Manual

THE BD SOFTWARE C COMPILER v1.5

Lifeboat Associates
1651 Third Avenue
New York, N.Y. 10028
Tel: (212) 860-0300
TWX: 710-581-2524 (LBSOFT NYK)
Telex: 640693 (LBSOFT NYK)

BD Software

C

C Compiler v1.5
User's Guide

Distributed by:
Lifeboat Associates
1651 Third Avenue
New York, N.Y. 10028
Tel.: (212) 860-0300
TWX: 710-581-2524
Telex: 640693

Disclaimer

Copyright Notice

Table of Contents

Chapter 1

Introduction

Leor Zolman
BD Software
P.O. Box 9
Brighton, Massachusetts 02135
(617) 782-0836

1.1 Hello There

Thank you for purchasing BDS C. This software package is one programmer's personal implementation of the minicomputer language **C**, geared down specifically for microcomputers running the CP/M operating system. The primary design goal of BDS C was to allow C programmers to move forward at a steady, **efficient** pace as structured programs are developed. To this end, the compiler and linker were made fast enough to compile, link and execute programs repeatedly without wearing out the programmer's patience.

There is nothing quite so annoying as waiting for a slow language processor to crunch through its task, only to be told (after many minutes) of some trivial errors which then require the entire process to be repeated. With BDS C, such errors usually show up within seconds after starting up the compiler...and programmers can their spend time **programming** instead of muttering obscenities at slow computers.

1.2 Pricing Philosophy

The pricing of software packages for personal microcomputer systems can hardly be termed an "exact science". It is rather difficult to apply simple economics to the problem because software for this market is so easy to duplicate illicitly. Attempting to take all ramifications of such duplication into consideration when establishing pricing policies can only lead to a nervous breakdown. Yet, many vendors have adopted the view that software must be priced outlandishly high both to imply superior quality and to provide an adequate return on a per-copy basis assuming that many illegal copies become made.

Looking at the situation from the opposite viewpoint, I made the decision early on to sell BDS C at a very **low** price, in order to maximize the number of copies actually sold and discourage ripoffs by the only means that seems to have any effect in this market: producing a product that is deemed a good value for the money by most of its users. After three years I still think the price is correct and expect to keep on selling the package cheaply in the future.

1.3 Availability

For the first three years of its lifespan, BDS C was available exclusively from one central distributor. The original reason for the exclusivity was to give the author freedom from distribution problems as the package evolved technically. Unfortunately, true widespread distribution of the package was made difficult in those three years because vendors and distributors could not deal directly with BD Software.

BDS C is no longer being sold on an exclusive basis. The package is available on 8" disk directly from BD Software, or in a wider variety of formats from a growing number of new direct distributors. For example, the BDS C User's Group is now selling the compiler to first-time buyers in addition to performing its original services or providing compiler updates, library volumes and newsletters to registered users. Other dealers will be popping up soon; inquiries from potential dealers and distributors are welcome.

1.4 Forget End-User Royalty Arrangements...

Aside from the standard agreement that the package be used on one system only[1], there are no 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.

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 by this author. 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[2]. This probably wouldn't bother me if I were at least 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[3] operating system[4]. The compiler itself runs on 8080/Z80 microcomputer

---

1. Each user implicitly accepts the standard license in the act of unsealing the diskette envelope, though the license agreement must be signed and returned for proper registration.

2. You know who you are!

3. Unix is a trademark of Bell Laboratories.

4. 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.

systems equipped with the CP/M[5] operating system, and generates code to be run either under CP/M or at any arbitrary location in ROM or RAM (although there must be a read/write memory area available at run time somewhere in the target machine.)

The main objective of this project was 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 a friendly environment in which to develop CP/M utility applications, with an emphasis on an elegant, efficient human interface for both compiler and end-application usage.

Unfortunately, the implementational mechanics of the C language do not conform as well to the 8080's hardware characteristics as they do to the PDP-11's[6]. Operations natural to the 11 (such as indexed-indirect addressing, a crucial necessity when dealing with automatic storage allocation) expand into rather inefficient code sequences on the 8080. Thus BDS C is not likely to become quite as universal a systems programming language to the 8080 as UNIX C is to the PDP-11; but then, as better microprocessors replace present 8 bit machines, you can bet there will be C compilers available that generate code efficient enough to resign application-oriented assembly language programming to the history books. Consider this package a warm-up to that era.

In summary, BDS C's big tradeoff when compared to assembly language programming is a loss of run-time object code efficiency, both spatial and temporal, in favor of greater structure and comprehensibility throughout the development stage. For just about all educational and most systems programming applications, I believe the sacrifices are rather minimal in contrast to the benefits.

1.6 System Requirements

The practical minimum hardware configuration required by BDS C is a 40K **CP/M 2.x** system. Most sample programs included in the package will compile (without segmentation) and run on a 48K system.

---

5. CP/M is a trademark of Digital Research, Inc.

6. PDP is a trademark of Digital Equipment Corporation.

BDS C loads the entire source file into memory at once and performs the compilation in-core, as opposed to passing the source text through a window. This allows a compilation to proceed quite rapidly in contrast to conventional algorithms. The main bottleneck for most modestly-sized compilations is now the disk I/O involved in reading in the source text and writing out the CRL file, even though these operations take place as fast as a standard CP/M system can handle them. A minor restriction under this scheme is that a source file must fit entirely into memory for the compilation; this may sound bad to you at first, but 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, each containing a variable number of functions. Partitioning a program into several files serves to minimize recompilation time following minor changes as well as keep the individual source files from overflowing available memory restrictions.

1.7 New Features of v1.5

WARNING! Version 1.5 of the BDS C Compiler will not work under pre-2.x versions 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.

1.7.1 Functional Changes to Major Commands

1.7.1.1 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.

1.7.1.2 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.

1.7.1.3 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 specifier. 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.

1.7.1.4 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** for previously referenced functions only, while all CRL files named before the **-f** flag are **loaded** in their entirety. This makes the general format of a CLINK command line be:

```
A>clink <main file> [<other files in prog>] [-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.

1.7.2 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 end of file */
  ```

- 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()));
          ...
```

1.7.3 Miscellaneous New Features

The entire external data area is now cleared to zero by the run-time initializer 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 BDSCIO.H header file and mess with commenting/uncommenting the variable declarations in order to get alloc and free to work.

In the documentation department, the User's Guide has been greatly overhauled with use of the FinalWord text processor (from Mark of the Unicorn, rah!) driving a Diablo 630 printer with the "96-Bold-PS" metal daisywheel. There is now a table of contents, an index, and contiguous page numbering throughout the **entire** body of the manual, including the appendices (which were scattered sheets tacked on to the end of the Guide in earlier releases).

1.7.4 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 above) 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 (see section 4).

5. The hardware related defined constants from previous versions of the BDSCIO.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.

1.8 Other New Features: A Summary for Pre-v1.4 Users

There has been a hefty amount of revision, expansion and clean-up applied to the package since the last generation (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 some of his earlier kludgery and short-cut algorithms.

1.8.1 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.

1.8.2 Better Buffered I/O

On a higher level, the buffered I/O library can now be trivially customized to use any number of sectors for internal disk buffering. A general purpose standard header file, BDSCIO.H, controls the buffering mechanism and also provides a standard nomenclature for some of the constant values most commonly used in C programs. I recommend that all users carefully examine BDSCIO.H, become intimate with its contents, and use the symbols defined there in place of the ugly constants previously abundant in the sample programs. For example, the symbol **ERROR** is a bit more illuminating than **-1**.

1.8.3 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.

1.8.4 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.

1.8.5 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.

1.8.6 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 <bdscio.h>
   ```

   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 BDSCIO.H can be used.

2. All buffers for file I/O that were formerly declared as 134-byte character arrays should now be declared as BUFSIZ-byte character arrays. For example, a declaration such as:

   ```
   char ibuf[134];
   ```

   becomes:

   ```
   char ibuf[BUFSIZ];
   ```

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 <u>within</u> 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 had accepted up to sixteen. To allow more than eight 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.

1.9 How to Use The Compiler

1.9.1 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 initializer and subroutine module
DEFF.CRL	Standard ("Default") function library
DEFF2.CRL	More library functions

CC.COM and CC2.COM together form the actual compiler. CC reads in a given source file from disk, crunches on it, leaves an intermediate file in memory, and automatically loads in CC2 to finish the compilation and produce a CRL file as output.[7] The CRL (mnemonic for C ReLocatable) file contains the generated 8080 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 automatically) for needed subordinate functions. When all such functions have been linked, a COM file is produced.

For convenience, the CLIB program is provided for the manipulation of CRL file contents.

1.9.2 Configuration

BDS C commands should simply come up running under any CP/M system, without any special configuration procedure necessary. There are several optional features of the compiler and linker that may be configured by the user to increase the flexibility of the package. This subsection explains each of those options and how to select them.

If Running MPM II:

If you are running BDS C under MP/M II, you must re-assemble the run-time package with the "MPM2" symbol equated to 1 (it comes configured to 0). Simply edit the CCC.ASM file, assemble it using ASM, use LOAD to create CCC.COM, then rename that to C.CCC.

1.9.2.1 CC and CLINK configuration

Make sure to have your master distribution disk safely tucked away somewhere before attempting these modifications!

There are several user-configurable features in CC.COM and CLINK.COM controlled by a specific bytes of memory very close to the beginning of each command file. In order to change these features, use DDT or SID to read CC.COM or CLINK.COM into memory, make the changes using the s command, hit control-C, and use the CP/M **SAVE** command to write the modified command back to disk.

7. 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

To convert the hex "NEXT" address printed by DDT or SID into the decimal number you must give to the **SAVE** command in order to save the modifed version to disk, use the following algorithm: first, take the leftmost two hex digits and compute their decimal equivalent (e.g., 3C80 yields 3C, which is 60 decimal). Then, subtract 1 from that **only if** the rightmost two digits are 00 (for example, the 60 above would remain 60 because the rightmost two digits of 3C80 are 80, not 00). The final value is the number to give **SAVE.**

Both CC.COM and CLINK.COM contain an identically structured five-byte configuration block. The base address of the block for CC.COM is 0155h, and for CLINK it is 0103h. The structure of the block is as follows:

Addr.	Function	Default value
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 (0 or 1)	01 (enabled)
base+4	Perform warm-boot when finished	00 (don't)

Note that each item in the block is exactly one byte in length.

The first two configuration bytes specify a default disk and user area to be treated as a "library directory", or "default area" by CC and CLINK. For CC, the library directory specifies where to find the files named in **#include** directives when the filename is enclosed in angle brackets[8], and also where to find CC2.COM for the second phase of compilation. For CLINK, this says where to find the files DEFF.CRL, DEFF2.CRL, DEFF3.CRL (if present) and C.CCC, as well as where to obtain other CRL files named on the CLINK command line that cannot be found in the directory from which the "main" CRL file was taken.

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 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 is to be the default library user area. Both the library disk and user area come configured to FFh; thus, the distribution version of the

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

v1.50 compiler and linker behave the same as earlier versions, in which the currently-logged drive and user area were always assumed to contain the library files by default.

The third configuration byte designates which CP/M drive contains the $$$.SUB file that exists during "Submit File" processing. The possible values are the same as for the default library disk as 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 $$$.SUB file on drive A, that is the way CC and CLINK come configured by default. But, if the user has customized his system to put the $$$.SUB file on, say, the current drive instead of always on drive A, then this byte would be changed from 01h to 0FFh.

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 (and final) byte controls whether CC and CLINK perform a warm-boot when done processing or return directly to the CCP without any disk activity. The comands 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 a non-zero value.

In summary: this configuration scheme allows 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.

Note that CC2.COM does not need to be configured; CC.COM passes it all the relevant information upon transfer of control.

1.9.2.2 Run-Time Package Options

CCC.ASM contains some equated symbols that may be customized by the user. See Chapter 2 for details of how to reassemble the run-time package and suitably modify the library.

> The beginning user should be warned that the run-time package reconfiguration process is rather involved, and the savings to be gleaned under any standard CP/M environment is usually not worth the trouble.

The NFCBS symbol in CCC.ASM 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, simply change this to the desired value (each additional file makes the run-time package about 38 bytes longer.)

1.9.2.3 BDSCIO.H and HARDWARE.H Configuration

The standard I/O header file BDSCIO.H contains the defined constant NSECTS, which controls the size of file buffers for the 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/deblocking 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.

System-dependent hardware characteristics, such as I/O port numbers, masks, etc., are kept in the HARDWARE.H header file. Before any programs which include it are compiled, HARDWARE.H should be modified to reflect the hardware characteristics of the target computer system.

1.9.3 A Typical Compilation

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

The compiler is invoked with the command:

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

After printing its sign-on message, CC will read in the file FOO.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 FOO.CRL to disk.

The next step brings in the linker:

```
        A>clink foo [other files & options, if any] <cr>
```

Unless there are unresolved function references, the file FOO.COM will be produced, ready for execution via

```
        A>foo [arguments] <cr>
```

> **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 <u>argc</u> and <u>argv</u> to be miscalculated.

Following are the detailed command syntax descriptions.

1.9.4 CC — The Parser

Command format: 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 file name may appear a list of compilation options, each preceded by a dash. The currently supported options are:

-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. This option is useful for detecting mismatched comment delimiters: an unclosed comment will make all subsequent text disappear during **-p**, and the last visible text tells you where the badly-delimited comment begins. Note that this output may be directed to the CP/M "list" device by typing control-P before invoking CC.

-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 specifier, then an intermediate ".CCI" file is written to disk for later processing by an explicit invokation of CC2, and no attempt is made to auto-load CC2.
-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.
-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.
-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 address specified by this option overlaps part of the program or the operation system in the final command file.

-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** but **don't** use **-o**.
CC2 must be successfully auto-loaded by CC in order for **-o** to have any effect.

-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 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.

-r x Reserves xK 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.

-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.

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 1.9.2). The resulting code is optimized for speed.

1.9.5 CC2 — The Code Generator

Command format: CC2 name <cr>

Normally CC2.COM is loaded automatically by CC 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 zero.
0104	01	-o	Zero if -o given (optimize for speed), else 1.
0105-6	0100h	-m	Origin address of C.CCC at object run-time.
0107-8	none	-e	Explicit external address (if -e given to CC).
0109	00	-e	True if explicit external addr given, else 0.

The 16-bit values must be in reverse-byte order (low order byte first, high last).

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 and control will return to command level.

Example:

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

1.9.6 CLINK — The C Linker

Command format: 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 in the configuration block (see section 1.9.2). 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 drive and user area (see section 1.9.2) 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 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:

-s

Print out a load map and statistics summary on the console.

-f (filename...)

Cause 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 eariler 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.

-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.

-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 zeroes.
-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[9]. 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.
-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 specifier 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.
-n	Makes the resulting COM file preserve the CP/M CCP (Console Command Processor) at run-time, instead of overlaying the CCP with the runtime 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.

---

9. 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
```

-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.)
-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.
-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.

-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.

-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 1.9.2).

-d ["args"] 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.

-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.

Examples:

```
A>clink ted -s -t 6000  -o joyce <cr>
```

Here, CLINK expects the file TED.CRL to contain a **main** function, which is then linked with all functions from TED.CRL and any needed functions from DEFF.CRL, DEFF2.CRL and, if it exists, DEFF3.CRL[10]. A statistics summary is printed out when finished, the run-time stack is set to start at 6000h and grow down (leaving

10. 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.

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:lois 6/c:vicky -f janet  -s  <cr>
```

In this example, CLINK loads all functions from LOIS.CRL (on drive B:) and VICKY.CRL (in user area 6 on drive C:), links in any needed functions from JANET.CRL (from disk B, since the disk where LOIS.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 1.9.2), and prints out a statistics summary when done. Since no **-t** option is given, CLINK assumes all the TPA (Transient Program Area) is available for code and data. The COM file generated is named LOIS.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.

1.9.7 CLIB — The C Librarian

Command format: 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 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 **appends** and **transfers** 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** or the **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 v1.50
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>
```

1.10 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.)

1.11 Strangenesses

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. 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.

1.12 Last Words

Please report bugs to:

Leor Zolman
BD Software
P.O. Box 9
Brighton, Massachusetts, 02135
(617) 782-0836 (evenings best, before 1:00 AM EST)

Please don't hassle distributors with technical bug reports; by reporting any bug you may encounter directly to BD Software, you'll vastly improve the chances of my ever hearing about the problem and supplying a fix within a short amount of time.

I gratefully thank the following individuals for their invaluable feedback and support during the debugging phase of this compiler's development:

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

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.

The BDS C User's Group has been organized; For information on how to get **inexpensive** updates of the compiler, receive the User's Group newsletter, and/or get access to contributed programs, contact:

BDS C User's Group
P.O. Box 287
Yates Center, Kansas 66783
(316) 625-3554

Note that the BDS C package is now available from the User's Group for sale to first-time buyers.

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 to link in machine code subroutines together with normally compiled C functions. It describes the CRL format in detail, and how to produce assembly language functions in CRL format so that they can be treated just like any other functions by the C Linker. The parameter-passing and calling conventions used for C functions are described, along with some useful subroutines existing in the run-time package.

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" format (with far less deviation from standard assembly language than the obsolete "CMAC.LIB" macro package of pre-1.46 releases) then automatically converts the .CSM source file into an .ASM source file for assembly with ASM.COM. A CP/M "Submit" file named CASM.SUB is provided to automate most of this 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 in order for completeness. So here goes...

2.2.1 CRL Directories

The first four sectors of a CRL file[11] 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 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[12].

Following 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[13]:

```
46   4F   CF   05   02   59   49   50   45   C5   50   03
F    O    O'   nn   nn   Y    I    P    E    E'   nn   nn

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

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

12. 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.

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

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 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.

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	48	41	d2
g	e	t	c	h	a	r'	p	u	t	c	h	a	r'

53	45	54	4D	45	CD	00
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
```
[14]

2.3 BDS C Register Allocation and Function Calling Conventions

2.3.1 The Stack

All argument passing on function invokation, 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 exactly 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, the value given as argument to **-t** is used to initialize the SP. If the -[illegible] option is used then the SP is

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

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** having to perform a warm-boot.

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 living on the one stack that 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.

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 nth 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 ot 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 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.

2.4.1 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       ;get offset into HL
        dad     d               ;add to base-of-externals 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            ;single-byte-offset, double-byte 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.

2.4.2 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.

2.4.3 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.

2.4.4 Source Files

CCC.ASM is the source for the run-time package, in which all the above mentioned routines are documented. The header file BDS.LIB contains definitions of all entry points to the routines within C.CCC (the assembled CCC.ASM) as provided in the distribution version of the package. All your CSM-format source files should contain the directive

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

so that the necessary subroutines may be referred to directly by name in your programs. If you have 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.

2.5 Generating Code to Run At Arbitrary Locations and/or In ROM

Normally, BDS C produces a CP/M transient command file ready to run in read/write memory located at the base of the user area (100h), in response to a direct command to the Console Command Processor (CCP). Under such normal circumstances, the run-time package (C.CCC) and its private read/write memory area occupy the first 1500-or-so bytes of the command file, and the compiled code (commencing with the "main function) follow immediately thereafter.

If all you ever want to do is generate CP/M transient commands, then you're all set. But in order to generate code that can run at a different location or be placed into ROM, it is necessary to: a) customize the run-time package, b) re-assemble the machine-coded portions of the function library, and c) recompile the C-coded portions of the library. Here is the general procedure for customizing the package in this manner:

1. Alter and re-assemble the run-time package (CCC.ASM) to reflect the desired configuration. If the target code will not be operating under CP/M, setting the appropriate EQU to zero will eliminate some CP/M-specific support code and reduce the size of both the run-time package and r/w memory area contained within the run-time package. Non-CP/M assembly will also cause the CP/M-dependent entry points within the run-time package to remain undefined, so you won't accidentally generate code which calls them in an environment where they are not defined. Also be sure to set the appropriate EQUs to define the code origin of the package and the r/w memory location for the package's private data area.

2. After assembling CCC.ASM, you cannot simply LOAD the CCC.HEX file to produce a binary image unless the origin is exactly at the base of the TPA. If your origin is elsewhere, use DDT or SID to read the file into memory and move it down to the base of the TPA, the re-boot CP/M and use the SAVE command to write the new C.CCC image back to disk in binary form. After the binary image of CCC.ASM is produced (be it named CCC.COM or whatever), rename it to be: C.CCC.

3. Edit the file BDS.LIB so that all addresses match the values obtained from assembly of your new CCC.ASM. A good way to check this step is to rename BDS.LIB to be BDS.ASM, assemble it, and compare the values at the left margin from BDS.PRN to those in CCC.PRN.

4. Using CASM, process the machine language library files (DEFF2A.CSM, DEFF2B.CSM and DEFF2C.CSM) yielding CRL files. If you are configuring the package for a non-CP/M environment, you'll probably want to first purge all the CP/M-related functions from the library files before assembly. Note that most file I/O and system-dependent functions have been placed in DEFF2C.CSM for convenience.

5. When using CC.COM to compile code for a non-standard load address, use the -m option to inform the compiler of the new run-time package origin address (if different from 0x100). Make sure to re-compile STDLIB1.C and STDLIB2.C using -m, and use CLIB to create a new DEFF.CRL composed of all functions from STDLIB1.CRL and STDLIB2.CRL.

6. Use the -l, -t and -e options to tell CLINK the load address, top of r/w memory and base of external data area, respectively, of the target program.

7. Burn the PROMs!

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.[15] This is a collection of useful C functions, in CRL (C ReLocatable) format, available for use by all C programs. CLINK automatically searches the library files[16] **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 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.

---

15. DEFF.CRL contains all the C-coded functions from STDLIB1.C and STDLIB2.C, while DEFF2.CRL contains all the assembly language functions from DEFF2A.CSM, DEFF2B.CSM and DEFF2C.CSM (assembled using the CASM facility).

16. 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.

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

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.

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.

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.)

inp(n)

Returns the eight-bit value present at input port n.
For memory-mapped input, use the peek function.

outp(n,b)

Outputs the eight-bit value b to output port n.
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, the 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 srandl.
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 new, "better quality" random number generator, written by Prof. Paul Gans to emulate the CDC 6600 random number generator in use at the Courant Institute of Mathematical Sciences. The initialization mechanism was later added for semi-compatibility with the srand and srandl conventions.
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 srandl 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, srandl 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.

```
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 execed was generated by the C compiler and it needs to share external variables with the execing 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 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-allignment restrictions. See the book for details.
The standard header file BDSCIO.H must be **#included** in **all files** of a program that uses alloc and free pair, since there is some crucial external data declared therein.

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).

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 BDSCIO.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

3.2.1 The CIO Function Package for Direct Console I/O

The getchar and putchar functions supplied in the standard library (and described below) do not allow absolute control over the console. Instead, they are designed to be the most useful for conventional applications without requiring any initialization or special thought. If you have an application in which it is important to have **complete** control over all characters sent to and received from the system console device, then assemble and use the CIO function package supplied in source form only (CIO.CSM) with the BDS C v1.50 distribution package. CIO provides alternate versions of the getchar, putchar and kbhit functions as well as a new function ttymode which supports changing console interface operating characteristics dynamically.

int getchar()

> Returns next character from standard input stream (CP/M console input.)
> Re-boots CP/M when control-C is typed.
> Carriage return echos CR-LF to the console output and returns a newline ('\n') 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 value 255 should be checked for instead to detect control-Z, but note that in this case an actual data value of 255 would be indiscernable 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.

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.
If a control-C is detected on console **input** during a putchar call, program execution will halt and control will return to command level, allowing the user to abort any program doing console output (via putchar calls) by typing control-C. Since the provided putchar function uses BDOS calls to both output characters to the console and check for input at the console, the special CP/M flow-control character (control-S) is recognized and may be used to freeze printouts done via putchar.

putch(c)
char c;

Like putchar, except that the console input is NOT interrogated for control-C during output, allowing type-ahead during console output on interrupt-driven systems. If you like this feature and want all putchar calls mapped into putch calls, simply place the preprocessor directive

```
#define putchar putch
```

somewhere in the BDSCIO.H header file, and be sure to include the header file in all programs.

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 termiating 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:

d	decimal integer format
u	unsigned integer format
c	single character
s	string (null-terminated)
o	octal 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 specifed 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.
An enhanced version of printf, incorporating the e and f format conversions for floating point values used in Bob Mathias's floating point package, is available for compilation in the file FLOAT.C.

```
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 assignment suppression character (*) works, but field width specification is not supported.
The arguments to scanf **must be pointers!!!!!.**
Note that input strings (denoted by a **%s** conversion specification in the format string) are terminated only when the character following the **%s** in the format string is scanned.
Returns the number of items successfully assigned.
For a more detailed description of scanf and printf, see Kernighan & Ritchie, pages 145-150.

3.3 String and Character 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.

```
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.

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

Copies the string s2 to location 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");
```

```
int strlen(string)
char *string;
```

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

```
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 values represented 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: UNIX C programs will sometimes assign negative values to character variables, since UNIX 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.

```
int getval(strptr)
char **strptr;
```

A spin-off from initw and initb:
Given a pointer to a pointer to a string of ascii values separated by commas, getval returns the current value being pointed to in the string and updates the pointer to point to the next value. (Why can't strptr be a simple pointer to characters?[17]
When the terminating null byte is encountered, a value of -32760 is returned. Initw will thus not accept a value of -32760. If you need to use that value, go into STDLIB.C and change the terminating value to some other value (you'll have to change getval and initw.)

17. Because the pointer-to-characters pointing to the text string must be **changed** by the getval routine; any value which is to be altered by a function must be manipulated through a **pointer** to that value. Thus, a "pointer to characters" must be manipulated through a "pointer to pointer to characters".

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 the buffered functions next.

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 string.

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 descriptor 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 specifier 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 specifier 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 certain bizarre 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

A new file I/O error diagnostic facility has been incorporated into BDS C v1.50. Whenever an error occurs, the usual -1 (ERROR) value is returned by the troubled function. Anytime this happens, 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 -1 (ERROR) is the only error indicator; write returns the number of sectors written, which should be considered an error if not equal to the number of sectors it was **told** to write.

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.

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.
Seeks should **not** be performed on files open for buffered I/O.

```
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 unlink(filename)
char *filename;
```

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

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

Renames the file in the obvious manner.
The specified file **must not be open** while rename is being used on it.
Returns -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 even reflect data written to new extents before they are closed, unlike the raw BDOS function (number 35) used to compute file size.

```
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

```
int fopen(filename, iobuf)
char *filename;
struct _buf *iobuf;
```

Opens the specified file for buffered (one datum at a time) input, and initializes the buffer pointed to by iobuf. Iobuf should be a BUFSIZ-byte area reserved for use by the buffered I/O routines. The value of BUFSIZ is determined by the BDS C standard I/O header file (BDSCIO.H), which should be **#include**-ed in any program using buffered I/O.
The technical structure of the buffer is

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

but all that really matters to the user is that it is a BUFSIZ-byte area, declarable by

```
char samplebuf[BUFSIZ];
```

Return value is the file descriptor for the opened file; it need not be saved after the initial test for an error, since the file descriptor value

is automatically maintained in the I/O buffer for use by all other buffered I/O functions.
-1 returned on error.

```
int getc(iobuf)
struct _buf *iobuf;
```

Returns the next byte from the buffered input file opened via fopen having buffer at iobuf. No special codes are recognized; control-Z comes through as control-Z (not -1), CR and LF are ordinary characters, etc.
The values 0 and 3 may be used in place of the iobuf argument with any buffered input function, to direct the input from the console or the reader:
"getc(0)" is equivalent to "getchar()".
"getc(3)" reads a character from the CP/M "reader" device.
-1 is returned on error **or** on physical end-of-file.
When reading in text files with getc, **both** the value 0x1a (CPMEOF) **and** the normal physical end-of-file value (-1, or ERROR) should be regarded as end-of-file markers, since some CP/M text editors neglect to place a 0x1a (control-Z, CPMEOF) byte at the end of text files under some circumstances.

```
ungetc(c, iobuf)
char c;
struct _buf *iobuf;
```

Pushes the character c back onto the input buffer at iobuf. The next call to getc on the same file will then return c. No more than one character should be pushed back at a time.

```
int getw(iobuf)
struct _buf *iobuf;
```

Returns next 16 bit word from buffered input file having buffer at iobuf, via two consecutive calls to getc.
-1 returned on error.

```
int fcreat(filename, iobuf)
char *filename;
struct _buf *iobuf;
```

Creates a file named filename (first deleting any existing file by the same name) and opens the file for buffered output. Iobuf should point to a BUFSIZ-byte buffer.
Returns the fd for the file, or -1 on error.

```
int putc(c, iobuf)
char c;
struct _buf *iobuf;
```

Writes the byte c to the buffered output file having buffer at iobuf. Iobuf should have been initialized by a call to fcreat.
No translations are performed; text lines can be separated by either CR-LF combinations (for compatibility with standard CP/M software) or by newline characters a la UNIX (for increased efficiency and straightforwardness.)
The values 1 through 4 may be used in place of iobuf with any buffered output routines 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,1)" is equivalent to "putchar(c)".
"putc(c,2)" writes the character to the CP/M "list" device.
"putc(c,3)" writes the character to the CP/M "punch" device.
"putc(c,4)" writes the character to the standard **error** stream, which is always the console output 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.
When writing out text to a file, be sure to terminate the text with a control-Z (0x1a, CPMEOF) byte.
Returns -1 on error.

```
int putw(w, iobuf)
struct _buf *iobuf;
```

Writes the 16 bit word w to buffered output file having buffer at iobuf, via two consecutive calls to putc.
Returns -1 on error.

```
int fflush(iobuf)
struct _buf *iobuf;
```

Flushes output buffer iobuf, 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 aborted before the exit routine closes all files).
Fflush is for use with buffered output files; attempting to use it on an input file will have no affect.
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(iobuf)
struct _buf *iobuf;
```

Closes the specified buffered I/O file (it may have been opened for either reading (via fopen) or writing (via fcreat). If the file was opened for writing, then an automatic fflush is performed to flush the output buffer before closing the file.
NOTE: Before closing a buffered output file that has had text written to it, be sure to put out a CP/M "End of text-file" marker (CPMEOF) to the file.

```
int fprintf(iobuf, format, arg1, arg2,...)
struct _buf *iobuf;
char *format;
```

Like printf, except that the formatted output is written to the buffered output file having buffer iobuf instead of to the console.
Returns -1 on error.

```
int fscanf(iobuf, format, arg1, arg2,...)
struct _buf *iobuf;
char *format;
```

Like scanf, except that the text input is scanned from the input buffer iobuf instead of from the console. The present version of fscanf requires that each line of data be scanned completely; any items left on a line read from a file after all format specifications have been satisfied will be discarded.
Returns the number of items successfully assigned, or -1 if an error occurred in reading the file.

```
char *fgets(str, iobuf)
char *str;
struct _buf *iobuf;
```

Reads a line in from the specified buffered input file and places it in memory at the location pointed to by str.
This one is a little tricky due to the CP/M convention of having both a CR (carriage-return) and LF (newline) at the end of text lines. In order to make text easier to deal with from C programs, fgets automatically strips off the CR from any CR-LF combinations that come in from the file. Any CR characters not immediately followed by LF are left intact. The LF is included as part of the string, and is followed by a null byte.
There is no check on the length of the line being read in; care must be taken to make sure there is enough room at str to hold the longest line imaginable (a line must be terminated by a newline character before it is considered complete).
Zero is returned on EOF, whether it be a physical EOF (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, iobuf)
char *str;
struct _buf *iobuf;
```

Writes the null-terminated string from memory at str into the specified buffered output file. Newline characters are converted into CR-LF combinations to keep CP/M happy. If a null (zero byte) is found in the string before a newline, then there will be no line terminator at all appended to the line on output (allowing partial lines to be written.)

3.5 Plotting Functions for DMA Video Boards

setplot(base, xsize, ysize)

Defines the physical characteristics (starting address, dimensions) of a memory-mapped "DMA" video board such as the Processor Technology (R.I.P) VDM-1. Base is the starting address of the video memory, xsize is the number of lines in the display, and ysize is the number of characters per line. Setplot need only be called once at the start of program execution; from then on, the functions clrplot, plot, txtplot and line will know about the given parameters.

clrplot()

Clears the memory-mapped video screen (fills with ASCII spaces.)

plot(x, y, chr)
char chr;

Places the character chr at coordinates (x,y) on the video screen. (x,y) is read as: x down, y across, where

```
0 <= x < xsize,
0 <= y < ysize.
```

txtplot(string, x, y, ropt)
char *string;

Places an ASCII string on the screen at position (x,y); If ropt is non-zero, then each byte of the string is logical OR-ed with the value 0x80 before being displayed. This forces the high-order bit to a 1, causing the character to appear in reverse-video on some boards (such as the VDM-1) or do other funny random things with other boards.

line(c, x1, y1, x2, y2)

This function draws a crooked line (because there is no **way** to make a line look **straight** with 64 by 16 resolution!) between the points (x1,y1) and (x2,y2) inclusive. The line is made up of the character _c_. _Line_, as distributed, only works with a 64 by 16 board.

Chapter 4

Notes to APPENDIX A of "The C Programming Language"

4.1 Introduction

BDS C is designed to be a subset of UNIX C. Therefore, most parts of the C **Reference Manual** apply to BDS C directly; the purpose of this appendix is to annotate the sections that BDS C does not follow to the letter.

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**[18]. 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[19]. 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

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

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

```
        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:

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

Case is ignored 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[20]. The external variable mechanism in BDS C is handled much like the unnamed COMMON facility of FORTRAN. For example: if 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.

As long as all common external declarations are kept in a single ".H" file, and **#include** is used within each source file of a program to read in the ".H" file, there shouldn't be any trouble. Well, relatively little anyway.

---

20. 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.

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 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.

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-specifier declaration-list ;
```

There are no "storage class" specifiers.

8.1 Storage class specifiers

Not implemented.

8.2 Type specifiers

The type-specifiers are

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

The type **register** will be 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.

Alas, BDS C wouldn't 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.

One significant "misfeature" of BDS C is 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. Due to the relative simple-mindedness of the BDS C compiler (and its programmer), the preceding declaration ends up having the same meaning as

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

On the brighter side, 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 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-specifier:
        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-specifier 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

Sorry; no initializers allowed.

All external variables are now automatically initialized to zero (note that this was not true for pre-1.50 versions of the compiler).

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

Here the book is slightly confusing (and if the book didn't confuse you, the following clarification surely will...)

The **for** statement is not completely equivalent to the **while** statement as illustrated, 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.[21] 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 BDSCIO.H in your program, since there are several external data objects required by alloc and free declared in BDSCIO.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 designators 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

21. 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.

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.

The **sizeof** operator cannot correctly evaluate the size of an array.

18.2 Declarations

The complete syntax for declarations is

declaration:
    type-specifier declarator-list **;**

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

declarator-list:
    declarator
    declarator **,** declarator-list

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

struct-or-union-specifier:
        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-specifier 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[22], 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.

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

Appendix A

Miscellaneous Notes

- The = operator is used for assignment only. The relational operator is equal to is represented by the == operator. Be careful not to confuse them; using the wrong one will never cause the compiler to generate any diagnostic messages, because the resulting expressions will be syntactically correct even if they don't have the desired effect.

- 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 is not especially intelligent in some 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.** 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 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 **cases** 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;
   ```

   The same operation can be written as

   ```
   x = (a < b);
   ```

   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.

6. 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");
   ```

   may just as well be written as

```
        if (a)  printf ("A is non-zero");
and     if (!a) printf ("A is zero");
```

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.

Appendix B

Error Messages Explained

B.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

Note: an unterminated comment can draw all sorts of strange error messages from the compiler. If you get one of the following messages and have no clue to the cause, try giving the **-p** option to CC and check if the code just seems to "cut off" at some point in the printout. If so, that's probably the location of an unclosed comment, since all the following text would be 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 some similar typo.

Missing semicolon
: This error usually means just what it says, but keep in mind that there are some cases where a missing semicolon will draw a less meaningful error message.

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 function
: 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 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 specifier (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 strucure 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 is encountered for a particular **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.

B.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 wierd 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. A prevalent cause: are you attempting to subscript the **argv** formal parameter in you main function without having declared it 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
At attempt 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
Something illegal 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.

B.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
: Not enough 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 tne BDS C User's Group.

Appendix C

Some Mistakes Commonly Made By Beginning C Programmers

There are several aspects of the C language that tend to cause a great deal of brow-beating when tackled 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.

C.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 hacker might interpret the = operation as a conditional test instead of the assignment expression it is in actuality.

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.

C.2 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.

C.3 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; i.e, the declaration

```
        char *foo;
```

would be later followed by a statement such as

```
        *foo = 'a';
```

before foo is ever initialized, 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;
        ...
        *foo = 'a';
```

C.4 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 **deallocated** 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!!

C.5 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, then 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 wholla! 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.

C.6 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 former example, though, endext is implicitly declared (in context) as a function returning an **int**.

Appendix D

Dynamic Overlays in C Programs

In order to allow C programs to be longer than physical memory without resorting to exec or execl (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, we must first decide just where the swapped-in code is to reside. Earlier versions of BDS C had local data frames growing up from low memory, starting where the externals ended. This made it difficult to determine the lowest memory location safe to swap overlays into. The scheme suggested then for handling overlays was to leave sufficient room between the end of the root segment code and start of the external data area to accommodate the largest possible swapped-in segment combination.

BDS C presently allocates all storage for local data on the high-memory stack. The original overlay scheme is still recommended, though; here is the modified memory map, accommodating this method of handling overlays:

```
low memory:  base+100h:  C.CCC run-time utility 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.

In version 1.5, 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 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 swappable 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[23] 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 swappable 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. To actually execute the segment, you have to call it 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 */
...
```

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 STDLIB2.C. 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 ram+115h...if you've never done any

---

23. 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.

low-level hackery, you get the value of the 16-bit address at location BASE+115h by using indirection on a pointer-to-integer (or -unsigned.) Note that location BASE+115h 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>
```

where 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 swappable 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 E

The CASM Assembly-language-to-CRL-Format Preprocessor For BDS C v1.50

The only means previously provided to BDS C users for creating relocatable object modules (CRL files) from assembly language programs was a painfully complex macro package (CMAC.LIB) that only operated in conjunction with Digital Research's macro assembler (MAC.COM). This was especially bad because MAC, if not already owned, cost about as much as the entire BDS C package to purchase. This document describes the program "CASM", supplied to eliminate the need for "MAC". CASM is a preprocessor that takes, as input, an assembly language source file of type ".CSM" (mnemonic for C aSseMbly language) in a format much closer to "vanilla" assembly language than the bizarre craziness of CMAC.LIB, and writes out an ".ASM" file which may then be assembled by the standard, ubiquitous CP/M assembler (ASM.COM). CASM automatically recognizes which assembly language instructions require relocation parameters, and inserts the appropriate pseudo-operations and extra opcodes into the resulting ".ASM" 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.

E.1 Creating CASM.COM

CASM is supplied in source form only 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.

E.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
CASM.SUB	Submit file for performing the entire conversion of a CSM file into CRL format
ASM.COM (or MAC.COM)	Standard CP/M utility, for assembling the output of CASM.
DDT.COM (or SID.COM)	Standard CP/M utility, for converting the HEX output of the assembler into binary CRL format.

The pseudo-operations that CASM recognizes as special control commands within a .CSM file 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"

This op 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:

```
        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. 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.

3. **INCLUDE** (and **MACLIB**) only work for one level of inclusion.

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.

6. The .HEX file produced by ASM after assembling the output of CASM cannot be converted into a binary file by using the CP/M LOAD command; instead, DDT or SID must be used to read the file into memory, and then the CP/M SAVE command must be issued to save the file as a .CRL file. CASM inserts a line into the ASM file ending in the character sequence "!.",

specifically so that the line will be flagged as an error... the user may then look at the value printed out at the left margin to see exactly how many 256-byte blocks need to be SAVEd after using DDT or SID to get the file into memory. The reason that LOAD cannot be used is that CASM puts out the code to generate the CRL File directory at the end of the ASM file, using the "ORG" pseudo-op to set the location counter back to the base of the TPA. The LOAD command aborts with the cryptic message "INVERTED LOAD ADDRESS" when out-of-sequence data of this nature is encountered. Rather than having CASM write out the directory information into a new file and then append the entire previous output onto the end of this new directory file, I decided to require the user to enter a SAVE command.

7. The CASM.SUB submit file may be used to perform the entire procedure of converting a .CSM file to a .CRL file, except for entering the final SAVE command. For a file named "FOO.CSM", just say:

```
submit casm foo
```

and enter the "SAVE" command just the way it instructs you to when processing is complete.

Appendix F

BDS C File I/O Tutorial

F.1 Introduction

The library functions provided with BDS C for performing file input/output fall into two major catagories: 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, as 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.

F.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);      /* "filename" can be either a literal */
                               /*  string or any expression that      */
fd = creat(filename);          /*  evaluates to a character pointer  */
```

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[24], as long as

24. 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.

what they are asked to do is possible, while buffered functions only go to disk when a buffer 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, followingread 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:

```
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 hunk from file1 and write it out to file2;
                if (any error has ocurred)
                        complain and abort
        }
        close all files;
}
```

And here is the actual C program to perform the copy operation:

```
#include <bdscio.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 */

                           /* make sure exactly 2 args were given    */
        if (argc != 3)
                perror("Usage: A>copy file1 file2 <cr>\n");

                           /* try to open 1st file; abort on error */
        if ((fd1 = open(argv[1],0)) == ERROR)
                perror("Can't open: %x\n",argv[1]);

                           /* create 2nd file, abort on error: */
        if ((fd2 = creat(argv[2])) == ERROR)
                perror("Can't create: %s\n",argv[2]);

                           /* Now we're ready to move the data:      */
        while (oksects = read(fd1, buffer, BUFSECTS)) {
                if (oksects == ERROR)
                        perror("Error reading: %s\n",argv[1]);
                if (write(fd2, buffer, oksects) != oksects)
                        perror("Error; probably out of disk space\n");
        }

                           /* Copy is complete. Now close the files: */
        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 BDSCIO.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[25], 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

25. This has no effect if called before the file was opened, as in the case where the wrong number of parameters have been given and the "argc != 3" test succeeds.

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.

F.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, logically, an I/O buffer. Within this structure is a large character 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 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, or 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, getc, and fclose. Fopen is called to open an existing disk file, associate it with a user-provided I/O buffer, and initialize that buffer for receiving data from the file. Getc 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 there are the functions fcreat, putc, fflush, and fclose again (fclose leads a double existence). Fcreat creates a new file and prepares an associated I/O buffer structure for recieving output. The data is written to the buffer via calls to putc, one byte at a time; whenever a putc 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 only functions that actually read and write data are getc and putc; functions such as fgets, fputs, fprintf, etc. do their reading and writing in terms of getc and putc.

Careful examination of the BDSCIO.H header file will reveal that the number of sectors useo 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 <bdscio.h>

main(argc,argv)
char **argv;
{
        char ibuf[BUFSIZ];          /* declare I/O buffer          */
        char linbuf[MAXLINE];       /* temporary line buffer      */
        int lineno;                 /* line number variabele      */

        if (argc != 2) {  /* make sure file was given */
                printf("Usage: A>pnum filename <cr> \n");
                exit();
        }

        if (fopen(argv[1],ibuf) == ERROR) {
                printf("Can't open %s\n",argv[1]);
                exit();
        }

        lineno = 1;                 /* initialize line number    */

        while (fgets(linbuf,ibuf))
                printf("%3d: %s",lineno++,linbuf);

        fclose(ibuf);
  }
```

The declaration of ibuf provides the I/O buffer area for use with fopen, getc and fclose. The symbolic constant BUFSIZ, defined in BDSCIO.H, tells how many bytes an I/O buffer must contain. This value will vary with the number of sectors desired for data buffering, as described above.

After checking the argument count and opening the specified file for buffered input, 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 preperation 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 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 <bdscio.h>

main(argc,argv)
char **argv;
{
        char ibuf[BUFSIZ], obuf[BUFSIZ];
        int c;

        if (argc != 3) {
                printf("Usage: A>ucase file newfile <cr> \n");
                exit();
        }
        if (fopen(argv[1],ibuf) == ERROR) {
                printf("Can't open %s\n",argv[1]);
                exit();
        }
        if (fcreat(argv[2],obuf) == ERROR) {
                printf("Can't create %s\n",argv[2]);
                exit();
        }

        while ((c = getc(ibuf)) != EOF && c != CPMEOF) {
                if (putc(toupper(c),obuf) == ERROR)
                        printf("Write error; disk probably full\n");
                        exit();
        }

        putc(CPMEOF,obuf);
        fclose(obuf);
        fclose(ibuf);
 }
```

This time there are two buffered I/O streams to be dealt with: the input file and the output fiie. The first task is to check if 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 and fcreat are called, 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 two possible end-of-text-file values: EOF and CPMEOF. Normally, the last thing in a text file **should** be a CPMEOF (control-Z) character. But, some text editors neglect to place the CPMEOF character at the end of a file if the file happens to end exactly on a sector boundary; in this case, CPMEOF will never be seen and the physical end-of-file value (EOF) must be detected. The complication this causes is rather tricky...the EOF value returned by getc is -1, which must 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 = getc(ibuf)
```

would result in a value having the type **char** and could never possibly equal EOF as tested for in the program. If getc ever returned EOF in such a case, "c" would end up being equal to 255 (the **char** interpretation 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 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 getc to upper case, and then we use putc to write the resulting byte out to the output file. To be neat, errors are checked for: the program terminates if putc returns ERROR.

As soon as an end-of-file condition is detected, we write out a final CPMEOF (control-Z) character to terminate the output file. The way this particular program is set up, the CPMEOF will be appended to the output file whether or not the input file ended with a CPMEOF. Finally, fclose is used to close the input and output files.

For a large-scale example of buffered I/O usage, see CASM.C. Also, take some time to inspect the files BDSCIO.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.

Appendix G

BDS C Console I/O: Some Tricks, Clarifications and Examples

G.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.

When the major documentation for BDS C (i.e. the User's Guide) was originally prepared several years ago, I made the stuffy assumtion that all users would realize how the bdos and bios library functions could be used to perform direct console I/O through the BDOS and BIOS. In reality, the use of the bios and bdos functions for such purposes might only be self-evident to CP/M system programmers, and even then only to those programmers having experience driving the CP/M console from assembly or machine language programs.

G.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[26].

G.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 (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).

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

26. 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.

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 carraige-return character to a '\n' on input, etc. One early user remarked that he would like putchar to be immune from control-C; for him I added the putch library function, which works just like putchar except that control-C doesn't stop the program when typed at the console. A bit later it became evident that neither putchar nor putch are adequate when CP/M must be prevented from ever even sampling the physical console input. At that point I added the bios function, so that users could do their I/O directly through the BIOS and totally bypass the frustrating character-eating BDOS.

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 character from the console */
{
        return bios(3);
}

kbhit()         /* return true (non-zero) if a character is ready */
{
        return bios(2);
}

putchar(c)      /* write the character 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
          ultra-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) bios(0);    /* on Ctl-C, reboot   */
          if (c == CPMEOF) return -1;                 /* turn Ctl-Z to -1   */
          if (c == '\r') {                /* if CR typed, then      */
                    putchar('\r');        /* echo a CR first, 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, */
                    bios(0);                       /* then reboot   */
}                                  /* else ignore the input completely */
```

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 <u>getchar</u> and <u>putchar</u> (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 program RALLY.C (available from the BDS C User's Group). 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 RALLY.C, at the expense of non-portability for the object code; each user must individually configure his header files to define the unique port numbers, bit positions and polarities of the I/O hardware controlling his console. It would have been much easier if the BIOS contained just one more itty bitty subroutine to test console output status, but life is tough sometimes.

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 <u>getchar</u>, <u>putchar</u>, etc. really work, see their source listings in DEFF2A.CSM. And if there's something you want to do with the console and can't figure out how despite this document, I'm always available for consultation (at least whenever I'm near the phone.)

<u>G.4 The CIO Function Library</u>

A new utility package named CIO (supplied in source form as CIO.CSM) has been included with BDS C v1.50 for use in applications requiring total direct console I/O control.

Appendix H

The Floating Point Function Package

Bob Mathias

H.1 Introduction

The components of the floating point package are:

1)	FLOAT.C:	File of support functions, written in C
2)	FP:	The workhorse function (in DEFF2.CRL)
3)	FLOATSUM.C	A Sample use of all this stuff

Here's how it works: for every floating point number you wish to work with, you must declare a five (5) element character array. Then, pass a pointer to the array whenever you need to specify it in a function call. Each of Bob's functions expects its arguments to be pointers to such character arrays.

The four basic arithmetic functions are: fpadd, fpsub, fpmul and fpdiv. They each take three arguments: a pointer to a five character array where the result will go, and the two operands (each a pointer to a five character array representing a floating point operand.)

Note that the result may be placed into either of the arguments with no ill effects. I.e., the operation:

```
fpmult(foo,foo,foo);
```

will successfully square foo and place the result in foo.

To initialize the floating point character arrays to the values you desire and print out the values in a human-readable form, the following functions are included:

- ftoa Converts a floating point number to an ASCII string (which you can then print out with "puts"). NOTE: Explicit use of this function is not necessary when using the specially customized printf facility in FLOAT.C.
- atof Converts an ASCII string (null terminated) to a floating point number.
- itof Converts integer to floating point.

H.2 Detailed Function Summary

The following functions allow BDS C compiler users to access and manipulate real numbers. Each real number must be allocated a five (5) byte character array (char fpno[5]). The first four bytes contain the mantissa with the first byte being the least significant byte. The fifth byte is the exponent.

```
fpcomp(op1, op2)
char op1[5],op2[5];
```

Returns:

```
1   if op1> op2
-1  if op1< op2
0   if op1= op2
```

As with most floating point packages, it is not a good practice to compare for equality when dealing with floating point numbers.

```
char *fpadd(result, op1,op2)
char result[5], op1[5], op2[5];
```

Stores the value ofop1 + op2 in result. op1 and op2 must be floating point numbers. Returns a pointer to the beginning of result.

```
char *fpsub(result, op1, op2)
char result[5],op1[5],op2[5];
```

Stores the value of op1 - op2 in result. op1 and op2 must be floating point numbers. Returns a pointer to the beginning of result.

```
char *fpmult(result, op1, op2)
char result[5],op1[5],op2[5];
```

Stores the value of op1 * op2 in result. op1 and op2 must be floating point numbers. Returns a pointer to the beginning of result.

```
char *fpdiv(result, op1, op2)
char result[5],op1[5],op2[5];
```

Stores the value of op1 / op2 in result. op1 and op2 must be floating point numbers. A divide by zero will return zero as result. Returns a pointer to the beginning of result.

```
char *atof(op1, s1)
char op1[5],*s1;
```

Converts the ASCII string s1 into a floating point number and stores the result in op1 The function will ignore leading white space but NO white space is allowed to be embedded withing the number. The following are legal examples:
"2", "22022222222383.333", "2.71828e-9", "334.3333E32".
"3443.33 E10" would be ILLEGAL because it contains an embedded space. The value of the exponent must be within the range: -38 <= exponent <= 38. A pointer to the result is returned.

```
char *ftoa(s1, op1)
char *s1,op1[5];
```

Converts the floating point number op1 to an ASCII string at s1. It will be formatted in scientific notation with seven (7) digits of precision. The string will be terminated by a null. Returns a pointer to the beginning of s1.

```
char *itof(op1, n)
char op1[5];
int n;
```

Sets the floating point number op1 to the value of integer n. n is assumed to be a SIGNED integer.

H.3 General observations

Floating point operations must be thought of in terms of **function calls** rather than simple in-line expressions; special care must be taken not to confuse the abilities of the compiler with the abilities of the floating point package. To give a floating point number an initial value, for instance, you cannot say

```
char fpno[5];
fpno = "2.236";
```

Instead, to achieve the desired result you'd have to say:

```
char fpno[5];
atof(fpno, "2.236");
```

Moreover, let's say you want to set a floating point number to the value of an integer variable called "ival". Saying

```
char fpno[5];
int ival;
...
fpno = ival;
```

will not work; you have to change that last line to

```
itof(fpno, ival);
```

Some more examples:

The following will add 100.2 and -7.99, storing the result at the five character array location _a_:

```
fpadd(a, atof(b, "100.2"), atof(c, "-7.99"));
```

(note that _b_ and _c_ must also be five character arrays)

The following would **not** add 1 to _a_ as both _op1_ and _op2_ must be floating point numbers (actually pointers to characters...)

```
fpadd(a,a,1);  /* bad use of "fpadd" */
```

All of the above functions are written in C, but most of them call a single workhorse function called _fp_ to do all the really hairy work. This function has been placed into the DEFF2.CRL; it is the only machine-coded part of the package. The source code for the _fp_ function is available from the BDS C User's Group, or send a SASD (Self-Addressed, Stamped 8" Disk) to BD Software for a copy.

Appendix I

A Long Integer Package for BDS-C

Rob Shostak
August, 1982

I.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.

I.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 (availalbe 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 file LONG.CRL contains the compiled functions given in LONG.C, and DEFF2.CRL contains the workhorse function long.

Appendix J

The TELEDIT Telecommunications Program and Mini Screen-Editor v1.1

This Documentation by Nigel Harrison

Editor's note: "Teledit" is a cross between the original BDS Telnet program, Ward Christensen's MODEM program, its MODEM7 derivative, a C version called XMODEM, and Nigel's tiny screen editor. In fact, this program was used for a while as the primary editor on a CDC 110 micro system, until they got Mince up. It may not be the LAST modem program you'll ever need, but it's gotta have more roots than any other! TELEDIT is available from the BDS C User's Group if not included on the BDS C distribution disk.

-leor

Teledit is a communications program for transmitting files and connecting to other systems or networks as an ASCII terminal. It has a simple editor with which one can manipulate lines of text collected during a session with a remote system. The file transmission modes allow sending/receiving either binary or text files. The modes selected from the menu are:

T: Terminal mode - no text collection

Teledit 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 The SPECIAL character of <ctrl>shift uparrow was chosen because it is unlikely to be struck accidentally. To change the SPECIAL character, recompile Teledit 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. While in terminal mode, the editor is entered by holding down the control key while typing the letter "E" (control-E). X mode prompts the user for a filename to be used for the gathered text. After 500 lines of text have been collected and saved in the text buffer, each additional line collected causes the console bell (alarm) to sound. 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.

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 Teledit, typically in between file transfers to inform the person of the next file to be transmitted.

E: Edit text collected

Enters the editor from the menu display. This will not work until X: terminal mode with teXt collection has been entered and a text collection file opened. Editor commands are described below.

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 Teledit 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 TELED.C source file.

Screen Editor

The editor prompts with "*" when entered. The current line usually appears directly below the prompt. The editor commands are:

A	Append (Yank in) a file and insert it before the current line.
B	Go to the Beginning of the file and display a page. The current line becomes the first line of the text file.
F	Find line that contains the pattern following the "F". If found, it becomes the current line. Search starts forward and wraps around.
I	Enters Insert mode. To leave insert mode, <ctrl>Z is struck, followed by <cr>. The escape insert mode sequence should only be entered at the beginning of a line; lines having an embedded Z are lost.

K	Kill the current line.
nK	Kill n lines, where n is a decimal number.
L<pattern>	Find line that begins with the pattern following the "L".
O	Overwrite lines of text until <ctrl>Z is struck. <ctrl>Z should be struck only at the beginning of a new line.
P	Page forward in text.
-P	Pages backward.
Q	Quit editor.
Sn<cr>	Sets screen size to n lines. The default is 22 for 24-line displays. S23 should be used for 25-line consoles, and S28 for the Control Data 110.
Z	Go to Zee end of the file. Convenient, for example, to go to the end then -P to view last page.
n<cr>	Move forward in the file by n line(s).
<cr>	Move forward in the file by one line.
-n<cr>	Move backward in the file by n line(s).
Space Bar	Move backward in the file by one line.
#	print number of text lines in file.

Installation

Teledit must be compiled in BDS C with the following constants special to the installer's particular environment:

```
#define HC "s*"          /* where s is the string necessary to home the
                            cursor on the user's console screen */

#define CLEARS "s"       /* where s is the string necessary to clear the screen
```

```
                    on the user's console       */
```

Both the BDSCIO.H and HARDWARE.H header files should be properly configured for the target computer configuration before TELED is compiled.

Any of the editor commands not needed can be removed by removing the corresponding case statement from the editor function in the source code, then recompiling Teledit.

Appendix K

CDB: A Debugger for BDS C

Version 1.2
4 November 1982

David Kirkland
5915 Yale Station
New Haven, Connecticut 06520
(203) 787-9764

Note: This appendix is the major portion of the documentation for the CDB debugging package, which is either distributed with BDS C or available from the BDS C User's Group for a nominal charge. The information provided here should be sufficient to let you decided whether or not to purchase the CDB package in case it is not included in the standard BDS C distribution package due to space limitation or other random logistic problems.

K.1 Introduction — An Explanation of the 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, and is a replacement for the standard CLINK distributed with BDS C. The L2 in the debugging package is a slightly modified version of the L2 linker written by Scott Layson of Mark of the Unicorn. The new L2 incorporates all the features of Layson's L2, along with the debugging features and minor bug fixes. 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 (whom I will refer to as the "user" in this document) to invoke the debugger. CDB interprets the command-line arguments entered by the user, prepares various in-memory data tables, and invokes CDB2.OVL, the final element of the package. CDB2, which resides in high memory immediately below the CP/M BDOS, loads the program to be debugged (the "target program") at the base of the TPA (the CP/M "transient program area," which starts at 0100 hex in normal CP/M systems). CDB2 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, which begins execution. 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. (Note: the proceeding sentence is not literally true; not every occurrence of one of the enumerated events causes CDB2 to be invoked. See the discussion below in part IV B of "system libraries" and the **-s** and **-ns** options to L2.)

In this document, square brackets [] are used to signify optional elements, that is, elements that can be omitted.

K.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.

Constructing L2

Because L2 needs no customization, there is no need for each user to prepare his own version. The version of L2 supplied obtains C.CCC and DEFF*.CRL from the currently logged disk; 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 L2 documentation entitled "The Mark of the Unicorn Linker for BDS C" by Scott W. Layson, which is distributed with the CDB package. However, two changes to the procedure must be observed: (i) there is no source file SCOTT.C that needs be used and (ii) the **-e** option to CC should be specified as "-e4800" if you are going to use another version of L2 to link the debugger version of L2, or "-E4c00" if you will use CLINK to link L2. In brief, the procedure is:

```
cc l2.c -e4c00          [ or -e4800]
cc chario.c
clink l2 chario        [ or l2 l2 chario ]
```

Where to put CDB2

Before constructing either CDB.COM or CDB2.OVL, it is first necessary to decide where CDB2.OVL will reside in memory. 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 0x4600 bytes long (that's 18K decimal); 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 fairly complex expressions that must be parsed (recursively) to dump symbolically variable contents, I think discretion is the better part of valor. Adding the numbers up, we get a total of 0x5400 bytes for the code, globals, and stack for CDB2; thus, CDB2 must start 0x5400 bytes below the start of the BDOS. Because my BDOS starts at 0xE406, my CDB2 sits at 0x9000 (and I will use this value in the examples that follow). (If you do not know the address of your BDOS, the simplest way to discover it is to use DDT to examine the address field of the jump instruction at location 0005. To do this, first type "ddt"; once DDT prompts you with a "-", type "L5". Your BDOS starts at the address listed in the first line of DDT's response, and should end in "06".)

The distribution disk contains a version of CDB.COM and CDB2.OVL set up for a system with a BDOS at or above D300. 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 D300, you may wish to customize the debugger to give you more memory for the target program; and if your system has its BDOS below D300, you must customize to get a working debugger.

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. While editing CDB.H, you may 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 (without the colon), such as "A", 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.

Constructing CDB

After changing CDB2ADDR in CDB.H, you are ready to compile the two source files for CDB:

```
cc cdb.c -e3200
cc build.c -e3200
l2 cdb build
```

The submit file CDB.SUB is provided to perform the above sequence.

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 0x4600 (the size of the code, given above in section B) 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 can either type

```
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
```

where xxxx is to be replaced by the external address (e.g. d600) or use the submit file CDB2.SUB. To use CDB2.SUB, simply type

```
submit cdb2 xxxx
```

Again, xxxx is the location of the externals. Or, say

```
submit cdb2 xxxx d:
```

where d is the drive letter on which the source files reside if not on the default drive.

Once the C files are compiled, you need to assemble the one assembler source file, DASM.CSM, which is written in the format described in "The CASM.C Assembly-language-to-CRL-Format Preprocessor", by Leor Zolman (distributed with BDS C). As described more fully in that document, enter the commands

```
casm dasm
asm dasm
ddt dasm.hex
g0
save 3 dasm.crl
```

To save you this inconvenience, especially if you don't have a compiled version of CASM 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. The submit file LCDB2.SUB is provided to perform the above linkage.

The debugger is now ready to use!

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**.

K.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.

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
```

Linkage: The -D, -S, and -NS Options of L2

To link the target program, the user must use the L2 provided with the package instead of CLINK. As described in the L2 documentation, L2 has a different command line syntax from CLINK; in addition, the debugger version of L2 has the following additional options:

-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** or **-ns** options, 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. The **-d** option without the **-s** or **-ns** option is thus similar to "-s deff deff2 deff3".
-NS	Specifies that there are to be NO system library files; this option is used to override the default that DEFF*.CRL are system libraries. For example,

```
12 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 (with an optional user number prefix) 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.

An standard invocation of CDB is:

```
cdb target
```

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 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 bar lib
```

The -l operand tells CDB that the files 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. Both local and global symbol information from FOO.CDB is loaded.

K.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.

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 (as in

```
a = 5; putchar('x'); while (*s) s++;
```

for example), a decimal notation is used; the first statement in line n is numbered n.0, the next n.1, etc. (So in our example, assuming the given line is the 7th line in a function, "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.

So far, no mention has been made to the count operand. 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.

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 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.

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]
```

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.

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.

K.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 (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.

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]

LIFEBOAT ASSOCIATES SOFTWARE PROBLEM REPORT

Please use this form to report errors or problems in software supplied by Lifeboat Associates. This form is designed to act as a transmittal sheet.

Software Product Name: ____________________ Media Format: __________

Version No.: __________ Serial No.: __________ Invoice No.: __________

Purchased From: ______________________________

____

____

Date of Purchase: ______________ Return Authorization #: __________
Has the software registration card been returned? ______

Computer Used: ______________ CPU (8080/8085/Z-80): __________

Disk Capacity: ________ Number of Drives: ____ Memory Size: ________

Operating System/Version (If not listed above): __________/__________

Software used with the above product, (e.g. list the BASIC used if you are reporting a problem with a Payroll program that uses it).

Name of Software	Version
______________________________	__________
______________________________	__________

Does the software come with sample or test programs? ______
If so, have you been able to use them successfully? ______

Please describe the problem you have encountered. Include references to the manual if appropriate. Try to reduce the problem to a simple test case. Enclose any appropriate programs (preferably on disk). If you feel that the problem may be caused by the disk being defective, you may prefer to return the original disk with this report to achieve the fastest resolution of the problem. (If so, call for a Return Authorization No. A handling charge may be incurred. No handling charge will be made if a product or portion thereof is returned **DUE TO DISKETTE MEDIA DEFECTS** within 30 days from the date of sale).

Information on product changes, bugs, fixes and current version numbers are published in **Lifelines**, our software newsletter.

PROBLEM DESCRIPTION: (Continue on additional pages if necessary)

Area Phone Num. Ext.

Name: ______________________________ (____) ___-_____(____)

Address: ______________________________ (____) ___-_____(____)

City: ______________ State: ________ Zip Code: ________

Return to: Lifeboat Associates
1651 Third Avenue
New York, N.Y., 10028

Technical assistance is available Monday - Friday, from 11:00 a.m. to 7:00 p.m., Eastern time.
1-(212) 860-0300
TWX: 710-581-2524 Telex: 640693

002prob.bn.09.81