size of the overlay area. Otherwise the storage allocator will begin to allocate memory within the overlay area.

In an attempt to limit diversion for the remainder of this document, I will assume that the **original** overlay scheme is being implemented as shown in Memory Map 2.

OK, with the generalities out of the way, let me say something about just how to create "root" segments and "overlay" segments with BDS C. First of all, we would like all functions defined within the root segment to be accessible by the overlay segment(s)…this is accomplished by causing CLINK to write out a symbol table file containing all function addresses to disk when the root segment is linked. The **–w** option to CLINK will do the trick; this symbol table will be used later when linking the swap-able segments.

When linking the root segment, use the **–e** option to set the external data area location. Keep in mind that there must be enough room below[25] the externals to hold the largest overlay segment at run time. If the **–e** option is omitted, CLINK will assume the external data starts immediately after the end of the root segment code and conflict with the overlay area (thus, **–e** may only be omitted when using the second overlay scheme as shown in Memory Map 3).

Within the code of the root segment, then, a swap-able segment is loaded into memory from disk by saying:

```
swapin(name,addr); /* read in a segment; don't run it */
```

where *addr* is the location following the last byte of root segment code. You can find this value by linking the root once without giving the **–e** option and reading the **–s** statistics written to the console after the linkage.

NOTE: Because CP/M is a sector-oriented operating system, the length of the file loaded into memory by *swapin* is always an integral number of 128-byte sectors long. That means that you should always allow for a little extra space at the end of the overlay segment memory area, up to 127 bytes more than the length of the actual overlay segment code (as displayed by the CLINK statistics summary).

To actually execute code within the overlay segment, you have to call the appropriate memory address indirectly using a pointer-to-function variable.

Here is an example. We'll declare a pointer-to-function variable called *ptrfn*, swap in a segment named *ovl1* at location 3000h, and call the segment. The sequence would look like this:

```
int (*ptrfn)();  /* can be whatever type you like */
ptrfn = 0x3000;
...
if (swapin("ovl1",0x3000) != ERROR)  /* check for load error */
        (*ptrfn)(args...);  /* if none, call the segment */
...
```

25. I'm using the term "below" in the sense that low memory is "below" high memory; graphically, at least in the preceding memory maps, "below" means toward the top of the page.

Note that the overlay code might not return any value after being called, but the pointer-to-function must be declared with SOME kind of type. Use **int** if nothing else comes to mind. When a segment is invoked, as above, control passes to the segment's "main" function. There is no reason at all to require parameters to be of the "argc" and "argv" form; there is nothing special about a "main" function other than the property it has of getting called first. The "main" function within the swapped-in segment is the **only** entry point allowed for the segment.

A simple *swapin* function is given in the standard library. It can be expanded to detect an attempted load over the external data area by comparing the last address loaded with the contents of location 0115h…if you've never done any low-level hackery, you get the value of the 16-bit address at location 0115h by using indirection on a pointer-to-integer (or –unsigned.) Note that location 0115h *always* contains a pointer to the start of the external data area.

Now we know how to do everything except actually create an overlay segment. OK, an overlay segment is basically just a normal C program, having a "main" function just like the root segment, except that the C.CCC run-time utility package is NOT tacked on to the front of an overlay segment (the C.CCC run-time package in the root segment will be shared by everyone.) The other difference between an overlay segment and the root segment is the load address; while the root segment always loads at the base of the TPA, an overlay segment may be made to load anywhere. Once you've compiled the overlay segment, you give a special form of the CLINK command to link it:

```
A>clink segment-name -v -l xxxx -y symbol-file [-s ...] <cr>
```

segment-name is the name of the CRL file containing the segment, **–v** indicates to CLINK that an overlay segment is to be created (so that C.CCC is not attached), and **–l** *xxxx* (letter ell followed by a hex address) indicates the load address for the segment. The **–y** option yanks in the symbol file created by the root segment. If this is omitted, then CLINK yanks in fresh copies of functions like "PRINTF" and "FOPEN", etc., even if they have already been linked into the root segment. By reading in the symbol table from the root segment, it is insured that any routines already linked in the root will be made available to the overlay segment. The root segment, though, cannot know about functions belonging to overlay segments through the use of a symbol table. That would require some kind of mutually referential linking system beyond the scope of this package. Oh well.

When linking an overlay segment, you might also specify **–s** to generate a statistics map on the console, and **–w** to write out an augmented symbol table containing not only the symbols read in from the root segment's symbol file, but also the swap-able segment's own symbols. This new symbol file may then be used on another level of swapping, should that be desired.

Time for an example: Let's say you've got a program ROOT.C, which will swap in and execute SEG1.C and then overlay SEG1.C with SEG2.C. ROOT.COM loads at 100h and ends, say, before 3000h. We'll load in the segments at 3000h, and set the base of the external data area to 5000h (this assumes neither segment is longer than 2000h.)

The linkage of ROOT would be:

```
A>clink root -e 5000 -w -s <cr>
```

This tells CLINK that ROOT.COM is to be a root segment (since no **–v** option was given), the externals start at 5000h, a symbol file called ROOT.SYM is to be written, and a statistics summary is to be printed to the console.

The linkage of each overlay segment would appear as follows:

```
A>clink seg1 -v -l 3000  -y root -s -o seg1. <cr>
```

This tells CLINK that SEG1.COM is to be an overlay segment (because of **–v**) to load at location 3000h, the symbol file named ROOT.SYM should be scanned for pre-defined function addresses, a statistics summary should be printed after the linkage, and the object file is to be written out as SEG1 (as opposed to SEG1.COM, to avoid accidentally invoking it as a CP/M command.)

Appendix B

Customizing The Run-Time Environment

B.1 Standard vs. Customized Environments

In its most common and simple usage, BDS C produces a transient command file ready to execute in response to a command typed at the Console Command Processor (CCP) under CP/M. Such a command file always executes in read/write memory located at the base of the TPA (transient program area) at address 100h. Under these normal circumstances, the run-time package (C.CCC) and its private read/write memory area occupy the first 1500-or-so (decimal) bytes of the command file, and the compiled code (beginning with the "main" function) follows immediately thereafter. This scenario may be termed, for the purposes of this appendix, as the **standard run-time environment** of a C program.

B.2 Simple Run-Time Package Customization

Most of this appendix describes how to alter the run-time environment for totally arbitrary system configurations, a procedure that requires the modification of both the run-time package and much of the machine-coded portions of the library.

There are certain aspects of the standard run-time environment, such as control over whether or not user areas are recognized at run time, that only require modification of C.CCC run-time package module options (not any of the library functions). To make one of these kinds of changes, just follow these steps:

1. Modify CCC.ASM as required by changing the EQU statements at the top of the file. Do not make any changes except for those well documented as customizable options.

2. Assemble CCC.ASM with whatever assembler you have handy, yielding CCC.HEX as the result of the assembly. If you know how to generate a binary image, go ahead and do so, naming it C.CCC, and skip to the last step.

3. If your assembler outputs a .HEX file, either use LOAD.COM or CLOAD.COM (source in CLOAD.C) to create a binary image. If you used LOAD, rename CCC.COM to C.CCC. If you used CLOAD, rename CCC.CRL to C.CCC.

4. Replace your old C.CCC with the new version. You are done.

B.3 Creating New Customized Environments

In order to generate code that runs at a different location in memory or in ROM (or both), it is necessary to **customize** the run-time environment and then to follow special compilation and linkage rules to insure consistency between separately compiled and/or assembled modules of a program. If any change involving either the insertion, deletion or rearrangement of code is made to the run-time package, that change then constitutes a **customization** of the run-time environment. Most assembly-language-coded library functions[26] reference the absolute addresses of code and data in the run-time package; therefore, any customizations made to the run-time package must be reflected in all the CSM library functions which are to be invoked in the new customized run-time environment.

The general procedure can be outlined as follows (don't actually try anything from just this outline; detailed instructions will follow later):

1. Customize the CCC.ASM run-time source module as necessary. Change the BDS.LIB header file to accurately reflect those changes made to CCC.ASM.

2. Re-assemble all needed portions of the CSM function library, using the new BDS.LIB created above.

3. Recompile the C-coded portions of the library, making sure to use appropriate CC command-line options to reflect the customized environment.

4. Be careful when assembling and linking modules for the new environment; watch out especially for mix-ups between standard and customized object files.

Here is a tip for creating customized run-time environments: do it all in a "user area" that is different than the one where your standard environment files are kept. A good starting point is to copy all needed source and command files to a new user area, and work in that area exclusively to both create the customized environment and to develop applications under it. The following files are all needed at some point for the following procedures: Your favorite text editor, CC.COM, CC2.COM, CLINK.COM (or L2.COM), CASM.COM, CASM.SUB, ASM.COM (or MAC.COM), DDT.COM (or SID.COM), CLOAD.COM, CLIB.COM, CCC.ASM, *.CSM, BDS.LIB, STDIO.H, STDLIB*.C.

> IMPORTANT: If you have configured your CC.COM, CLINK.COM, CASM.COM or L2.COM command files to search a "default" library disk and/or user area for common library files, you should "un-do" those configurations when working with customized run-time environments. The best way to do that is to have a resident copy of each such command file in your work area configured to search <u>only</u> the current drive and user area for <u>everything</u>.

A target program may need to run under CP/M or stand alone, in ROM or in RAM, at a different location in memory, or with a different set of initializations. Changing any one of these characteristics, or in fact making just about any kind of change at all in the run-time package,

26. The CSM files contain the source code, DEFF2.CRL is the assembled object library

produces a new customized environment and requires the re-creation of both the function library and run-time package object module, collectively to be known as the "run-time library".

By definition, the term "customized environment" implies unique variations from implementation to implementation. This makes it difficult to describe all the possible variations; therefore, a general procedure for making a new run-time library will be presented.

Here is the more detailed procedure for customizing the run-time library:

1. Starting with fresh copies of all the files listed above, find CCC.ASM and go to work on it with your favorite text editor. Alter all the appropriate EQU statements at the beginning of the file to reflect your desired run-time environment. See later sections in this appendix for details on this step.

2. Using ASM.COM or MAC.COM, assemble CCC.ASM yielding CCC.HEX and CCC.PRN.

3. Examine CCC.PRN to find out what value was assigned to the label "RAM", at the start of the read/write data area declarations near the end of the file. If you've chosen the "CPM" symbol to be TRUE then the value you are looking for appears in hex along the left margin of the line which reads "RAM equ $" near the end of the file. Otherwise, you've made "CPM" false and you had to enter the value of "RAM" explicitly in an EQU statement near the beginning of the source file.

4. Having determined the value of RAM, edit BDS.LIB and give the symbol similarly named "RAM" in that file the same exact value. Make sure all equated symbols in BDS.LIB match changes you may have made to CCC.ASM.

5. Temporarily rename BDS.LIB to be BDS.ASM, and assemble it to yield BDS.PRN. Compare BDS.PRN to CCC.PRN, to make sure all addresses and symbols match perfectly between the two files. If you find a discrepancy, track it down and fix it by altering either CCC.ASM or BDS.ASM accordingly. When all the values match, rename BDS.ASM back to BDS.LIB.

6. Convert the CCC.HEX file created back in step 2. into a binary image named C.CCC. If you know how to do this already, do it and proceed to the next step. Otherwise the following sub-procedure is presented:

    a. Compute the ddt <offset> for CCC.HEX by subtracting the origin address of the run-time package from 100h. For example, if you've set the origin symbol in CCC.ASM to 1000h, the <offset> would be the value of (100h – 1000h), which is F100h.

    b. Compute the <size> in 256-byte sectors of the run-time package object code. Given CCC.PRN, use whichever method for this step you feel comfortable with. If you are not sure of your result, round up. A <size> value too high will still work correctly, but a value too low will bomb either the linker or the target program at run time.

    c. Perform the following sequence:

```
A>ddt
-iccc.hex
r<offset>
^C
A>save <size> c.ccc
A>
```

where <offset> and <size> are the values computed in steps 1. and 2. You now have a C.CCC run-time package object module ready for linkage.

7. Create a CSM library source module (or set of modules) containing all the functions you expect you might use in target programs for the customized environment. Note that some CSM functions may not be useful for all environments; for example, most functions in DEFF2C.CSM would be useless when running under a non-CP/M environment. Making sure to use the modified BDS.LIB, put each new CSM source file through the CASM procedure, yielding a new CRL file or set of CRL files. This file (or set of files) is now ready to be linked with the new C.CCC and target programs.

8. Compile STDLIB1.C and STDLIB2.C with the "-M <origin>" option to CC.COM. The value of <origin> must be the same as that used in the previous steps of this procedure: the starting address of the run-time package. You may want to combine STDLIB1.CRL and STDLIB2.CRL into a new DEFF.CRL, using CLIB.COM. Make sure not to ever confuse this new DEFF.CRL with the one used in standard environment compilations.

9. The run-time library environment is now ready to use. When compiling C source programs, use the CC.COM **–M** option to inform the compiler of the new run-time package <origin> address (if different from 100h).

10. With CLINK, Use the **–L**, **–T** and **–E** options to specify <origin> address, top of r/w memory and base of external data area, respectively, for the target program. The L2 linker uses different option names (-ORG, –T and –E) to specify these same the same things.

B.4 Making Code Run Without CP/M

When programs are to be placed into read-only memory (ROM), that usually means **not** under CP/M and often not at memory address 100h. The technical procedures described above for building a new run-time package and library all apply here, and a few new rules come into play.

In CCC.ASM, the "CPM" symbol should be equated to FALSE and any other symbols in that cluster should be altered as needed. The effect of making the "CPM" symbol false is to eliminate all the CP/M-specific support routines from the run-time package. This will significantly reduce the size of the run-time package and, therefore, the size of the resulting compiled program (every byte counts, especially when the target system is ROM-based). Certain categories of library functions (such as the file I/O ones) will cease to have any meaning under this new configuration, so be sure to put together your new CSM source libraries carefully.

There are three important address attributes of the run-time environment that merit close examination. They are: 1) the origin address of the program, 2) the origin of the run-time package scratch pad RAM area, and 3) the exit address where control is passed following program termination (if ever). The symbol names for these three addresses are, respectively, "ORIGIN", "CPM" and "EXITAD". When the "CPM" symbol is TRUE, these values are all computed automatically in the run-time package source because of the known nature of the CP/M environment. Once "CPM" has been made FALSE, however, these values must be explicitly set by the user. The section of code in CCC.ASM where these values are entered appears as follows:

```
        IF NOT CPM              ;fill in the appropriate values...
ORIGIN: EQU     NEWBASE
                                ;Address at which programs are to run
RAM:    EQU     WHATEVER        ;run-time package scratch pad RAM area
EXITAD:
        EQU     WHENDONE        ;where to go when done executing
        ENDIF
```

All three of these equates should be configured to reflect the desired run-time environment. Note that the value of "ORIGIN" must be used as the argument to the "-M" option of CC.COM when compiling C source code for this environment, and also used as argument to the "-L" option of CLINK.COM (or "-ORG" of L2.COM) during linkage.

A Note About RAM areas:

There are two distinct RAM areas occupying any particular BDS C run-time environment. They are the "stack area" and the "run-time package scratch pad area".

The stack area is where all local ("automatic") data storage is allocated and where intermediate values and function parameters are pushed/popped. The address of the stack area **must** be specified by use of the **–t** CLINK option for non-CP/M environments. The standard value to supply for the stack area is the address of the byte following the **last** (highest) byte of the run-time environment's RAM area (since the stack grows down, never using its initial location).

The "other" RAM area is the run-time package scratch pad area, as specified by the "RAM" symbol in the discussion above. This refers to a relatively small area of memory needed by the run-time routines for temporary storage and miscellaneous dirty hacks requiring RAM. This value should be set to the **first** (lowest) location in the run-time environment's RAM area. After you assemble the run-time package, examine the PRN file for the address where the scratch pad area ends, and make sure there is enough room between that address and the end of the RAM area to accommodate the stack in its "worst" case of nested storage allocation. See Chapter 2 for a discussion of how much space the stack can take up.

This concludes our nitty-gritty discussion on customizing BDS C for non-standard environments. While it may take you several iterations of the procedure to become completely familiarized with it, the resulting compactness of code and high degree of environmental control should make for efficient, well-tailored applications.

Appendix C

BDS C Evolution: A Version-By-Version Update Summary

C.1 Changes for BDS C v1.6

Library Changes

Now buffered I/O is K&R compatible, basically. No doubt there will be more tweaking necessary.

The following functions have been **changed** for v1.6. That is, they have the same name as functions in previous releases, but their operation and/or parameter specifications have changed. Be **very careful** with any programs that use these functions!

FILE *fopen(filename,mode)
char *filename, *mode; Old format: fopen(filename, buffer)
Note: fcreat is GONE.
Returns NULL on error, **not** –1 !!
modes: "r", "w", "a", "rb", "wb", "ab" (b means binary; default is text mode)

int fgets(buf, maxlenth, fp)
char *buf;
int maxlength;
FILE *fp; Old format: fgets(buf, fp)
Note: not changing from the old format can be HAZARDOUS!

int getchar() Now works in either single-char or line-buffered mode (see cmode() below). Default is single-char mode for compatibility with previous versions.
Must **NOT** be used if the compiled program is going into ROM!!

int getc(fp)
FILE *fp; Now differentiates between text mode and binary mode. In text mode, CR's are ignored on input; all text files are presumed to have LF's following CR's.
See also *fgetc* below.

int putc(c, fp)
char c;

FILE *fp; Now differentiates between text mode and binary mode. In text mode, CR's are ignored (nothing written), and LF's ('\n') automatically write both CR and LF to the output file.
See also *fputc* below.

The following functions are new for v1.6:

int cmode(mode)
int mode; Sets getchar() character mode as follows:

```
mode 0: line buffered chars
mode 1: single chars (default)
```

Calling with mode 0 clears the internal buffer of any unsampled characters from previously active line buffered input.

int fgetc(fp)
Same as getc() (renamed for compatibility) int fputc(c, fp)
Same as putc() (renamed for compatibility)

int fread(buf, size, count, fp)
char *buf;
unsigned size, count;
FILE *fp; Read (size * count) bytes from buffered input file.

int fwrite(buf, size, count, fp)
Write size*count bytes to buffered output file.

VOID clearerr(fp) Clear errors in buffered I/O stream.

int feof(fp) TRUE if eof encountered on buffered I/O stream.

int ferror(fp) TRUE if error occurred on buffered I/O stream.

int hseek(fd, hoffset, offset, origin)
BYTE hoffset;
unsigned offset;
Allows 24-bit random record number for MP/M, TURBO-DOS, etc; hoffset is high-order byte of 24-bit random record number. Otherwise, like seek().
Note: Do not use origin value of 2 (relative to EOF) if file has been WRITTEN to since being opened.

int htell(fd) Returns high-order byte of random record number associated with file. Use tell(fd) to get low-order word.

initptr(str_tab, str1, str2, … , NULL)
char *str_tab[], str1, str2, …
Initialize string table

int memcmp(ptr1, ptr2, length)
char *ptr1, *ptr2;
unsigned length; Returns TRUE if the two sections of memory match perfectly. FAST.

putdec(n) Prints decimal value on console. Saves space if printf isn't needed for anything else.

The Standard header file is now STDIO.H, which should always be included in all programs, period. The storage allocation data (for alloc/free) is always declared, because the new I/O library uses it to obtain buffer storage for standard file I/O.

IMPORTANT: Since the new buffered I/O uses alloc/free to obtain storage, it is imperative that all applications obtain memory by using either sbrk or alloc/free. Do NOT use the endext() function to obtain a scratch workspace address! The best way to get the largest possible chuck of memory is to call alloc() with decreasing size parameters until it doesn't return NULL. For an example on how to do this, see what I've done with L2.C.

FOPEN takes the standard parameters "r", "w" and "a". Text mode is assumed by default, so ^Z's are understood and written appropriately. For binary mode, "rb", "wb" and "ab" must be used. The value returned by FOPEN upon error is now NULL, not ERROR! *** Watch out for this one! ***

PRINTF/SCANF have been beefed up. SCANF is supposed to do everything right, including partial lines. Mods to printf/scanf were contributed by Dan Grayson, Urbana, Ill., (217) 367-6384.

STDLIB1.C, STDLIB2.C and STDLIB3.C contain new C-coded library sources. These three files now comprise DEFF.CRL. STDLIB1.C contains buffered I/O; STDLIB2.C contains the printf/scanf families and some other assorted disk I/O stuff; STDLIB3.C contains the piddly remaining stuff.

Run-Time Package

The C.CCC run-time package has been generalized so that M80/L80 may be also used to reassemble the source file (CCC.ASM). See comments in CCC.ASM for instructions on how to use M80/L80 instead of ASM or MAC.

Restart vector optimization hacks incorporated. See below.

Compiler

Error detection and diagnosis beefed up. Standard error reporting format is now

```
Filename: line_no: Error message
```

I.e., no more "include @xxxx: yyyy: Error message".

The compiler can be configured to write errors to a disk file recognized by the companion RED screen editor. RED is now included in source form with BDS C. If RED is invoked while a BDS C error file is present in the current working directory, RED will automatically call up the appropriate source files as named in the error file, and allow editing of the source file at the point at which the errors occurred.

A new code optimization scheme has been introduced. If the target system has any interrupt vectors available for use by the object program at run time, then any such interrupt vectors may be "given" to the object program and thus make it shorter. This is accomplished by generating a customized run-time package module with the appropriate initializations for each interrupt vector that is to be used, then using the new compiler option "-z" followed by the numbers of each interrupt vector to be used (e.g., "-z12345" to use rst1 through rst5). If the compiled code is then linked with the special version of the run-time package, substantial code reduction is achieved once overall program size passes the break-even point set by the additional run-time package initialization overhead. This might typically be at about the 3-4K point. Note that the run-time package is set up to allow the easy selection of up to seven restart vectors (rst1-rst7) by toggling the associated EQU statements. Since rst6 is often used by CDB, and rst7 is often used by DDT/SID, it is not recommended that these vectors be used unless the code reduction is needed very badly. OF COURSE, IF A TARGET SYSTEM USES INTERRUPT-DRIVEN I/O OF ANY KIND, THOSE INTERRUPT VECTORS NEEDED FOR I/O MUST NOT BE USED BY AN OBJECT PROGRAM. BE CAREFUL!

Utilities

The RED Screen Editor has been added to the package. Provided in source form, this editor interfaces with the compiler for convenient program syntax correction. CC can be configured (or told on the command line) to write out an error file (PROGERRS.$$$) containing a record of all syntax errors found in the recent compilation. When RED is invoked, it looks immediately for this special error file. If found, then RED loads up the C source file in which the errors were found, along with the error file itself, and the user may walk through the errors by single keystroke commands, making corrections on the fly. Many thanks to Edward K. Ream for making RED available for inclusion with BDS C, and for enhancing the editor to interface so nicely with the compiler.

A BDS C Configuration program, CCONFIG.C, has been included to walk the user through the various compiler/linker configuration options. In previous releases, these options could only be changed by explicitly altering bytes of data within the CC.COM and CLINK.COM command files, using DDT or SID. The new CCONFIG program also tries to be as explanatory as possible about the various configuration options. Note that there are several new configuration options for v1.6, including RED error-file output control, CDB restart vector selection and other useful customizing features.

A nice BCD floating point package, written by Robert Ward, replaces "simple 4-banger" floating point library as a standard part of the BDS C package. This new package is evolved from "Money Math".

L2 has been updated to work with the new buffered file I/O. Since BDS C's buffered file I/O is only used for writing out the .SYM file, I made that entire mechanism conditionally compiled. The #define SYMFILE definition may be made FALSE to create a shorter version of L2.COM.

CASM has been improved to handle n-level nested includes, and it has been updated to work with the new buffered I/O. Note: The buggy NCASM.C is no longer. I just couldn't figure out what Kevin Kenny was trying to do with his conditional assembly processing, so I gutted all that out of the program. I'd still like to find the guy, though, so if ANYBODY KNOWS HOW I CAN REACH KEVIN KENNY, PLEASE LET ME KNOW!!!!!!!

The CDB debugger package now includes a configuration utility, CDBCONFG, to aid in customizing the CDB utility for individual systems.

The CP.C file copying utility, supplied as a sample source program, has been given a new "verify" option. CP now also allows wild-card user area prefixes, due to enhancements in the WILDEXP.C utility. See CP.C for detailed usage.

The WILDEXP.C wild-card expansion utility has been expanded to allow disk drive and user area specifiers on wild card designations. A wild-card user area specifiers searches through all user areas between 0 and 15 (you can make it search 0-31 by modifying the source), but this takes a little while to do.

A new sample program called DI.C has been provided. This is a simple file comparator utility for quick verification of the equality of (or minor disparity between) two versions of a file.

C.2 Changes for BDS C v1.5

This appendix describes the most significant changes made to each major update of BDS C, in reverse order of release. First the features new to v1.5 are described, then the features added to v1.4. Unless you have just updated from a pre-1.50 version of the package, you probably don't need to read this appendix.

> NOTE: Versions 1.5 and later of the BDS C Compiler require version 2.x (or higher) of the CP/M operating system. In order to take full advantage of CP/M 2.x I/O mechanisms without introducing really painful configuration complications, compatibility with CP/M 1.4 (or earlier versions) has been sacrificed. Users who cannot upgrade their CP/M's to version 2.x must go on using v1.46 of the compiler.

New Command Line Options:

- CC (formerly named CC1) now takes the option **–k**, to activate the Kirkland debugger mechanism. This makes CC write out a special symbol table file for later use by David Kirkland's C debugger package, and causes the compiler to generate special code sequences to allow the debugger to monitor program execution and handle breakpoints at arbitrary points in the code. The debugger package is not included on the standard distribution disk, but is available for nominal cost-of-media from the BDS C User's Group .

- If CC is given a filename without an extension, and the file as named does not exist, CC now will try adding ".C" to the filename and opening it that way.

- CLINK now takes a new option **–n**, which causes the resulting COM file to **not** perform a warm-boot after it is finished executing. This option has the same effect as v1.46's NOBOOT.C program (which is no longer needed when using CLINK, but is provided for use with the optional L2 linker available from the BDS C User's Group). Note that when **–n** is used, there is approximately 2K less user memory available during object code execution because the CCP is not overwritten.

- Another new CLINK option, **–z**, inhibits the clearing of all external data to zero during run-time initialization. If **–z** is not used, then all external data in programs linked under v1.50 is automatically zeroed before control is passed to the "main" function at run-time.

New Library File Searching Capabilities:

Both the compiler and linker (CC and CLINK) now have the ability to search for library files in a default CP/M drive and user area, sometimes in addition to the currently-logged drive and user area. If the user configures CC and CLINK as described in the configuration section below, then CC will know to search a default directory for included files named in angle brackets, and CLINK will know to search a default directory for the run-time package module and library object files. Also, if a CRL file is named on the CLINK command line and CLINK cannot find that file in the current drive and user area, then the default area (as configured) will be searched for that file.

Other New CC Features:

The filename given as argument to the **#include** preprocessor directive may contain an optional user-area prefix in addition to the optional logical disk-drive designator. The format for the filename is the same as the format of C library function filename parameters, as described below in the "Low-Level File I/O" subsection.

Other New CLINK Features:

- CLINK now accepts user area prefixes on CRL filenames given on the command line (except for the main CRL file, which must be in the current user area.) If an **explicit** disk drive and/or user area specification is given on the CRL filename to CLINK, then the default drive and user area (as configured by the user) will **not** be searched automatically. Application: if an explicit user area is given for a new test version of a CRL file, and a similarly named CRL file exists in the default library area, then the version in the default area will **not** be used if the explicitly named one cannot be found.

- CLINK now automatically loads **all** functions, by default, from each CRL file named on the command line in a linkage. The **–f** option is now reversed in sense from previous versions; i.e., when **–f** appears on a CLINK command line, then all subsequently named CRL files are **scanned** (only previously referenced functions are linked) while all CRL files named before the **–f** flag are **loaded** (every function in the file, whether it has been previously referenced or not, is linked). This makes the general format of a CLINK command line be:

```
A>clink <main file> [<other prog files>] [-f <lib files>] <cr>
```

- Other options may be interspersed in the command line, of course.

- CLINK will now automatically print out warning messages when the code and external data areas overlap and when the external data area ends above the base of the BDOS on the development system. These conditions usually indicate an error of some kind; nevertheless, the linkage will be completed and the user may decide whether or not to reconfigure the external data area for future compilations/linkages.

New Low-Level File I/O Features:

- All the low-level file I/O now uses the CP/M 2.2x random-record read and write calls. Therefore, files may be up to 8 megabytes in length instead of only up to 256K bytes as with pre-1.50 releases. The explicit random-record file I/O functions supplied in previous versions (*rread*, *rwrite*, *rseek*, *rtell*, *rsrec* and *rcfsiz*) are no longer included, since their functionality has been incorporated into the new versions of the standard library functions *read*, *write*, *seek* and *tell*.

- The "seek" function may be given an origin code of 2, meaning to seek relative to the end of the file. Note that the offset must be negative to make sense in this case, since the origin is at the end of the file and the offset value is **added** to the origin value. For example, the following call seeks to the next-to-last sector in the file:

```
seek(fd, -2, 2);  /* seek to 2nd sector from EOF */
```

- User number prefixes are now accepted wherever a filename argument is called for. Such a prefix consists of a decimal number between 0 and 31, followed immediately by a slash (/) character and then the filename (with or without an optional disk designator). This causes the file I/O mechanism to switch into the user area associated with each file for the duration of any I/O operation involving that file, then switch back to the current user area when done. Any filename may now take either an explicit disk designator, an explicit user area, or both. If both are given, then the user area specification ***must precede*** the disk designator. Here is an example:

```
if (open("0/A:DATABASE.DAT",2) == ERROR)
        exit(puts("Can't open the database, turkey."));
```

- Note that this allows programs in separate user areas to access a common data file kept on one particular drive and user area, instead of having a separate copy of the data file for each user area that requires it. If you are running the "ZCPR" public-domain CCP replacement program for CP/M, or any shell (such as "MicroShell") that searches special drives and user areas for command files, then that feature combined with the user-area enhancements to the file I/O library allow a very efficient utilization of the CP/M filesystem.

- There are some new functions that provide better diagnosis of errors caused by low-level file I/O calls. Whenever a call such as *open*, *read* or *write* returns a value of –1 (ERROR), the *errno* function may be called to return a more detailed error description

code explaining exactly what went wrong. The *errmsg* function may be used to return a pointer to a string corresponding to the error value returned by *errno*. A typical usage of these functions is as follows:

```
i = read(fd, buffer, 20);       /* try to read 20 sectors */
if (i == ERROR)                 /* if an error occurred...*/
        printf("Read error: %s ", errmsg(errno()));
...
```

Miscellaneous New Features:

- The entire external data area is now cleared to zero by the run-time initializor before control is transferred to the **main** function for program execution. This means that programs which use the storage allocator need no longer explicitly clear the _allocp variable before using the allocator.

- The external data declarations for the storage allocation functions *alloc* and *free* have been permanently enabled, so that it is no longer necessary to go into the STDIO.H header file and bother with commenting/uncommenting the variable declarations in order to get *alloc* and *free* to work.

Incompatibilities With Earlier Versions:

1. When the **#include** preprocessor directive is given a filename enclosed in angle brackets (**#include** <filename>), then the default drive and user area (as described in the configuration section below) is presumed to contain the named file. A filename enclosed in double quotes (**#include "**filename**"**) is presumed to reside on the currently-logged drive and user area, as in previous versions, unless the filename contains an explicit user area and/or disk designator.

2. BDS C v1.5 may only be used with version 2.0 or later of the CP/M operating system; CP/M 1.4 is no longer supported.

3. The run-time package has been modified, causing incompatibility with CRL files generated by previous versions of the compiler. In order to be used with version 1.5 components, a CRL file must have been generated by version 1.5 of the compiler. Old CRL files should be discarded.

4. CLINK now loads all functions from all named CRL files by default, regardless of whether or not they have been referenced by previously loaded functions in a linkage. The CLINK option **–f** now operates identically to the L2 linker's **–l** option.

5. The hardware related defined constants from previous versions of the STDIO.H header file have been removed from that file and placed into a new header file named HARDWARE.H, so that system-dependent parameters are kept separate from general ones. The console and modem port definition sections have been changed into a more general form to allow for both status-driven and memory-mapped I/O ports.

6. The *getline* function no longer includes a trailing newline character as part of the collected line of input text. Like *gets*, lines input through *getline* are terminated by only a single NULL character.

C.3 Changes For BDS C v1.4

There has been a hefty amount of revision, expansion and clean-up applied to the package since v1.3. A good portion of the changes were made in response to user feedback, while others (mainly internal code generation optimizations) resulted from the author's dissatisfaction with the early version's performance.

Library Sources Included:

The assembly language sources for the BDS C run-time package (CCC.ASM —> C.CCC) and all non-C-coded library functions (DEFF2?.CSM —> DEFF2.CRL) are now included with the package, so that they may be customized by the user for non-CP/M environments. The new compiler and linker each accept an expanded command line option repertoire allowing both the code origin and r/w memory data area to be specified explicitly, so that generated code can be placed into ROM. The run-time package may be configured for non-CP/M environments by customizing a simple series of EQU statements, and new special-purpose assembly language library functions may be easily generated with the help of the CASM assembly-language preprocessor program included with BDS C as standard equipment.

Better Buffered I/O:

The buffered I/O library can now be easily customized to use any number of sectors for internal disk buffering. A general purpose standard header file, STDIO.H, controls the buffering mechanism and also provides a standard nomenclature for some of the constant values most commonly used in C programs. All users should carefully examine STDIO.H, become intimate with its contents, and use the symbols defined there in place of many of the numeric constants previously abundant in early sample programs. For example, the symbol **ERROR** is more illuminating than when it is written as **–1**.

Directed I/O and Pipes:

For Unix enthusiasts, an auxiliary function package (written in C) named "DIO.C" has been included to permit I/O redirection and pipes a la Unix. If you do not need this capability, then it isn't there to take up space; if you **do** need it, then you simply add a few special statements to your program and specify DIO to CLINK at linkage time, then use a subset of the standard Unix redirection syntax on the CP/M command line.

One Stack is Better Than Two:

A single run-time stack configuration has replaced the two-stack horror used in the earliest releases. Function parameters are now passed **on the stack**, and local storage allocation also takes place on the stack. This leaves all of memory between the end of the externals (which still sit right on top of the program code) and the stack (in high memory) free for generalized storage

allocation; several new library functions (*alloc, free, rsvstk, and sbrk*) have been provided for that purpose.

Better Code Quality:

Last but not least, the code generator has been taught some optimization tricks. The length of generated code has shrunk by 25% (on average) and execution time has been cut by about 20% over version 1.32. Part of this cut in code bulk is due to the new CC option **–e**, which allows an absolute address for the external data area to be specified at compile time. This enables the compiler to generate absolute load and store instructions (using the **lhld** and **shld** 8080/Z80 ops) for external variables.

Incompatibilities With Pre-v1.4 Versions:

Because the run-time package has been totally reorganized for release v1.4, CRL files produced by earlier versions of the compiler will **not** run when linked in with modules produced by the new package. Therefore all programs should be recompiled with the current version, and old CRL files should be thrown away. There are also a few source incompatibilities that require a bit of massaging to be done to old source files. These are:

1. The statement

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

2. must be inserted into all programs that use buffered file I/O, and *should* be inserted into all other programs so that the symbolic constants defined in STDIO.H can be used.

3. Comments now **nest**; i.e., for each and every "begin comment" sequence (**/***) there must appear a matching "close comment" sequence (***/**) before the comment will be considered terminated by the compiler. This means that you can no longer comment out a line of code that already contains a comment by inserting **/*** at the start of the line; instead, a good practice would be to insert **/*** above the line to be commented out, and to insert ***/** following the line. Although complete comment nesting is something that UNIX C doesn't support, I feel it is important to have the ability to comment out large sections of code by simply inserting comment delimiters above and below the section. Otherwise, any comments *within* such a block of code have to be removed first.

For v1.4, the run-time package comes configured to support up to eight open files at any one time, but previous versions have accepted up to sixteen. To allow more than eight open files, the "NFCBS EQU 8" statement in the run-time package source (CCC.ASM) must be appropriately changed and the file re-assembled. See Chapter 2 for details on customizing the run-time package.

Appendix D

Error Messages Explained

D.1 CC Error Messages

For the duration of this document, the term *directory* will be used to denote some arbitrary CP/M logical drive **and** user area combination.

File I/O Errors

Close error

Disk drive door open? If not, you've got some strange kind of hardware problem.

Error on file output…disk full?

If not, check the hardware.

Can't find CC2.COM; writing CCI file to disk

There are two directories where CC searches to try and find CC2.COM. One of them is always the current directory, and the other depends on whether or not the **–a** option is used with CC. If so, then the directory specified in the option is searched; otherwise, the *default* directory (as defined in the configuration section of Chapter 1) is searched. This message is printed if CC2.COM cannot be found in the two directories searched.

Disk read error

Time to format some new floppies?

Cannot open: <*filename*>

The specified file cannot be found. If the user has configured CC to search a specific directory for **#include** files enclosed in angle brackets, then a user number, slash, and disk designator will be printed preceding the filename in this error message. If CC has not been configured, then only a disk designator will appear. Since a user number prefix is not allowed on the CC command line, the top level source file must always be in the current user area when CC is invoked, although it may be on a different logical drive.

Overflow Conditions

Sorry; out of memory The source file is too big to fit into memory. Either get more memory, in case that is possible, or break the source file into smaller pieces.

Out of symbol table space; specify more…
Use the **–r** option to reserve symbol table space for CC. Or, break the source file into smaller pieces.

Too many functions (63 max)
A single BDS C source file may only contain up to 63 function definitions. Programs having more than this many functions must be split into separate source files.

String too long (or missing quote)
Usually, this error is caused by missing double-quotes around character strings. If a string looks properly delimited, check to make sure you haven't tried to include a double quote character within the string without escaping the double quote (preceding it with a backslash).

Too many cases (200 max per switch)
Self-explanatory.

#include files nested too deep
This can happen if you try to have recursive includes.

String overflow; call BDS
This is a preprocessor string table overflow caused by having too many very long identifier names in **#define** directives. It should only happen for VERY big programs. A special version of the compiler with larger string space allocations may be obtained by sending a SASD (self-addressed stamped disk) to BD Software along with some kind of proof-of-purchase of the BDS C package.

Preprocessor Errors

Warning: Ignoring unknown preprocessor directive
If an unsupported preprocessor directive is encountered, this warning is printed. Currently, this is the *only* non-fatal diagnostic message.

EOF found when expecting #endif
Conditional compilation improperly delimited.

Not in a conditional block
This appears when something like **#endif** is encountered, when there was no previous **#if**, **#ifdef** or **#ifndef**.

Conditional expr bad or beyond implemented subset
CC only allows a subset of operators to be used in the **#if** preprocessor directive. See chapter 4 for a summary of the **#if** expression syntax.

Bad parameter list element
Bad identifier present in the formal parameter list of a function definition.

Missing parameter list The identifier from a parameterized **#define** appears without its parameters.

Parameter mismatch The identifier from a parameterized **#define** is used with a different number of parameters than in its definition.

Missing legal identifier An identifier is expected in an expression but none appears.

Syntax Errors

Unclosed comment in: <filename>
Usually an accurate diagnostic. If you get this message and have no clue to where the unclosed comment begins, try giving the **–p** option to CC and check the text immediately preceding the point where the code just seems to "cut off" at the end of the printout. That's probably the location of an unclosed comment, since all the subsequent text (that disappeared) would have been considered part of the comment and stripped from the source file before the printout.

Encountered EOF unexpectedly (check curly-brace balance)
Check for unclosed comments, and unclosed curly-braces. The User's Group program LCHECK.C may be used to check curly-brace nesting levels.

Unmatched right brace Either a left brace is missing, or there is an extraneous right brace.

Illegal external statement
This is usually caused by too many right braces in a function, causing the compiler to detect the end of a function definition prematurely.

Function definition not external
This happens when something that looks like a function definition is encountered within another active function definition. Probably it is just a missing semicolon after a function call, or a similar typo.

Missing semicolon
You can usually believe this message; keep in mind, though, that the line number given here always points to the **beginning** of the statement that the compiler thinks is unterminated. In a multi-level nested control structure (such as **if**…**else** or **if**…**if**), the missing semicolon might be several physical lines lower in the code than claimed by the line number appearing in the error report.

Expecting (
Typically encountered after the **while**, **if** or **switch** keywords.

Unmatched left parenthesis
This is another type of error that is usually detected, but might generate other less useful messages in certain cases.

I'm totally confused. Check your control structure!
This might be caused by extraneous characters or very erroneous curly-brace nesting.

Illegal { encountered externally
Possibly caused by mismatched curly braces.

Mismatched control structure
Another variation on the unequal curly brace nesting theme.

Expecting while
Is a **do…while** statement missing its **while**?

Illegal break or continue
break statements are only allowed inside loops and **switch** constructs. **continue** statements are only allowed inside loops.

Bad for syntax
Self-explanatory; check for the correct number (2) of semicolons and their placement.

Expecting { in switch statement
The expression portion of a **switch** statement must be followed by compound statement in curly-braces.

Bad case constant Each **case** constant must be either an absolute constant or a simple constant expression (symbolic constants are acceptable, of course).

Illegal statement This error is drawn when, for example, a **case** or **default** statement is found outside of a **switch** construct.

Syntax error
It takes something totally unintelligible to draw this error, such as a missing left double quote before a character string. An extraneous character in the file may also do it.

Bad constant Some expressions must be constant expressions, such as **switch** expressions and the values used for **case** constants.

Bad octal digit
If a numeric octal constant beginning with a zero contains the digits 8 or 9, this error is drawn.

Bad decimal digit This happens when a decimal constant contains bad characters, or else the user forgot to precede a hex constant with the sequence *0x*.

Curly-braces mismatched somewhere in this definition

This is a rather useful feature of the compiler: if the source text has too many left curly-braces, this error will point to the beginning of the function or data definition in which the first detected mismatch occurred.

Declaration Errors

Undeclared identifier: <name>

This might be a real identifier that just wasn't declared, or a misspelling of an identifier.

Bad declaration syntax Usually the compiler thinks it's processing a data declaration as soon as it sees a type designator (such as **char** or **int**). This error is drawn if the rest of the statement containing that keyword does not resemble a declaration.

Need explicit dimension size

An omitted dimension size in array declarations is only permitted when the array is a formal parameter to a function. If such an array is two dimensional, then only the first dimension may be omitted.

Too many dimensions BDS C allows only up to two dimensions per array variable.

Bad dimension value Dimensions in array declarations must be given as constants or constant expressions.

Redeclaration of: <name>

Aside from actually writing multiple conflicting declarations for a single variable, another way to draw this error is to declare a formal parameter of a function **inside the body of the function** instead of immediately before the body. Note that formal parameters are automatically given type **int** if not declared before the body of the function; therefore, a subsequent declaration of the formal parameter identifier as a local variable in the body of the function constitutes a redeclaration.

Expecting { in struct or union def

Self-explanatory.

Illegal structure or union id

This error is drawn when the identifier appearing in the structure tag position of a structure declaration was previously declared as something other than a structure tag.

Attribute mismatch from previous declaration

The elements in a structure declaration may be reused within other structures providing their major attributes (type and offset) are identical within each structure type. This error appears when a structure element name is re-used with different attributes.

Declaration too complex

This error is caused by too many levels of indirection, or too many parentheses for the compiler to handle.

Missing from formal parameter list: <name>

This happens when a declaration of a formal parameter appears before a function body, but no such parameter is present in the parameter list following the function name.

Bad parameter list syntax

Something other than a comma-delimited list of identifiers in the parameter list of a function definition draws this error.

Miscellaneous errors

<text>: option error

If CC detects some badly formed command line option, it will print the text it couldn't understand along with this message. Check the command line option descriptions in Chapter 1 to make sure you're giving the correct forms.

Compilation aborted by control-C

If the user types control-C on the console during a compilation, then this message gets printed and control is returned to command level. Note that console polling may be disabled by special configuration of CC.COM as described in the configuration section of Chapter 1. This may be required for certain interrupt-driven systems to allow type-ahead during compiler execution.

Can't have more than one default:

This is printed if more than one **default:** clause exists within a single **switch** construct.

Illegal colon

Colons (other than in literal strings) are only allowed as part of the ternary operator, or following a label, **case** or **default**.

Undefined label used

Label references (allowed only in **goto** statements) must refer to a label local to the current function definition.

Duplicate label

A particular identifier may only be used for one label per function.

D.2 CC2 Error Messages

Note: some of the file I/O errors printed by CC2 are the same or very similar to the messages listed above for CC, so they will not be repeated in this section.

File I/O, Syntax, Overflow and Other Miscellaneous Errors

Can't create CRL file No more directory slots on the output drive?

CRL Dir overflow: break up source file

There are only 512 bytes of directory space allocated for each CRL file. It is possible to overflow the directory space for a single source file by having too many functions defined that contain 8 or more characters in their names (only the first 8 characters of each name are actually stored in the directory.) Either shorten your function names, or reduce the number of functions per source file.

Internal error: garbage in file or bug in C

If this happens during CC2, it is probably a compiler bug. Please contact BD Software for assistance.

Illegal statement Something totally weird was encountered.

Missing { in function def.

Usually, whatever draws this error is not really the start of a function definition, but for some reason the compiler thinks that the previous (or current) function has been terminated and another is beginning. Check for too many right curly-braces in the program.

Missing semicolon Missing semicolons after expression statements will usually be detected and diagnosed correctly.

Sorry, out of memory. Break it up!

The file is too large. Usually, if a file gets through CC then it will also make it through CC2, although there are exceptions.

The function <foo> is too complex; break it up a bit

There are certain internal tables that cannot handle too big a function. Rather than require the user to set a bunch of confusing parameters telling the compiler how much space to reserve for various tables and lists, I decided to set most table sizes constant and allow for fairly hefty functions…but only up to a point. Properly structured C programs shouldn't draw this message.

Sub-expression too deeply nested

The most common cause of this error is a multiple assignment statement that goes on forever. The solution is simply to break the line up into smaller chunks.

Compilation aborted by control-C

Unless the appropriate CC configuration byte is customized to zero by the user (see the *Configuration* section in Chapter 1), typing control-C on the system console device will terminate a compilation, print this message and immediately return to command level.

Errors in Expressions

Lvalue required An object is required that can have its address taken, or that must be legal on the left of an assignment operator.

Lvalue needed with ++ or – – operator
Only simple variables can be auto-incremented or auto-decremented.

Bad left operand in assignment expression
If the expression on the left of an assignment operator cannot have a value assigned to it, this error is drawn. For example, a character **array** is not an lvalue, although it may be subscripted to produce a legal lvalue.

Mismatched parenthesis An expression following a left parenthesis is terminated by a matching right parenthesis.

Mismatched square brackets
A subscript following a left square bracket is not immediately followed by a matching right square bracket.

Bad expression This is the general "I give up" message printed when an expression (or what is supposed to be an expression) does not make any sense to the compiler. That does not necessarily mean that the error is obvious, but usually it is.

Bad function name
This is printed when the compiler sees an identifier followed immediately by a left parenthesis, and the identifier has been previously declared as something **other** than a function name.

Bad arg to unary operator
The operand of a unary operator is not of appropriate type for that operator.

Expecting : Did you intend to write a **?:** expression and forget to include the colon?

Bad subscript Is an array subscript of the proper type for a pointer arithmetic operation? For example, a subscript in an array expression cannot be a pointer.

Bad array base You are attempting to subscript something that cannot be subscripted. One possibility: are you attempting to subscript the **argv** formal parameter in your *main* function without having declared **argv** correctly?

Bad structure or union specification
The expression to the left of the **.** (period) operator is not a legal structure or union base.

Bad type in binary operation
Certain types of variables cannot appear together in a binary operation; for example, you cannot add two pointers (although you may subtract them, yielding a result scaled by the size of the objects begin pointed to), or perform most bit-wise and obscure operations on non-simple-variable objects.

Bad structure or union member
The expression to the right of a **.** (period) or **—>** operator is not a valid structure or union element.

Bad use of member name
The identifiers declared as members of a structure or union cannot be used outside of a structure or union operation.

Illegal indirection Atattempt is being made to operate on some object as if it were a pointer, when the object is *not* a pointer.

Encountered EOF unexpectedly
This is either a bad syntax error or a sign of file damage. Badly matched curly-braces might also be responsible, although the present version of the compiler will usually be more specific about those kinds of errors.

Bad argument list Somethingillegal was found in the parameter list for a function call, such as a semicolon or other keyword not legal in an expression.

Missing or misplaced (An expression in parentheses was expected, such as following the **while** keyword, and no left parenthesis was found.

Missing or misplaced) An expression which began with a left parenthesis was not followed by a closing right parenthesis. This might be due to an extraneous character in the middle of the expression.

D.3 CLINK Error Messages

Note: many of the possible file I/O errors printed by CLINK are self-explanatory; only the ones requiring some comment are shown here.

No user area prefix allowed on main filename
User area prefixes are allowed on all filenames **except the first** on the CLINK command line.

Dir full No more directory space in which to create a new output file.

Error writing: <filename>
Probably out of data space on the disk.

Can't close: <filename> Hardware error?

No main function in <filename>
The first CRL file named on the CLINK command line must contain the **main** function for the program you are linking. Note that the L2 linker (available from the User's Group) does not have this restriction.

Missing function(s): <list-of-names>
The named functions were not found in the files listed on the command line or in the standard library files. If you used the **–f** option to cause files to be scanned instead of loaded, it's possible some of the named functions were present but not loaded because no previous functions had referenced them. In this case, simply re-scan the files containing the missing functions.

Warning! Externals extend into the BDOS!
This is printed when the ending address of the external data area is greater than the base of the BDOS on the system being used for compilation. If the code is to be run in another environment where there won't be any conflict, this message may be ignored. But don't try to run the program on the system where linkage drew this message…

Warning! Externals overlap code!
This is printed when the starting address of the external data area is less or equal to the last code address of the program. Usually it means the externals were placed too low with the **–e** option. If you are creating code for a customized environment where the code resides above the externals, just ignore the message.

Out of memory Notenough memory to perform the linkage. Try using the L2 linker, which can link programs up to about 8K larger than CLINK can.

Bad symbols A symbol file being read in via use of the **–y** option contains badly formatted entries.

Ref table overflow The forward-reference table ran out of space. Use the **–r** option to reserve more space. Usage is "**-r** *xxxx*", where *xxxx* is given in hexadecimal. 600 is the default; try 800 or A00, etc., until the error goes away.

SYM file symbol already defined: <symbol>
A symbol being read in via use of the **–y** option is identical to a function already loaded and defined. The original value is kept, since that function has already been loaded and/or defined, and the new one is thrown away.

Ignoring duplicate function: <name>
A function in a CRL file being loaded has a name identical to a function already loaded from a previous file. The original is kept, and the new version is ignored.

Sorry; 255 funcs maximum

CLINK can only handle up to 255 functions in a single linkage. If you need to link a larger number of functions, obtain the L2 linker from the BDS C User's Group.

Index

&& operator 86
|| operator 86
_allocp 196
.CCI file 14
.CCI files 16, 17
#define 85, 87
#if 85
#include 9, 13, 85, 88, 196, 198

A

aborting compilation 14, 17
aborting linkage 18
abs 46
alloc 51, 84, 196
append to string 60
argc 26, 146
argc & argv 13
arghak 36
argv 26
ASM assembler 151
atoi 62
auto-loading CC2 14
auto-loading of CC2 17

B

BCD Function Package 161
BCD package 161
BD Software's address 1, 27
bdos 44, 146
BDS C User's Group 193
BDS.LIB 37, 184
BDSCIO.H 161
begin 77
bios 44
biosh 44
block memory assignment 48
block memory comparison 48
block move 48
blocks 77, 83
Buffered I/O 197
buffered I/O 63, 69

C

C.CCC 15, 183, 197
C Reference Manual 75
C User's Group 3
call 46
calla 46
case conversion 60
CASM 29
CASM.SUB batch file 152
CASM Utility 151
CC 7, 13, 193
CC2 7, 17
CC.COM 186
CCC.ASM 11, 36, 184, 197
CCI files 7
CCONFIG.COM 8
CCP 10
cdb debugger 193
CDB restart vector control 11
cfsize 67
chaining 49
chaining with parameter passing 50
character processing 59
clearerr 72
CLIB 22
CLIB commands 22
CLINK 18, 43, 193, 194
CLINK.COM 186
CLINK debug mode 18
CLINK interactive mode 18
CLOAD utility 152
close 65
closing files 72
cmode 55
CMODEM 157
codend 51
command line parameters 26
Comment nesting 14
comment nesting 76
comments 198
concatenation 60
conditional compilation 85, 88
configuration 7
Console polling 9
console switch register 44

constant expressions 86, 145
control-Z 55, 70
CP/M 4, 196
creat 64
creating CRL files 24
CRL directory 22, 29
CRL Files 22
CRL files 151
CRL format 29
CSM files 43, 184
csw 44
curly-brace substitutes 77
customized environments 183

D

DDT 185
debugger 109
debugging 37
declarations 79, 87
Default Library Area 9
default library area 18
DEFF15.CRL 161
DEFF.CRL 12
DEFF.CRL and DEFF2.CRL 43
DEFF files 9, 18, 197
delete file 67
DIO.C 197
Directed I/O 197
disk buffering 12
disk designator 18, 63
division by zero 78

E

end 77
endext 33, 51
EOF 55, 70
errmsg 68, 196
errno 68, 195
error handling 64, 68
error messages 199
error recovery 146
exec 49
exec functions 19
execl 49
execution speed 16, 17
execv 50
exit 44
external data 33
External data area 30
external data area 19, 30, 77, 84
external data boundary locations 51
external data initialization 21
external data starting location 14
external definitions 83
externs 51

F

fabort 67
fcb 68
fcbaddr 68
fclose 72
feof 72
ferror 72
fetch routines 35
fflush 72
fgetc 70
fgets 73
file control block 68
File I/O 195
file I/O 63
Filenames 63
floating point (BCD)function package 161
floating point package 161
fopen 69
for statement 83, 146
formatted output 58
forward reference table 20
fprintf 73
fputc 70
fputs 73
fread 71
free 52, 84, 196
fscanf 73
function entry protocol 33
function modules 30
fwrite 71

G

get file size 67
getc 70
getchar 55
getline 56
gets 57
getw 70

H

HARDWARE.H 12, 157, 196
hardware.h 12
hseek 66
htell 67

I

identifier name restrictions 76
index 62
initb 62, 148
initialization 62

initializers 82
initptr 63
initw 62, 148
inp 45
iobreak 55
isalpha 59
isdigit 59
islower 59
isspace 60
isupper 59

K

kbhit 56
keywords 76

L

L2 110
labels 83
language restrictions 75
library file searching 43
library source organization 36
list of needed functions 31
listing CRL file contents 23
load address 19
loading library files 19
loading library functions 18
loading overlays 51
logical connective operators 79
long integer package 172
longjmp 53
lprintf 58

M

M80 assembler 151
ma1toh 36
machine code subroutines 46
main function 30
max 46
max no. of open files 198
maximum CRL file size 22
memcmp 48
MicroShell 195
min 46
movmem 48
MP/M 26, 65

N

NFCBS run-time package option 12
NOBOOT 194
nrand 47
NSECTS 12

O

oflow 67
open 64
opening files 64, 70
Optimization 15
Optimization control byte 10
order of evaluation 86
outp 45
overflow 64, 67
overlap of code and data 15
overlay loading 51
overlays 20, 21, 177

P

parity bits 10
pause 46
peek 45
pipes 197
pointers to arrays 81
poke 45
port-driven I/O 45
preprocessor directives 87
printf 57
printing source file 15
putc 70
putchar 56
puts 56
putw 71

Q

qsort 49

R

rand 47
random number generation 47, 48
raw I/O 63, 64
re-entrant code 35
read 65
RED editor 89
RED error file control 10
RED error file output 15
ref table overflow 20
register designator 79
relocation parameters 32
rename 67
reserving symbol table space 15
Restart optimization 16
ROM-based applications 184
ROM-ing code 19
ROM preparations 183
royalties 4
RSTNUM 11

RSTNUM run-time package option 11
rsvstk 53
run-time package origin address 15
run-time package RST optimization 12

S

sbrk 52
scanf 58
scanning library files 19
seek 66, 195
setfcb 68
setjmp 53
setmem 48
shift operators 79
sign extension 78
sizeof 79, 86
sleep 46
sorting function 49
source function limits 16
source text 13
sprintf 60
srand 46
srand1 47
sscanf 60
stack 32, 197
stack initialization 20
stack safety margin size 53
stack utilization 32
STDIO.H 196, 197
stdio.h 12, 26
STDIO.H“ 198
STDLIB*.C 186
storage allocation 52, 196, 197
storage classes 75, 77, 79
strcat 60
strcmp 60
strcpy 60
string comparison 60
string copy 61
string length 61
string processing 59
stripping parity 10
strlen 61
structure and union declarations 82
Submit Files 9, 16, 26
swapin 50, 179
switch statement 83
symbol table file 20
system requirements 5

T

TELED 12
tell 66
time delays (sleep) 46
tolower 60
topofmem 33, 51
toupper 60
transferring functions between CRL files 23
type specifiers 79

U

unary operators 86
ungetc 70
ungetch 56
Unix 4
unlink 67
USAREA run-time package option 11
user area prefix 18, 63
User area recognition 11
User Areas 9, 195
user areas 10
USERST 11
USERST run-time package option 11

V

variable scope 77

W

warm boot 32
warm boot inhibition 20
Warm boots 10
write 65

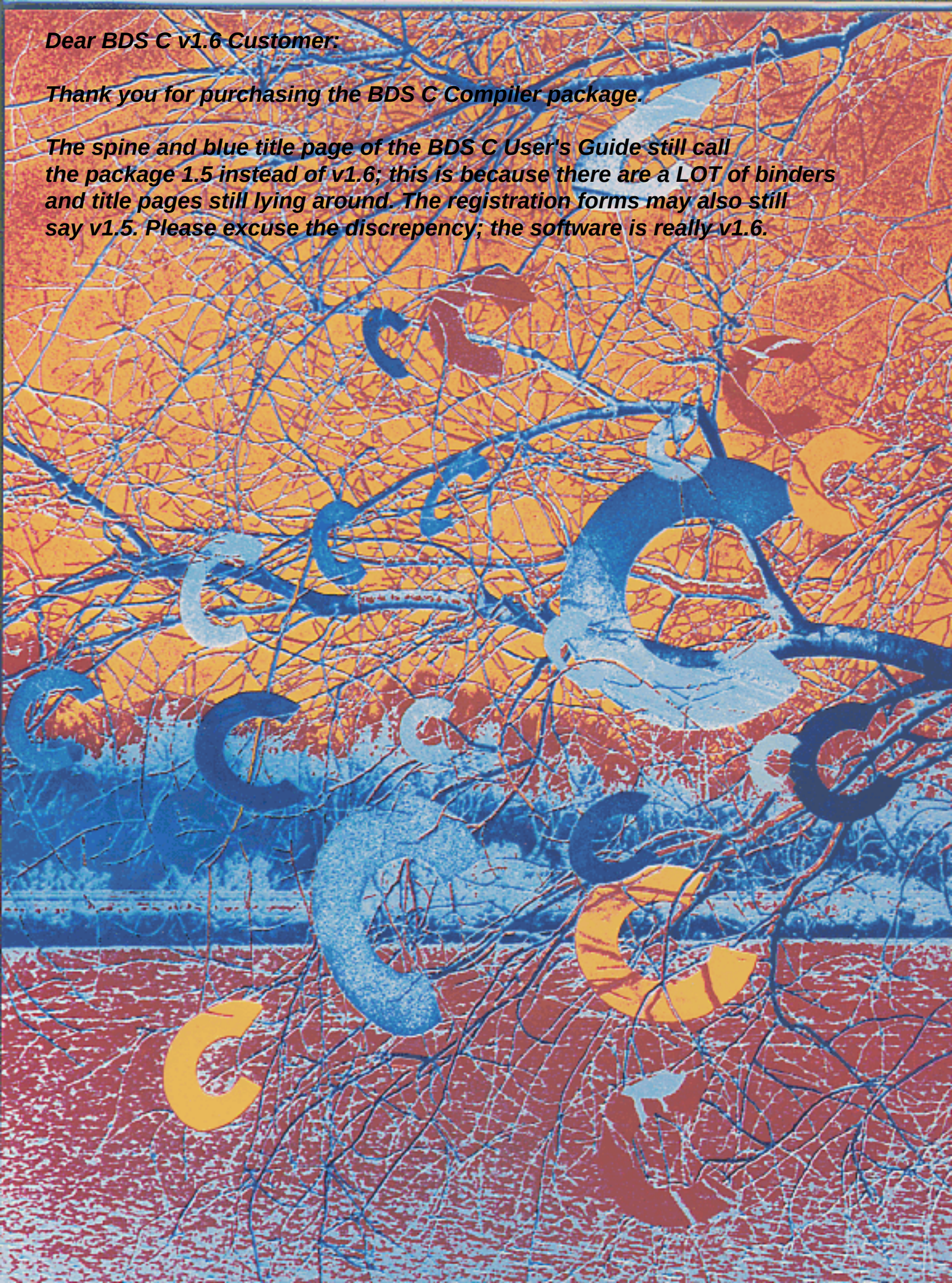
Y

yank symbols 20

Z

ZCASM utility 151
ZCPR 195
ZOPTn run-time package option 12

Dear BDS C v1.6 Customer:

Thank you for purchasing the BDS C Compiler package.

The spine and blue title page of the BDS C User's Guide still call the package 1.5 instead of v1.6; this is because there are a LOT of binders and title pages still lying around. The registration forms may also still say v1.5. Please excuse the discrepency; the software is really v1.6.