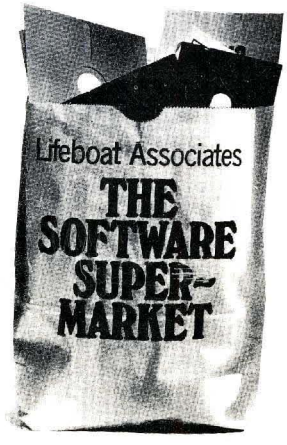
MANUAL

The BD Software C Compiler v1.4



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 Compiler v1.4
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

BD Software C Compiler v1.4
User's Guide

Leor Zolman
BD Software
33 Lothrop st.
Brighton, Mass. 02135

Introduction

I'm not even going to *bother* comparing C to BASIC or FORTRAN.

So, left with a few paragraphs to fill with an introduction, allow me to explain why this software package is so inexpensive:

Before a selling price is set for a program in the microcomputer systems environment, the seller must decide whether or not large-scale ripoffs are be expected. For a $300 BASIC interpreter, yes, one might expect ripoffs, so the price is deemed "justifiable" by the vendors to insure an acceptable profit margin or "discourage" ripoffs (?).

Hmmphh.

As far as BDS C is concerned, the price was set assuming there will *not* be any ripping off, since I feel (as I have been advised numerous times) that the compiler is really worth more than its selling price. The last few years, though, have seen a proliferation of prohibitively expensive quality software, and that fact (along with the realization that if *I* were shopping for a compiler like C, *I* would possibly copy it from a friend if it were priced any higher) has held the price down to a reasonable level.

There are no licenses or royalty agreements connected with this package, aside from the standard agreement that the package be used on one system only (which each user implicitly agrees to in the act of unsealing the diskette envelope.) Thus, 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 the burden of having to pay BD Software any royalties. The whole idea behind this policy is to encourage potential software vendors to use C for their development work, and then perhaps to include source listings of their code with their packages and thereby promote the use of C.

Lifeboat Associates are the *exclusive* distributors of the BDS C package for CP/M systems. The disk you've received is legitimate *only* if it has a Lifeboat label (with the shopping bag) affixed to it, and on that label is a description of the package (made by a hand stamp) with the serial number filled in. No matter where you bought your disk from, it should have originated at Lifeboat; if you have any suspicions that the disk you've paid for might be a bootleg, *please* contact either myself or Lifeboat about it immediately so we can put an end to such treachery.

Remember: If you rip C off or give it away, you will *not* be robbing some big corporation; you'll be screwing an individual programmer who's trying to market some useful software at a reasonable price and still remain solvent.

Objectives and Limitations

The BDS C Compiler is the implementation of a healthy subset of the C Programming Language developed at Bell Laboratories.[1] The compiler itself runs on 8080/Z80 microcomputer systems equipped with the CP/M[2] 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, from the minicomputer to the microcomputer environment, a bit of the powerful, structured programming philosophy on which the Unix[3] operating system is based. BDS C provides a friendly environment in which to develop CP/M utility applications, with an emphasis on elegant human interfacing for both compiler use and operation of the end-applications.

Unfortunately, the lexical oddities of C's linguistic structure do not conform as readily to the 8080's hardware characteristics as they do to the PDP-11's.[4] Operations natural to the 11 (such as indexed-indirect addressing--a crucial necessity when dealing with automatic local 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 11; but then, as better microprocessors soon replace the 8 bit machines, you can bet there will be C compilers available that generate code efficient enough to resign assembly language programming to the history books. Consider this package a warm-up to that era...

BDS C's big tradeoff (when compared to assembly language programming) is a loss of object code efficiency (both spatial and temporal), at run-time, in favor of a high degree of structure and comprehensibility at the development stage. In education, as well as in other non time-critical applications (such as non-gargantuan systems programming), I believe the sacrifices are rather minimal in contrast to the benefits.

New Features of V1.4: A Summary for Users of Earlier Versions

There has been a hefty amount of revision, expansion and clean-up applied to the package since the last release (v1.3x). 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. BDS C version 1 has just about saturated its framework; version 2 is now being developed in close conjunction with the MARC Disk Operating System (the work of Edwin P. Ziemba) to provide a unified software development system for release sometime in 1981. MARC is a "Unix-like" operating system that happens to fit quite comfortably in non-gargantuan 8080/Z80-based machines. MARC and BDS C should get along nicely, and the price for the combined package ought to prove tempting...but

---

1. See The C Programming Language by Brian W. Kernighan and Dennis Ritchie (Prentice Hall, 1978) for a proper 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.
2. CP/M is a trademark of Digital Research, Inc.
3. Unix is a trademark of Bell Laboratories.
4. PDP is a trademark of Digital Equipment Corporation.

this section is supposed describe new features of *this* software package, so here goes:

The assembly language sources for the BDS C run-time package (CCC.ASM --> C.CCC) and all non-C-coded library functions (DEFF2.ASM --> 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 that allows both the code origin and r/w memory data area to be specified explicitly, so 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 MAC (Digital Research's macro assembler) and the nifty new macro package (CMAC.LIB) included with BDS C as standard equipment (sorry, MAC isn't.)

On a higher level, the buffered I/O library can now be trivially customized to use any number of sectors for internal disk buffering (older versions were limited to one sector of buffering unless a special function package called BIGFIO.C was used; BIGFIO.C is no longer necessary.) A new general purpose 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'.

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 hog up space; if you DO need it, then you simply add a few special statements to your program and specify DIO.CRL at linkage time, then use the standard redirection syntax on the CP/M command line.

Documentation on all the miscellaneous new library functions has finally found its way into the User's Guide, and the Function Summary section now goes into a little more detail on some of the confusing aspects of the file I/O mechanism.

On the technical side, version 1.4 employs a single run-time stack configuration instead of the two-stack horror used in previous releases. All function parameters are now passed *on the stack*, and all 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.

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 compiler option **-e** *xxxx*. This option to CC1 allows an absolute address for the external data area to be specified at compile time, thus enabling the compiler to generate absolute loads and stores (using the **lhld** and **shld** 8080 ops) for external variables.

Incompatibilities With Earlier Versions

Since the run-time package has been totally reorganized since the last release, 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 1.4, and old CRL files should be thrown away. There are also a few source incom-

patibilites that require a bit of massaging to be done to old source files. These are:

0. 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.

1. 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];
```

2. Comments now *nest*; i.e., for each and every "begin comment" construct ("/*") there must be a matching "close comment" ("*/") 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 a "/*" at the start of the line; instead, a good practice would be to insert a "/*" above the line to be commented out, and insert a "*/" following the line. Although this is something that UNIX C expressly disallows, 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; formerly, any comments *within* such a block of code had to be removed first.

In version 1.4, the run-time package comes assembled 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 the "CRL Format" section for details on customizing the run-time package.

System Requirements

The practical minimum system configuration required by BDS C is a 32K CP/M environment. Most sample programs included in the package will compile (without segmentation) and run on a 48K system.

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 be performed quickly; 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 CP/M can handle them. The drawback to this scheme is that a source file must fit entirely into memory for the compilation. This may sound bad at first, but it isn't really. Consider: a *program* in C is actually a collection of many smaller *functions*, tied together by a **main** function. Each

function is treated as an independent entity by the compiler, and may be compiled separately from the other functions in a program. Thus a single program may be spread out over many source files, each containing a number of functions; breaking files up this way serves to minimize re-compilation time following minor changes as well as keep the individual source files small enough to fit in memory.

Using the Compiler

The main BDS C package consists of four executable commands:

CC1.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

CC1.COM and CC2.COM together form the actual compiler. CC1 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.[1] 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 (and DEFF.CRL and DEFF2.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.

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

For example, here is the sequence required for compiling and linking a source file named *foo.c*:

The compiler is invoked with the command:

```
A>cc1 foo.c <cr>
```

After printing its sign-on message, CC1 will read in the file *foo.c* from disk and

1. If desired, the intermediate file produced by CC1 may be written to disk and processed by CC2 separately; then, the intermediate file is given the extension **.CCI**.

crunch for a while. If there are no errors, CC1 will then give a memory usage diagnostic and load in CC2. 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>

Following are the detailed command syntax descriptions:

CC1 -- The Parser

Command format: **CC1** *name.ext* [options] <cr>

Any name and extension are acceptable, provided the file having the exact given name exists. By convention, the extension *should* be "**.c**". If the extension is omitted, CC1 will **not** automatically tack on a default extension for you. The extension (if required) must be stated explicitly.

If a disk designation is given for the filename (e.g. "b:foo.c") then the source file is assumed to reside on the specified disk, and the output also goes to that same disk.

Typing a control-C during compilation will abort the compilation and return to CP/M.

Following the source file name may appear a list of option characters, each preceded by a dash. 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.

-a x Auto-loads CC2.COM from disk *x* following successful completion of CC1's processing. By default, CC2 is assumed to reside on the currently logged-in disk. 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.

-d x Causes the CRL output of the compiler to be written to disk *x* if no errors occur during CC1 or CC2. If the **-a z** option is also specified, then this option specifies which disk the **.CCI** file is to be written to. 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.
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 CC1 for this option 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. If this option is given, then the compiler can generate code to access external data directly (using lhld, shld, etc. type 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, possibly eating into the explicitly specified external data area. **CC2 must be successfully auto-loaded by CC1 in order for this option to have any effect.** See also the CLINK option **-e** for more confusing details.

-o

Causes the generated code to be optimized for speed. Normally, the code generator replaces some awkward code sequences with calls to special subroutines in the run-time package; while this reduces the size of the code, it also slows it down because of the extra subroutine linkage overhead. If the **-o** option is specified, then many of the subroutine calls are disposed of in favor of in-line code. This results in faster but longer object programs. For the fastest possible code, the **-e** option should also be used. If you want the code to be as *short* as possible, use the **-e** option but don't use **-o**.
CC2 must be successfully auto-loaded by CC1 in order for this option to have any effect.

-r x Reserve *x*K bytes for the symbol table. If an "Out of symbol table space" error occurs, this option may be used to increase the amount of space allocated for the symbol table. Alternatively, if you draw an "Out of memory" error then **-r** may be used to decrease the symbol table size and provide more room for source text. A better recourse after running out of memory would be to break the source file up into smaller chunks, though. The default symbol table size is 8K for 0000h-based CP/M systems and 7K for 4200h-based systems.

-c Disables the "comment nesting" feature, causing comments to be treated in the same way as by UNIX C and previous version of BDS C; i.e., when **-c** is given, then a line such as

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

is considered a *complete* comment. If **-c** is *not* used, then the compiler would expect another '*/' sequence before the 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 CC1, the compilation process will abort immediately instead of loading in the second phase (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. THEN worry.

Examples:

```
A>cc1 foobar.c -r10 -ab <cr>
```

invokes CC1 on the file *foobar.c*, setting symbol table size to 10K bytes. CC2.COM is auto-loaded from disk B.

```
A>cc1 c:belle.c -p -o <cr>
```

invokes CC1 on the file *belle.c*, from disk C. The text is printed on the console (with line numbers) following **#define** and **#include** processing, CC2.COM is auto-loaded from the currently logged disk (unless CC1 finds errors) and the resulting code is optimized for speed.

See the BDS C handbook (either printed or contained in the disk file C.DOC) for more examples.

CC2 -- The Code Generator

Command format: **CC2** *name* <cr>

Normally CC2.COM is loaded up automatically by CC1 and this command need not be given. If given explicitly, then the file *name*.**CCI** will be loaded into memory and crunched upon.

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 CC1, a disk designation on the filename causes the specified disk to be used for input and output.

When CC1 auto-loads CC2, several bytes within CC2 are set according to the options given on the CC1 command line. If CC2 is invoked explicitly (i.e., not auto-loaded by CC1) 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 is the configuration of data values that need to be set (addresses are for 0-based CP/M; add 4200h for the modified versions):

Addr	default	option	function
0103	00	-a	Non-zero if CC2 has been auto-loaded, else zero
0104	01	-o	Zero if -o option (optimize for speed) desired, else 01
0105-6	0100h	-m	Origin address of C.CCC at object run-time
0107-8	none	-e	Explicit external starting address (if -e given to CC1)
0109	00	-e	Non-zero if an explicit external data address is specified

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.)

At any time during execution, if a control-C typed on the console input then compilation will abort and control will return to CP/M.

Example:

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

CLINK -- The Linker

Command format: **CLINK** *name* [other names and options] <cr>

The file *name*.CRL must contain a **main** function; *name*.CRL along with any other CRL files given will be searched (from left to right, in order of appearance) in an attempt to resolve all function references. After all given files have been searched, DEFF.CRL and DEFF2.CRL (the standard library files) will be searched automatically.

By default, CLINK assumes all CRL files reside on the currently logged in disk. If a disk designation is specified for the main filename, then *that* disk becomes the default

for all CRL files given on the command line. Each additional CRL file may contain a disk designation to override the default.

Should any unresolved references remain after all given CRL files have been searched, CLINK will enter an interactive mode, and you will be given the opportunity to specify other CRL files, re-scan the old ones, and see what functions are still missing.

Note that if there is much cross-referencing between files (not a good practice) then it may be necessary to re-scan some files several times before all references are resolved.

Control-C may be typed during execution to abort the linkage and return to CP/M.

Intermixed with the list of file names to search may be certain linkage options, preceded by dashes. The currently implemented options are:

-s Print out a statistics summary and load map to the console.

-f *file_name* (New for v1.44) Force the linking of each and every function in the file *file_name*.**CRL** into the program, regardless of whether or not the functions have yet been referenced from a higher level. This option is useful for specifying .CRL files containing alternate versions of some of the standard BDS C library functions, such as "putchar" and "getchar".

If a function in *file_name*.**CRL** has already been loaded from a previous CRL file, then a message will be printed to that effect and the new version of the function will be not be used.

-t xxxx Set start of reserved memory to xxxx (hex). The value xxxx becomes the operand of an **lxi sp** instruction at the start of the generated COM file.[1] Under CP/M, the value should be large enough to allow all program code, local, and external variable storage needed to fit below it in memory at run-time. If you are generating code to run in ROM, then the highest address of the read/write memory area *plus one* should be given here.

-e xxxx Forces beginning of external data area to be set to the value xxxx (hex). Normally (under CP/M) 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* or *execl*) to make sure that the new command's notion of where the external data begins is the same as the

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

```
lhld base+6   ;where "base" is either 0000 or 4200h
sphl
```

old one'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 use the *maximum* of all those addresses as the operand of the **-e** option for all files when you CLINK them again. You'll have to CLINK all the files twice, except for the file 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** option specified to CC1, then CLINK will automatically know about the address then specified on the CC1 command line; but if any of the other CRL files specified in the linkage contain functions compiled by CC1 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. You *may* include CRL files that were compiled by CC1 without use of the **-e** option *only if you specify* **-e** to CLINK with an argument equal to that used to compile the main CRL file.

-o new_name — Causes the COM file output to be named *new_name*.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 instead of a name, then the COM file is written to the specified disk without affecting the name of the file.

-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 defined 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 therein 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 swappable segments, the **-y** option

should be 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 as many levels down as required: while linking a swappable 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 "swapped-in" segment can do some "swapping-in" of its own, etc. etc. 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, the new symbol will be ignored and a message will be displayed on the console to that effect.

If any of the symbols in the symbol file have already been defined, 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 User's Guide appendix on the subject of overlays.

-l xxxx — Specifies 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 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*. 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 CC1.

-c x — Instructs CLINK to obtain DEFF.CRL, DEFF2.CRL and C.CCC from disk *x*. By default, the currently logged disk is assumed to contain these files.

-d ["args"] — To aid debugging, this option causes the COM file pro-

duced by the linkage to be immediately executed (instead of being written to disk.) If a list of arguments is specified (enclosed in quotes), then the effect is as if the COM file were invoked from the CCP with the given command line options. This option must 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.)

-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 foobar -s -t6000 -o lucinda <cr>

expects the file FOOBAR.CRL to contain a **main** function, which is then linked with any other needed functions from FOOBAR.CRL and DEFF*.CRL. A statistics summary is printed out when finished, memory at 0x6000 and above is to be untouched by the COM file when running, and the COM file itself is to be named LUCINDA.COM. All disk I/O during linkage is performed on the currently logged-in disk.

A>clink b:ronni lori c:adrienne -s <cr>

takes the "main" function from RONNI.CRL (on disk B), links in any needed functions from RONNI.CRL and LORI.CRL (on disk B), ADRIENNE.CRL (on C) and DEFF.CRL and DEFF2.CRL (on the currently logged in disk), 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 RONNI.COM by default (since no **-o** option was given) and the file is written to the currently logged in disk.

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 so because the linker uses the number of bytes declared for externals in the main source file as the allotment of external space for the resultant COM file. Also, because external variables in BDS C are actually more like FORTRAN COMMON than UNIX C externals, the ordering of external declarations within each individual source file of a program is very important. See the section entitled "Notes to Appendix A..." for more details.

CLIB -- The C Librarian

Command format: **CLIB** <cr>

The CLIB program is provided to facilitate the manipulation of CRL file contents. CLIB allows you to transfer functions between CRL files; rename, delete, and inspect

individual functions; create CRL files; and check out CRL file statistics.

Before delving into CLIB operation, it would be 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. Along with each function comes a list of "relocation parameters" for use by CLINK at linkage time. Also stored with each function are the names of all functions called by the given function. Collectively, the code, relocation list, and needed functions list make up a *function module.*

The first four sectors of a CRL file make up the *directory* for that file. In the directory is a list of all function modules appearing in the file, and their locations 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 the size of a single CP/M extent (16K).

For more detailed information about CRL files, see the section entitled "Adapting 8080 Machine Code Subroutines to the CRL File Format."

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 diddling, say

***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 now be specified for the filename, making the old **s** command obsolete (previous versions allowed only one disk to be used at a time, with the **s** command selecting the disk to be worked with.)

To *close* a file, 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**.

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

When a CRL file is opened, a copy of the file's *directory* (first 4 sectors) is loaded into RAM. Any alterations made to the file (via the use of the *append, transfer, rename,* and *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. The *function name* may include the special characters * and ? if an ambiguous name is desired. All functions matching the ambiguous name will be transferred (except for the "main" function, which can never 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 RAM). CLIB is then made aware that the function buffer is occupied. 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 *transfer* or *"extract-append"* as described above. *CLIB will not allow you to create a CRL file if another CRL file already exists by the same name.*

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 the associated directory space in the associated CRL file until that file is **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>

All 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, 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, 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.

To exit back to CP/M, give the *quit* command with no arguments, or type control-C.

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 v1.3
Function buffer size = xxxxx bytes

*open 0 deff

*make b:new

*open 1 b:new

*transfer 0 1 p*
```

```
*close 1

*quit

A>
```

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 CC1 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 look like:

```
CC1 $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*.)

Strangenesses

1) When using PIP to move CRL files and C.CCC around between disks, make sure to specify the **[o]** option so that PIP doesn't abort the operation upon encountering the first 0x1a byte in the file. This may not be necessary on newer versions of PIP, but if part of your file disappears after a PIP transfer, at least you'll know what to do.

2) 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.

The .CRL Function Format and Other Low-Level Mechanisms

Introduction

This section is addressed toward assembly/machine language programmers needing the ability to link in machine code subroutines together with normally compiled C functions. It describes the CRL format and how to transform a machine language subroutine into the format appropriate for .CRL files, so that the subroutine can be treated just like any other function by the C Linker. Also described are the calling conventions for function linkage and some utility routines available to assembly programmers in the run-time package.

Included with version 1.4 of BDS C is a macro library called CMAC.LIB, for use with Digital Research's MAC macro assembler. This library greatly simplifies the conversion of assembly language subroutines into CRL functions.

With CMAC.LIB, creating a CRL file from any given assembly source routine is as simple as adding a few pseudo-ops, assembling, loading, and changing the COM extension to CRL.

Although it is not absolutely necessary to know how a CRL file is organized in order to effectively use the macro package and MAC to produce CRL files, a detailed description of the CRL format is in order for general information and for the benefit of users lacking MAC. So here goes...

CRL Directories

The first four sectors of a CRL file[1] make up the *directory*. Each function module in the file has a corresponding entry in the directory, consisting of the module's name (up to eight characters [upper-case only to work correctly with CLIB in versions before 1.2] 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.[2]

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 function module to make the next module's address line up on an even (say, 16 byte)

---

1. Locations 0x100 - 0x2ff (using C's notation for hexadecimal values) in memory if you are **ddt**-ing the file.
2. The function module addresses within a CRL file are all relative to 0x0000, and the directory resides from 0x0000 to 0x01ff. The lowest possible function module address is 0x205 (locations 0x200 - 0x204 are reserved.) When using **ddt** to examine a CRL file, remember that all addresses must be offset by 0x0100 (or 0x4300 for "modified" CP/M.) For example, if the directory lists a particular function module as beginning at address 0x15cf, then you'd look at memory location 0x16cf (or 0x58cf) to see it.

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:[1]

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		

In some early version of the compiler, the word **main** was recognized as a keyword, and converted into a one-byte code having the value 0x9D. Thus, instead of seeing the sequence "MAIN" (with the N's high order bit set) in old .CRL files, you'd just see the 0x9d byte and an address. The new linker and librarian can both still handle that strange case, but the new compiler doesn't put out 0x9D's for "MAIN" anymore.

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 CC1 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. In CRL files which do not contain a **main** function, these bytes are unused.

Function Modules

Each *function module* within a CRL file is an independent entity, containing (in addition to the binary machine-code image of the function itself) a set of relocation

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

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

List of Needed Functions

If the function you are building calls other CRL functions, then a list of those function's names must be the first item in the module. The format is simply a contiguous list of upper-case-only names, with bit 7 high on the last character of each name. A zero byte terminates the list. A null list is just a single zero byte.

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

```
47 45 54 43 48 41 d2 50 55 54 43 48 41 d2 53 45 54 4d 45 cd  00
g  e  t  c  h  a  r' p  u  t  c  h  a  r' s  e  t  m  e  m' (end)
```

Length of Body

Next comes a 2-byte value specifying the exact length (in bytes) of the *body* (to be defined next.)

Body

The *body* portion of a function module contains the actual 8080 code for the function, with origin 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 *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>.
```

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 all instructions within the body which reference a local address. CLINK takes every word being pointed to by an entry in this list, and adds a constant to it which equals the value of the address where the first byte of the function ends up residing in the resultant COM file.

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 $2n + 2$ bytes.

For example, a function which contains four local jump instructions (which begin, respectively, at locations 0x22, 0x34, 0x4f and 0x61) would have a relocation list looking like

04 00 23 00 35 00 50 00 62 00.[1]

Calling Conventions and Register Allocation

All argument passing on function invokation, as well as all local (automatic) storage allocation, now take place on a single stack at run time. The stack pointer is kept in the SP register, and is initialized to the very top of the CP/M TPA in the standard configuration (or to the value specified as argument to -t at linkage time.) 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.

When a C-generated function receives control, it will usually: 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.[2] Note that the old value of BC must always be preserved for the calling routine.

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

(BC) + *loffset*

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

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

---

1. Note that the addresses of the instructions must be incremented by one to point to the actual address operands needing relocation.
2. 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 address calculation hopelessly confusing for poor lazy compiler hackers like me.

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 between the current SP and the first formal parameter (two 16-bit values: the saved BC, and the return address to the calling routine.) Note that this scheme presupposes that each formal parameter takes exactly 2 bytes of storage. When 4-byte variables come into play, the general formula falls apart and the location of each parameter will depend on the types of the other parameters. But let's leave something for version 2...

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 to the state it was in 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.

Formally, the responsibilities of a 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 can be considered "safe" from alteration by the subordinate function; by convention, the function that is called must always preserve the BC register value that was passed to it. All functions produced by the compiler do this.

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. 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 in the HL register.

How Much Space Does the Stack Take Up?

The new single stack scheme has all local (automatic) data storage, formal parameters, return addresses and intermediate expression values living on the one stack up in high memory. Usually the stack pointer is initialized to the very top of memory (the BDOS area) and grows down from there (the -t option to CLINK may be used to override that default.) 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. 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 new 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. *Endext()* is thus the first byte of memory available to the user.

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.

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 indirection to get
inx h           ;value into HL
mov h,m
mov l,a
```

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

```
call sdei
```

```
        db eoffset      ;if 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.

The second class of subroutines are used primarily for fetching the value of a function argument off the stack into HL and A. 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 first, then you'd have to call *ma2toh* in order to fetch the *first* argument, *ma3toh* to fetch the second, and so on for *ma4toh* and the rest.

Another way to deal with function arguments is to call the routine called *arghak* as the *very first thing* you do in your function (even before pushing BC.) *Arghak* copies the first seven function arguments off the stack to a contiguous 14-byte area 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...assuming your function doesn't call others which copy *their* arguments down there. After *arghak* has been called, the first argument will be stored at absolute location *arg1*, the second at *arg2*, etc.

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.

Again, 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 assembly language source files should contain the MAC directive

```
        maclib bds
```

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.

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 TPA (100h or 4300h), in response to a direct command to the Console Command Processor. 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 (commenc-

ing 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) reassemble 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 toward such ends:

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 much CP/M-related support code and reduce the size of both the run-time package and the required r/w memory area; non-CP/M operation will also cause the CP/M-dependent entry points within the run-time package to remain undefined, so you won't accidentally generate code to use them while developing assembly functions. 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.

   After the binary image of CCC.ASM is produced (be it named CCC.COM or whatever), rename it to be: C.CCC.

   Note: 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, then re-boot CP/M and use the "save" command to write the new C.CCC back to disk in binary form.

2. 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.

3. Using MAC, assemble the machine language library routine file (DEFF2.ASM), load it, and rename it DEFF2.CRL. If any functions in DEFF2A.ASM are needed, then assemble that file also, rename it DEFF2A.CRL, and use CLIB to transfer everything in there over to DEFF2.CRL. If you are configuring the system for a non-CP/M environment, you'll have to purge all the CP/M-related functions from DEFF2.ASM and DEFF2A.ASM before assembly. See the comments in CMAC.LIB for instructions on the use of the special pseudo-ops for creating CRL files with MAC.

4. When using CC1 to compile code for a non standard (base-of-TPA) load address, specify the **-m** option to inform the compiler of the new run-time package origin address. Make sure to re-compile STDLIB1.C and STDLIB2.C using **-m**, and use CLIB to create a new DEFF.CRL composed of everything from STDLIB1.CRL and STDLIB2.CRL.

5. 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.

6. Burn the PROMs!

Debugging Hint

Use of the **-o** option to CC1 will make interactive debugging of the generated code (using, say, SID) easier, since this will avoid the in-line data bytes that usually follow value fetching calls to the run time package.

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

Included in the BDS C package are the files DEFF.CRL and DEFF2.CRL, making up the standard library.[1] These files contain a collection of useful C functions, in CRL (C ReLocatable) format, available for use by all C programs. CLINK automatically searches the library *after* all other CRL files given on the command line have been searched once; 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.

CLINK begins its task by loading in the **main** function from the CRL file specified as the first argument on the command line. If **main** calls any other functions (it usually does), then each such function is searched for in the first CRL file, loaded if found, and recursively examined for any functions *it* may need. If there are still more functions needed after loading everything that was needed from the first CRL file, then the other CRL files on the command line (and finally DEFF.CRL and DEFF2.CRL) are scanned. Because CLINK never yanks up a function unless some previously loaded function has made a reference to it (or the **-f** option is used), you may have to go back and re-scan some files after the first pass has been completed. This only happens when a function defined in one of the first CRL files isn't used at all until a function in a *later* file calls it. By avoiding this type of backward-reference, the need for re-scanning may be eliminated.

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 C-type notation intended to illustrate how a *definition* of that function would 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 value are really used and a value of type **char** would work just as well.

The only time it is necessary to actually *declare* a library function before it is used in a C program is when the function returns a value having a type other than **int**, and that value is used immediately in an expression where the type has some significance. A bit of experience will help to clarify when it is proper or unnecessary to declare cer-

---

1. For version 1.4, DEFF2.CRL contains all the assembly language functions from DEFF2.ASM and DEFF2A.ASM (assembled using MAC, CMAC.LIB and BDS.LIB), while DEFF.CRL contains all the C-coded functions from STDLIB1.C and STDLIB2.C.

tain 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:

I. GENERAL PURPOSE FUNCTIONS

1. char csw()

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

2. 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.

3. int bdos(c,de)

Calls location RAM+5 (where RAM = 0x0000 for 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 A and B (low-order 8 bits in A, high-order 8 bits in B.) For CP/M 2.x, this is the same as the value returned in HL.

4. 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. 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.
Return value is the value returned in A by the BIOS call.
There are some BIOS calls that require a parameter to be passed in DE, and that return their result in HL. Note that a special version of *bios* that supports this format, call it *biosh*, may easily be written in terms of the *call* function by noting that memory locations 1 and 2 (or 4201h and 4202h) contain the address of the second entry in the BIOS jump vector table.

5. 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.

6. 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.)

7. inp(n)

Returns the eight-bit value present at input port *n*.

8. outp(n,b)

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

9. 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* input; before *pause* can be used again, a *getchar()* call must be done to clear the status.
There is no return value.

10. sleep(n)

Sleeps (idles) for n/10 seconds (on an 8080). The only way to abort out of this before it wakes up is to type control-C, which reboots CP/M.
No return value.

11. 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 HL.
The subroutine must, of course, maintain stack discipline.

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

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

13. int abs(n)

Returns absolute value of *n*.

14. int max(n1,n2)

Returns the greater of two integer values.

15. int min(n1,n2)

Returns the lesser of two integer values.

16. srand(n)

Initializes pseudo-random number generator.
If *n* is zero, then *srand* asks the user to type a carriage return and starts to count, internally. When a key is finally hit by the user, the current value of the count is used to initialize the random seed.
If *n* is non-zero, then *n* itself is used as the seed.

17. srand1(string)
    char *string;

Like *srand*(0), except that the given string is printed as a prompt instead of the canned "Hit return after a few seconds:" message. Unlike *srand*, though, the character typed is **not** gobbled up; you must do a *getchar* to clear it.

18. int rand()

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

```
rand() % n
```

19. 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 *srand1* 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 *srand1* function: the string pointed to by *prompt_string* is printed on the console, and then the machine waits for the user to type a character while constantly incrementing an internal 16-bit counter. As soon as a character is typed, the value of the counter is plastered throughout the 48-bit seed. Note that the console input is *not* cleared; a subsequent *getchar()* call is required to actually sample the character typed.
The final form simply returns the next value in the random sequence, with the range being

0 < *nrand*(1) < 32768.

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

20. setmem(addr,count,byte)

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

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

Moves a block of memory *count* bytes in length from *source* to *dest*. This new version 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.

22. 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. See the *OTHELLO.C* program for a good example of using *qsort*.

23. 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. A string constant (such as "foo") is perfectly reasonable, since it evaluates to a pointer.
If the command to be executed was generated by the C compiler, then it should have been linked with the CLINK option **-e** specified if external variables need to be shared between the *exec*ing and *exec*ed files. See the CLINK documentation for details on the proper usage of this 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...to allow this, the external data starting address must be made the same for all files involved, using the CLINK option **-e**.

Returns -1 on error...but then, if it returns at all there must have been an error.

24. 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 CCP of CP/M. For example, the call

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

would have the same effect as if the command

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

were given to CP/M from the console. 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 80 characters.
-1 returned on error (again, though, if it returns at all then there must have been an error.)

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

Similar to *execl*, except that the argument texts must be placed into an array instead of specified explicitly in the calling sequence. 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 one is NULL. This mechanism allows chaining with a variable number of arguments to be performed.
If the program *filename*.**COM** is not found, then the message "Broken Pipe" will be printed on the console and control will return to CP/M.

26. 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 would normally be used to load in an overlay segment for later execution via an indirection on a pointer-to-function variable; it may be used to load in any type of file, though.
Returns -1 if there is an error in reading in the file. Control is *not* transferred to the loaded file.

27. 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 (see *externs()* below.)

28. char *externs()

Returns a pointer to the start of the external data area. Unless the -e option was used with CC1 and/or with CLINK, this value will be the same as *codend()*.

29. char *endext()

Returns a pointer to the first byte following the end of the external data area.

30. char *topofmem()

Returns a pointer to the last byte of the TPA (this is normally the top of the stack.) The value returned by *topofmem()* is *not* affected by use of the -t option at linkage time.

31. 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, slightly simplified for the case where type-allignment restrictions are nonexistent. See the book for details.
Note that the

```
#define ALLOC_ON 1
```

statement in the header file BDSCIO.H must be un-commented (enabled) and STDLIB1.C re-compiled to allow use of *alloc* and *free*. See the comments in BDSCIO.H for more details on this process.
BDSCIO.H must be **#included** in *all files* of a program that uses the alloc-free pair, since there is some crucial external data declared therein. Your best bet would be to put an

```
#include "bdscio.h"
```

statement at the start of the global (.H) header file that contains all your external declarations.

32. 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 *alloc-free* pair maintain a linked list of data structures and can tolerate any order of allocation/de-allocation.
Calling *free* with an argument not previously obtained by a call to *alloc* can do miserable things to your system.
See *alloc()* above.

33. 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 *alloc()* and *free()* 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.)

34. rsvstk(n)

This should be used before any calls to *sbrk* or *alloc*, so that the storage allocation functions 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.)
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.)

II. CHARACTER INPUT/OUTPUT

35. int getchar()

Returns next character from standard input stream (CP/M console input.)
Re-boots CP/M on control-C.
Carriage return echos CR-LF to the console output and returns the *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 -1 is to be recognized. If you declare *getchar* to be a character or assign its return value to a character variable, then the value 255 should be checked for instead (to detect the EOF character, control-Z.)

36. 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 pushed back since the last *getchar()* call, then the value of that character is returned.

37. 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.

38. putchar(c)

Writes the character *c* to the standard output (CP/M console output.)
The newline ('\n') character is transformed into a CR-LF combination.
If a control-C is detected on console **input** during a *putchar* call, program execution will halt and CP/M will be re-booted. If any other character is typed during a *putchar* call, then that character will be completely ignored.
If you don't want the console input interrogated during console output, use the *putch* function, described next:

39. putch(c)

Like *putchar*, except that the console input is NOT interrogated for control-C (or anything else) during output; any characters detected at the console input will be thrown away.

40. puts(str)
    char *str;

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

41. char *gets(str)
    char *str;

Collects a line of input from the standard input into the buffer *str*.
Returns a pointer to the beginning of *str* (the value *gets* was called with.)
The BDOS call to buffer up a line of input is used; hence, the length of the provided buffer must be at least 3 bytes longer than the longest string you ever expect entered. Caution dictates making the buffer *large*, since an overflow here would most probably destroy neighboring data.

42. 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, forcing 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 has been implemented for v1.43 of the package, and is very useful for printing hexadecimal values; the

feature had been neglected in previous versions.
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.

43. 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 field width specification is not supported, but the assignment suppression character (*) works OK.
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.

III. STRING AND CHARACTER PROCESSING

44. int isalpha(c)
    char c;

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

45. int isupper(c)
    char c;

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

46. int islower(c)
    char c;

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

47. int isdigit(c)
    char c;

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

48. int toupper(c)
    char c;

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

49. int tolower(c)
    char c;

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

50. int isspace(c)
    char c;

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

51. 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.

52. 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.

53. 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.

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

Returns:
    a positive value if s1 > s2
    zero if s1 = = s2
    a negative value if s1 < s2
(ASCII collating sequence used for comparisons)

55. 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", you'd 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. Thus, if the latter construction is preferred, then *foo* must be declared as a pointer to characters. 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), 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] = '\0';
```

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,"whatever_the_beep");
```

56. int strlen(string)
    char *string;

Returns the length of *string* (the number of characters encountered before a zero-byte is detected.)

57. 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.
Zero returned if no legal value found.

58. 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 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.

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

The character equivalent of the above. *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: UNIX C programs will sometimes assign negative values to character variables, since UNIX C character variables are *signed* 8 bit quantities.

With BDS C, negative values can only be meaningfully assigned to normal **int** variables.

60. 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?[1])
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, you're welcome to go into STDLIB.C and change the terminating value to be whatever your heart desires (you'll have to change *getval* and *initw*.)

IV. FILE I/O

There are two general categories of file I/O functions in the BDS C library. The low-level (*raw*) 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 text-line at a time. The raw functions will be described first, and the buffered functions (beginning with *fopen*) later.

Whenever a function takes a filename as an argument, that filename must be either a literal string or a pointer-to-characters that points to a legal filename (actually, a literal string *is* a pointer to characters.) Legal filenames may be upper or lower case, but there must be no white space within the string. The filename may contain a leading disk designator (single character) followed by a colon to specify a particular CP/M drive; the default is the usual currently-logged disk. 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 unprintable characters, but the mechanism still isn't entirely foolproof. Be careful when processing filenames.

---

1. Because the pointer-to-characters pointing to the text string must be *altered* by the *getval* routine; any object which is to be altered by a function must be manipulated through a *pointer* to such an object. Thus, a pointer-to-characters must be manipulated through a pointer-to-pointer-to-characters.

61. int creat(filename)
    char *filename;

Creates a (null) file with the given name, first deleting any existing file having that name. The new file is automatically opened for 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.

62. int unlink(filename)
    char *filename;

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

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

Renames the file in the obvious manner.
The file specified must *not* be open while being renamed.
This function always returns -1 for CP/M 1.4 and earlier versions of CP/M; For 2.0 and MP/M, it *should* return 0 for success and -1 only on error.

64. int open(filename,mode)
    char *filename;

Opens the specified file for input if *mode* is zero; output if *mode* is equal to 1; 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.

65. int close(fd)

Closes the file specified by the file descriptor *fd*, and frees up *fd* for use with another file. With version 1.4, disk accesses will only take place when a file that was opened for *writing* is closed; if the file being closed was only open for *reading*, then the fd is freed up but no actual CP/M call is performed to close the file.
Close does **not** do an automatic *fflush* for buffered I/O files.
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 (perhaps because there is a chance that garbage was written into it), use the *fabort* function.

66. 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 the data written into it.

67. 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 *m* blocks actually left in the file (where $0 < m < n$), then *m* would be returned on that call, 0 on the next call (provided *seek* isn't used), and then -1 on subsequent calls.

68. 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 error, or the number of records successfully written. If the retun value is non-negative

but different from *nbl*, it probably means you ran out of disk space; this should be regarded as an error.

69. int seek(fd,offset,code)

Modifies the next read/write record (sector) pointer associated with file *fd*.
If *code* is zero, then sets the r/w pointer to *offset* records.
If *code* is equal to 1, then sets the r/w pointer to its current value **plus** *offset* (*offset* may be negative.)
A return value of -1 indicates that the resulting offset was out of range for the given file (cannot seek past EOF). If this occurs, the internal data for the file usually get screwed up royally; the file should be closed (or *fabort*-ed) and re-opened before any further operations on it take place. Under CP/M, it is possible to seek without error to any point within the currently active extent (16K byte portion) of a file, but subsequent *read* or *write* operations under such circumstances may cause unpredictable results.
*Seek*s should not be performed on files open for buffered I/O.

70. 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.

71. 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. Former versions of the package used a fixed-length buffer (134 bytes, to be exact) which limited the I/O buffering to one sector at a time; the 1.4

package allows the user to customize the size of the I/O buffers by changing a **#define** statement in the BDSCIO.H file. See the comments in BDSCIO.H for more details.
The technical structure of the buffer is

```
struct _buf {
    int _fd;
    int _nleft;
    char *_nextp;
    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 all needed information is automatically maintained in the I/O buffer. Note that the new *fclose* function, for closing buffered I/O files, eliminates the need for saving the file descriptor returned by *fopen* since the *close* function need no longer be used.
-1 returned on error.

72. 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.
getc(0) is equivalent to *getchar()*.
getc(3) reads a character from the CP/M "reader" device.
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. -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 error value (-1, or ERROR) should be checked for when testing for end-of-file, since some CP/M text editors neglect to place a 0x1a byte (control-Z, CPMEOF) at the end of a text file under certain circumstances.

73. 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.

74. 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.

75. 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.

76. 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 (LF) characters a la UNIX (for increased efficiency and straightforwardness.)
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.
When writing out text to a file, be sure to terminate the text with a control-Z (0x1a, CPMEOF) byte.
The values 1, 2, and 3 may be used in place of *iobuf* with any buffered output routines to direct

the output character to the console, list device, or punch device instead of to a file.
A call to *fflush* should always be made before closing the file (*fclose* is used to close a buffered output file.)
Returns -1 on error.

77. 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.

78. int fflush(iobuf)
    struct _buf *iobuf;

Flushes output buffer *iobuf*. I.e., it makes sure that any characters that may currently be in the output buffer make it into the file on disk. *Fflush* does **not** close the file.
Note that an automatic flush takes place whenever the output buffer fills up; *fflush* need normally be called only once right before the file is closed (via *fclose*.)
Fflush is to be used only with buffered *output* files. Doing an *fflush* on an input file is both meaningless and dangerous to the integrity of the file.

79. int fclose(iobuf)
    struct _buf *iobuf;

Closes the buffered I/O file specified (it may have been opened for either reading [via *fopen*] or writing [via *fcreat*]). *If the file was opened for writing, then an fflush* call should have been performed immediately before the *fclose* call.

80. 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 at *iobuf* instead of to the console.
Returns -1 on error.

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

Like *scanf*, except that the text input is scanned from the buffered input at *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 occured in reading the file.

82. 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 and a LF at the end of 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 (Note that LF is the same as '\n'.) 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 (alias LF alias '\n') 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 is returned (the same as the passed value of *str*.)

83. 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.)

84. int setfcb(fcbaddr,filename)
    char *filename;

Initializes a CP/M file control block located at address *fcbaddr* with the null-terminated name pointed to by *filename*.
The next-record and extent-number fields of the fcb are zeroed.
If any screwy characters (the kinds 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.

85. 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.

V. PLOTTING FUNCTIONS (FOR MEMORY-MAPPED VIDEO BOARDS)

86. 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; *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. If you are using a Processor Tech VDM-1, *setplot* need not be called at all; the parameters are automatically set up for the VDM-1 as part of the start-up sequence for every C-generated COM file.

87. clrplot()

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

88. 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.

89. 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.

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

Line **only** works with a 64 by 16 board.
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*.

Notes to APPENDIX A of
The C Programming Language

(For the BDS C Compiler)

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 these notes is to document the *other* parts.

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 be ignored.

Here is a summary of the most significant ways in which BDS C differs from UNIX C:

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

2) 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.

3) The complexity of declarations is restricted by certain rules.

4) No initializers are allowed.

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

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

7) Only a bit of intelligent code optimization is performed.

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

9) BDS C is written in 8080 assembler language, *not* in C itself. If BDS C were written in itself, the compiler would be five times as long and run incredibly slower. Remember that we're dealing with 8080 code here, *not* PDP-11 code as in the original UNIX implementation.

The following is a section-by-section annotation to the **C Reference Manual.**[1] 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 resident on Intel 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. As might be expected, BDS C will also run on any machine that is upward compatible from the 8080, such as the Zilog Z-80 or Intel 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 CC1 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.[2] Thus, function names should always be written in a single case (either upper **or** lower, but not mixed) to avoid confusion. For example, the statement

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

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

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

and

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

are equivalent.

2.3 Keywords

BDS C keywords:

int	**else**
char	**for**

1. Appendix A of The C Programming Language.
2. Function names are stored internally as upper-case-only.

struct	**do**
union	**while**
unsigned	**switch**
goto	**case**
return	**default**
break	**sizeof**
continue	**begin**
if	**end**
register	

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

On terminals not supporting the left and right curly-brace characters { and }, the keywords **begin** and **end** may be used instead. Note that you *cannot* have any identifiers in your programs named either "begin" or "end".

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: a local structure member or 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 compound statements be *avoided*. Sooner or later, future releases of BDS C will have a declaration mechanism identical to UNIX C.

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.[1] The external variable mechanism in BDS C is handled much like the unnamed COMMON facility of FORTRAN. So, 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

1. 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.

c, then *all* of the variables must be declared so that **d** and **e** (from the second file) do not clash with **a** and **b** (from the first) 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.

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, say, *getc* to a character variable, then a -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. Watch it.

Most arithmetic on characters is accomplished by converting the character to a 16-bit quantity and zeroing the high-order byte. In some non-arithmetic operations, such assignment expressions, BDS C will optimize by ignoring the high order byte when dealing with character values. 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.

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.

7.5 Shift operators

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

7.11, 7.12 Logical AND and OR operators

These two operators have *equal* precedence in BDS C, making parenthesization necessary in certain cases where it wouldn't be necessary otherwise. The only excuse I can offer to compiler hackers is this: BDS C does not create a syntax tree in parsing arithmetic expressions.

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.

There are no other "adjectives" allowed:

unsigned int foo;

must be written as

unsigned foo;

8.3 Declarators

Initializers are not allowed. Thus,

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 pointers to functions returning pointers to functions returning pointers to structures of type foo.

Alas, BDS C wouldn't allow that particular declaration.

Here is what BDS C *will* allow:

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

A special kind of scalar type is a **pointer-to-function**. This 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 critters internally, pointers to pointer-to-function variables will not work correctly, although pointers to functions returning any 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 zot bar[20][8];
union mumble *bebop[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.

Lastly, it must be mentioned that explicit pointers-to-arrays are not allowed. 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 is the same in meaning as

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

On the brighter side, any formal parameter declared to be an array is internally handled as a "pointer-to-array," causing an automatic indirection to be performed whenever the appropriate 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.

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 *cannot* be the same as any regular local variable 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.
External variables are *not* automatically initialized to zero.

8.7, 8.8 Type names

Not applicable to BDS C.

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.
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 is correct since the increment is *implied* (to occur at **contin:**) rather than written explicitly.
This is merely a documentation bug in the book; 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\n");
```

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* the **case** or **default**. For example,

```
case 'x': foobar: Sat_Nite_Live = Funny;
```

is incorrect, and should be changed to

```
foobar: case 'x': Sat_Nite_Live = Funny;
```

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 causes a similarly named *external* identifier to disappear temporarily. 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 elements. 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 of the instances. The following sequence, for example, uses the identifier "cptr" in a legal 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 0115h or 4315h at run-time) contains a pointer to the base of the external variable area: all external variables are accessed by indexing off that two byte value.[1]

1. The -e *xxxx* option to CC1 may be used to locate the external variable area at ab-

The amount of space allocated for external variables is equal to the space needed by all external variables 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; the names don't necessarily have to be identical for each corresponding external variable in separate files (although naming them differently is just asking for trouble), but the types and storage requirements should certainly correspond.[1]

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

12.1 Token replacement

Only the simple text-substitution command

#define *identifier token-string*

is implemented. Parameterized **#defines** are not supported.

12.2 File Inclusion

Either quotes or angle brackets may be used to delimit the filename; both have exactly the same effect.

Although file inclusion may be nested to any reasonable depth, error reporting does not recognize more than one level of nesting. Try experimenting with the "**-p**" option of CC1, varying the level of inclusion nesting, to see exactly what happens.

12.4 Line Control

Not supported.

---

solute location *xxxx*, thereby considerably speeding up and shortening the code produced by the compiler. Even so, all the declaration constraints must still be observed.

1. Reminder: if you use the library functions *alloc* and *free*, you must include the header file "bdscio.h" with ALLOC_ON defined, and make sure that STDLIB1.C was also compiled with ALLOC_ON enabled; there are several external data objects required by *alloc*
and *free* declared within bdscio.h, and omission of these declarations within any source file having external variables would cause an undesirable data overlap.

15. Constant expressions

BDS C will simplify constant expressions at compile-time only when the constant expressions occur in one of the following places: following left square brackets, following the **case** keyword, following assignment operators, following left parentheses, and following the **return** keyword. Any constant expression not falling into one of those categories is guaranteed to *not* be simplified at compile-time.

The standard procedure for insuring the compile-time evaluation of constant expressions *when such expressions fall inside larger expressions involving variables* is to enclose the constant expressions in parentheses. Thus, statements such as

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

will not be simplified, and in general will generate more (and slower) code than the better form:

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

18.1 Expressions

The unary operators are:

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

The binary operators **&&** and **||** have *equal* precedence.
sizeof 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
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 preprocessor directives

#define *identifier token-string*
#include "filename"
#ifdef *identifier*
#ifndef *identifier*
#else
#endif
#undef *identifier*

are all now supported, but with some restrictions:

The '#' character must be in the first column of the line, and there may be no space between the '#' and the rest of the preprocessor directive name.

There is *no nesting* of conditional compilation directives allowed. I.e., after either an **#ifdef** or **#ifndef** is encountered, there must occur either an **#endif** or an **#else** *before* another **#ifdef** or **#ifndef**. Breaking this rule may not bomb the compiler, but it isn't too likely to yield the desired result, either.

#Defines may appear anywhere in the source file, their scope extending until the end of the file or until the identifier is re-**#defined**. Parameterized **#defines** are not supported.

File inclusion may nest to any depth (although mutually inclusive files may just manage to bomb CC1), but both the us "-p" option with CC1 and error reporting for

CC1 and CC2 become easier to deal with if you limit yourself to non-nested inclusion.

The Mistakes Most 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.

1) 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 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 pointed to by the variable *foo*. If *foo* has never been initialized to anything, then the 'a' byte would be placed at some totally random location in memory. The correct procedure here would have been to declare a buffer area, assign the address of that area to *foo*, and *then* use foo in the manner above. Such a sequence would appear as:

```
char buffer[50], *foo;
foo = &buffer;
...
*foo = 'a';
```

where the character 'a' is placed into the first byte at *buffer*.

2) Functions must not return pointers to their own local data! As soon as a function returns to its caller, storage that was local to that function is *deallocated* and made available to 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 string 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: either 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 make the destination string area external. Each method has advantages over the other; passing a destination area on each call allows many such re-

turned 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 local to a called function to remain valid after that function has returned!!

3) 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 as 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-squiggly brace that marks the beginning of the function body. Formal parameters which are not declared are assumed to be simple **int** values; should a formal parameter accidentally be declared within the actual function body, the compiler would correctly give a "redeclaration" error, since once 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 an **int**.

Whenever a function call is made, *copies* of the values of any formal parameters are passed to the function. All such values are 16 bits in length (at least with BDS C v1.4). This means that structures, arrays, unions, and any data type not inherently 16 bits in size cannot be copied and passed to a function; *pointers* to such data types, though, can. 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, a function beginning with the sequence

```
int arraysum(array)
int array[100];
{
        ...
}
```

may appear to take an array of 100 elements as a formal parameter. Actually, only a *pointer* to that array is passed, but the usage is the same as if it were an actual array. The big difference, though, is that if you change any element in the array here, you'll be changing that element for the calling program also, while changing a simple non-array formal parameter would *not* alter the original value from which the parameter was copied (back in the calling program.) 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 lie the

high points *and* the more confusing points of C.

Miscellaneous Notes

1) The " = " operator is used for *assignment* only. The relational operator *'is equal to'* is represented by " = = ". Be careful not to confuse them.

2) 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 { and } characters on their terminals can still use the compiler. Aesthetically, in my opinion anyway, the braces make for much more readable code than **begin** and **end** do, and should be used whenever possible.

3) Error recovery is not especially intelligent in some cases. If either CC1 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.

   Also, the line number given by CC2 in error reports is not always guaranteed to be accurate. CC1 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 CC1 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 CP/M 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. So, instead of guessing about where the proper punctuation *should* be, it aborts to let you fix the error quickly and try again.

3) The "*argc* and *argv*" mechanism for passing command line arguments to a C main program is implemented identically to its UNIX model, except for one thing: CP/M, since it never preserves the name of the .COM file executed, makes it tough to get *argv*[0] pointing to the command name itself. Thus, *argv*[0] will contain garbage. Don't use it for anything.

   Note that argc is, by convention, always positive, and equal to the number of arguments specified *plus one*. Arguments on the command line are treated as *strings* in all cases, not as values. If you need to specify string arguments containing imbedded spaces, then double quotes (e.g. "string containing spaces") may be used to delimit such arguments.

   All alphabetic characters on the command line are converted to upper case by CP/M. Thus, when scanning command options, be sure to check for upper case (or use the *tolower* function.)

4) Although initializations are not supported, a couple of convenience functions have been provided to allow initialization of 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), use *initb*.

Both of these are documented in the previous section.

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");
```

5) When using the function *getchar* under CP/M, the input character is automatically echoed to the console output as it is typed. About the only portable way to suppress this echo is to use the *bios* library function to read the console; note that this causes carriage returns to actually be returned as carriage returns instead of being converted to newlines a la *getchar*.

Also, the *getchar*, *putchar* and *ungetch* functions may only be used for *console* input and output. On UNIX, these routines are generalized since the operating system allows a user to specify that the main input to a program come from, say, a file instead of the console. This is known on UNIX as *directed I/O*. A common technique used in the book's sample programs is to scan through an input file by using *getchar*; this only works as long as the input to the program can be directed from a file. Since CP/M does not support this mechanism, all such sample programs should be rewritten using the BDS C buffered I/O functions (*fopen, getc, etc.*) instead of *getchar* and *putchar*.

The important point here is that UNIX achieves a high level of generality by assigning the standard input and standard output streams independently of their physical characteristics. A simple file copy program named *foo* written with *getchar* and *putchar* would simply echo the console input to the console output if invoked by typing

```
foo
```

but the same program would copy the file *bar* into the file *zot* if invoked with

```
foo <bar >zot.
```

To approach that level of generality with BDS C under CP/M, it should be noted that the buffered I/O functions can used for both file I/O, console I/O, and (for version 1.4) list device and reader device I/O. It still might take a little bit of extra coding effort to decide whether a user wants file I/O or console I/O, but the meaty parts of the I/O transfers can usually be coded in a general manner. Many users have asked why I haven't bothered to implement directed I/O in the run-time package, like Whitesmiths does. The reason is simple: CP/M is not UNIX. Under UNIX, the redirection is a function of the operating system, not the C compiler. I'd rather get C running on new operating systems that do support redirection (such as Ed Ziemba's MARC

DOS) than try to make up for CP/M's lack of versatility with warts-on-warts.[1]

One more note on this subject: *getchar*, upon receiving a carriage return from the console, automatically echoes a linefeed (in addition to the automatic echo of the CR) and returns a *newline* character. *Getc*, on the other hand, when used for inputting characters from a text file, does *not* change CR-LF combinations into newlines. If you'd like this to happen, write yourself a little routine (say, *getc2*) that calls *getc* and filters out CR-LFs by issuing a dummy call to *getc* following each CR encountered and returning a newline in such cases. Once this is done, the process of writing programs that are generalized to both console and file I/O should be as painless as possible under CP/M.

5a) When scanning through an input text file (using, say, *getc*), the logical-EOF character is a control-Z (0x1a). A return value of -1 from the fileread functions (*read*, *getc*, etc.) indicates a *physical* EOF (always on a block boundary) and will probably *not* coincide with the *logical* EOF (where the control-Z is.) Thus the correct algorithm for detecting the end of a text file must check for both of these possible values, and interpret the first one encountered as the EOF. Note that if you are assigning the return value of a function such as *getc* to a character variable, the the -1 physical-EOF condition value magically turns into 255 after assignment.

When writing output text files, be sure to terminate them with a control-Z in an attempt to maintain some kind of consistency; though that seems to be more than certain operating system developers have seen fit to do.

6) Unbuffered file I/O (using *open*, *read* and *write*) is done in terms of *blocks*, not *bytes*. If you wish to deal with single bytes at a time, it is necessary to use the buffered file I/O functions which, unfortunately, are slower (but not that much slower with the new user-configurable buffer size.)

On another speed note, I've found that the CP/M User's Group programs **FAST.COM** and **SPEED.COM**, written by Bob Van Valzah for 1.4 CP/M systems, do absolute *wonders* for the compilation time of all programs and the execution speed of file-I/O-bound programs. On my system, the average speed of *everything* has increased around three-fold under **SPEED**. If you've got a system that can handle these programs, but aren't taking advantage of them, you're really missing something.

7) In a high school environment, a couple of microcomputer systems running BDS C combined with copies of the book The C Programming Language for every student would provide an excellent setting for an introductory course in computer science. Teachers, take note!

8) The following tidbits should be kept in mind when striving for optimum efficiency in compiled programs:

---

1. By the way, just for the record, I **DO** like CP/M... after all, I've been hacking on it long enough to get this compiler to a respectable state. But the time has definitely arrived for a new generation of operating systems, with UNIX as the trendsetter for the time being. Onward to MARC...

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**. Of course, if values outside the character range (0-255) are expected then this information is not very useful.

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, CC1 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 CC1.

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.

   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\n");
or      if (a == 0) printf ("A is zero\n");
```

   may just as well be written as

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

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 in 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 certain cases.

9) Please report any bugs to:

Leor Zolman
~~33 Lothrop st.~~ PO BOX 9
Brighton, Massachussetts, 02135
(617) 782-0836 (evenings before 1:00 AM EST)

Please don't hassle Lifeboat with technical bug reports; they're the *publishers*, not the authors. By reporting any bugs you may encounter directly to me, you'll vastly improve the chances of having a fix for the problem in a short amount of time.

If you have any questions about the package, feel free to bug me about it (so to speak.) This gives me some idea of exactly what in the package is confusing and in need of more detailed documentation. At the time of this writing, there are approximately 1200 (legitimate) copies of BDS C out in the field, and I haven't yet been overplagued with phone calls. In fact, a vast majority of user feedback has proven very constructive. There is always the possibility, however, that sales will skyrocket and cause my phone call volume to rise to unmanageable proportions...thus I ask that questions about the compiler be mailed to the above address, if possible, instead of phoned in. If you think you've spotted a bug, though, please call, as I like to find out about bugs as soon as possible.

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

Lauren Weinstein	Sid Maxwell[1]
Leo Kenen	Bob Mathias
Rick Clemenzi	Bob Radcliffe
Tom Bell	The *Real* Cat
Jon Sieber	Al Mok
Scott Layson	Phillip Apley
Tony Gold	Charles F. Douds
Ed Ziemba	Robert Ward
Scott Guthery	Les Hancock
Earl T. Cohen	Ted Nelson
Sam Lipson	Ward Christensen
Dan MacLean	Jerry Pournelle

1. Extra thanx to Sid for, among other things, running off all my hard copy when I couldn't afford a working printer.

Mike Bentley
Carlos Christensen
Perry Hutchinson
Paul Gans
John Nall
Mark Miller
Jason Linhart
Calvin Teague
Bob Shapiro
Cal Thixton
Will Colley
Richard Greenlaw
Tim Pugh
Steve Ward
Tom Gibson
Roger Gregory
Don Lucas
Rev. Stephen L. de Plater
Nigel Harrison

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.

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

BDS C User's Group
Robert Ward, Coordinator
Dedicated Micro Systems, Inc.
409 E. Kansas
Yates Center, Kansas 66783
(316) 625-3554

Due to the large volume of assembly sources included with the 1.4 package, many of the sample C programs included with prior versions have been squeezed out of the distribution package. The BDS C User's Group will have all these programs, as should the CP/M User's Group eventually. I recommend that one of these groups be contacted and the sample programs obtained, especially if you are a novice C programmer; the language tends to be painful to pick up without lots of examples.

The CASM.C Assembly-language-to-CRL-Format Preprocessor
For BDS C v1.46
March 3, 1982

Leor Zolman
BD Software
33 Lothrop st.
Brighton, Massachussetts 02135

The files making up the CASM package are as follows:

CASM.C	Source file for CASM program
CASM.SUB	Submit file for performing entire conversion of CSM file to CRL
CASM.DOC	This file

Also needed:

ASM.COM (or MAC.COM)
DDT.COM (or SID.COM)

Description:

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 almost as much as BDS C 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 that the file properly assembles directly into CRL format. In addition, some rudimentary logic checks are performed: doubly-defined and/or undefined labels are detected and reported, and similarly-named labels in different functions are ALLOWED and converted into unique names so ASM won't complain.

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

FUNCTION <name>

Each function must begin with "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 directory of included 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. Note that for the current version of BDS C, only function names may appear in "external" lines; data names (e.g. for external variables defined in C programs) cannot be placed in "external" statements.

ENDFUNC
(or) 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>
(or) 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 custimzed 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, which would cause CASM to generate relocation parameters for those 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:

0. If a label appears on an instruction, it MUST begin in column one of the line. If a label does not begin in column one, CASM will not recognize it as a label and relocation will not be handled correctly.

1. 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.

2. INCLUDE and MACLIB only work for one level of inclusion.

3. 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.

4. 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.

5. The .HEX file produced by ASM after assembling the output of CASM cannot be converted into a binary file by using the LOAD.COM 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; this is the value to be used with the "SAVE" command.

   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 ORG to set the location counter back to the base of the TPA, and the "LOAD" command aborts with the cryptic message "INVERTED LOAD ADDRESS" when out-of-sequence data like that is encountered. Rather than require CASM to write out the directory into a new file and append the entire previous output onto the end of the directory, I require the user to have to enter a SAVE command. What the heck; you'd have to rename the file anyway if it were LOADed, right?

6. The CASM.SUB submit file may be used to perform the entire procedure of converting a .CSM file to a .CRL file. For a file named "FOO.CSM", just say:

```
submit casm foo
```

   and enter the "SAVE" command just the way says when all is done.

BDS C Standard Library Summary
v1.46 Edition -- March, 1982

Leor Zolman
BD Software
33 Lothrop st.
Brighton, Massachussetts 02135

This document contains an alphabetic summary of ALL general-purpose utiliy functions included in the BDS C package spread among several different source files. Note that there are quite a few more functions listed here than than apppear in the BDS C User's Guide; some functions were intentionally omitted from the User's Guide for portability reasons, and many others have come into existence since the last revision of the User's Guide.

The summary is organized by columns.

The first column shows the type of the result returned by the function. The second column shows the calling syntax and parameter types (if not int).

The next column shows a code naming the source file in which the function may be found; the codes are as follows:

C1	for STDLIB1.C
C2	for STDLIB2.C
D2	for DEFF2.CSM
D2A	for DEFF2A.CSM
FLT	for FLOAT.C
DIO	for DIO.C

The next column tells the page number in the BDS C User's Guide where the function is documented, if the function appears in the User's Guide at all. For any function that isn't documented in the User's Guide, there is probably documentation available in the source listing for that function (the source location is given in the preceding column.)

The final column contains references to a set of footnotes following the function list. If a function has an entry in the NOTE column, the corresponding footnote (or notes) should be examined for additional information about the function.

TYPE	FUNCTION	FILE	PAGE	NOTES
int	abs(a,b) int a,b;	C1	32	
char *	alloc(nbytes) unsigned nbytes;	C1	37	14
char *	atof(opl,s) char opl[5], *s;	FLT		1
int	atoi(str) char *str;	C1	44	
int	bdos(c,de)	D2	30	2
char	bios(n,c)	D2	30	
int	call(addr,a,h,b,d) unsigned addr;	D2	32	
char	calla(addr,a,h,b,d) unsigned addr;	D2	32	
int	close(fd)	D2	46	
	clrplot()	D2A	54	
char *	codend()	D2	36	
int	creat(filename) char *filename;	D2	46	
char	csw()	D2	30	

	dioflush()	DIO		
	dioinit(&argc,argv) int *argc; char **argv;	DIO		
char *	endext()	D2	36	
int	exec(filename) char *filename;	D2	34	3
int	execl(filename,arg1, arg2, ..., NULL) char *filename;	D2	35	3
int	execv(filename,argvector) char *filename, **argvector;	D2	35	3,16
	exit(n)	D2	30	
char *	externs()	D2	36	
	fabort(fd)	D2	47	17
char *	fcbaddr(fd)	D2	53	
int	fclose(iobuf) FILE *iobuf;	C1	51	
int	fcreat(filename, iobuf) char *filename; FILE *iobuf;	C1	50	
int	fflush(iobuf) FILE *iobuf;	C1	51	7
int	fgets(str,iobuf) char *str; FILE *iobuf;	C2	52	6,11
int	fopen(filename,iobuf) char *filename; FILE *iobuf;	C1	48	
char *	fpadd(res,op1,op2) char res[5], op1[5], op2[5];	FLT		1
int	fpcomp(op1, op2) char op1[5], op2[5];	FLT		
char *	fpdiv(res,op1,op2) char res[5],op1[5],op2[5];	FLT		1
char *	fpmult(res,op1,op2) char res[5],op1[5],op2[5];	FLT		1
int	fprintf(format, arg1, arg2, ...) char *format;	C2	51	4,9
char *	fpsub(res,op1,op2) char res[5],op1[5],op2[5];	FLT		1
int	fputs(str,iobuf) char *str; FILE *iobuf;	C2	53	6,12
	free(allocptr) unsigned allocptr;	C1	37	14
int	fscanf(iob,fmt,&arg1,&arg2,...) FILE *iob; char *fmt;	C2	52	4,10
char *	ftoa(s1,op1) char *s1; char op1[5];	FLT		
int	getc(iobuf) FILE *iobuf;	C1	49	8
int	getchar()	D2	38	20
int	getline(str,maxlen) char *str;	D2A		18
char *	gets(str) char *str;	D2	40	5
int	getval(strptr) char **strptr;	C1	45	
int	getw(iobuf) FILE *iobuf;	C1	50	
int	index(str,substr) char *str, *substr;	D2A		18
	initb(array,string) char array[], *string;	C1	44	
	initw(array,string) int array[]; char *string;	C1	44	
char	inp(port)	D2	31	
int	isalpha(c) char c;	C1	41	
int	isdigit(c) char c;	C1	42	
int	islower(c) char c;	C1	42	
int	isspace(c) char c;	C1	42	
int	isupper(c) char c;	C1	41	
char *	itoa(str, n) char *str;	FLT		
char *	itof(op1, n) char op1[5];	FLT		1
int	kbhit()	D2	39	
	line(c,x1,y1,x2,y2) char c;	D2A	54	
int	longjmp(jbuf) char jbuf[JBUFSIZE];	D2A		
int	max(n1,n2)	C1	32	
int	min(n1,n2)	C1	32	
	movmem(source,dest,count) char *source, *dest;	D2	34	
int	nrand(n [,prompt] or [,n1,n2,n3]) char * prompt;	D2	33	
int	open(filename,mode) char *filename; int mode;	D2	46	
	outp(port,val) char port, val;	D2	31	
	pause()	D2	31	
char	peek(port) char port;	D2	31	
	plot(x,y,c) char c;	D2A	54	
char	poke(addr, val) unsigned addr; char val;	D2	31	
	printf(format, arg1, arg2, ...) char *format;	C2	40	4,9
int	putc(c,iobuf) char c; FILE *iobuf;	C1	50	
	putch(c) char c;	D2	39	
	putchar(c) char c;	D2	39	20
	puts(str) char *str;	C2	40	
	putw(w,iobuf) int w; FILE *iobuf;	C1	51	

	qsort(base,nel,width,cmp) char *base; int (*cmp)();	C1	34	
int	rand()	D2	33	
unsigned	rcfsiz(fd)	D2A		
int	read(fd, buffer, nsecs) char *buffer;	D2	47	
int	rename(oldname, newname) char *oldname, *newname;	D2	46	
int	rread(fd, buffer, nsecs) char *buffer;	D2A		15
int	rseek(fd, offset, origin)	D2A		15
int	rsrec(fd)	D2A		15
	rsvstk(n)	D2	38	
int	rtell(fd)	D2A		15
int	rwrite(fd, buffer, nsecs) char *buffer;	D2A		15
char *	sbrk(nbytes)	D2	37	
int	scanf(format, &arg1, &arg2, ...) char *format;	C2	42	4,10
int	seek(fd, offset, origin)	D2		
	setfcb(fcbaddr, filename) char *filename;	D2	53	
int	setjmp(jbuf) char jbuf[JBUFSIZE];	D2A		
	setmem(addr, count, byte) char *addr; char byte;	D2	33	
	setplot(base,xsize,ysize)	D2A	53	
	sleep(ntenths)	D2	31	
	sprintf(str,format,arg1,arg2,...) char *str, *format;	C2	42	4,9
	srand(n)	D2	32	
	srand1(str) char *str;	D2	32	
int	sscanf(str,format,&arg1,&arg2,...) char *str, *format;	C2	42	10
	strcat(s1, s2) char *s1, *s2;	C1	43	
int	strcmp(s1, s2) char *s1, *s2;	C1	43	
	strcpy(s1, s2) char *s1, *s2;	C1	43	
int	strlen(str) char *str;	C1	44	
	swapin(filename,addr) char *filename; unsigned addr;	C2	36	
int	tell(fd)	D2	48	
char	tolower(c) char c;	C1	42	
char *	topofmem()	D2	36	19
char	toupper(c) char c;	C1	42	
	txtplot(string,x,y,ropt) char *string;	D2A	54	
	ungetc(c,iobuf) char c; FILE *iobuf;	C1	50	
	ungetch(c) char c;	D2	39	
	unlink(filename) char *filename;	D2	46	
int	write(fd, buffer, nsects) char *buffer;	D2	47	

NOTES:

1. This floating point function returns a pointer to a 5-byte floating point object, represented in a character array of length 5.

2. The "bdos" function returns HL equal to the value left there by the BDOS itself. Under standard CP/M, 8-bit values are returned in L with H cleared, and 16-bit values are returned in HL. Other "CP/M-like" systems do not always follow this convention, though, and the "bdos" function may take rewriting in order to work with certain system calls under systems such as "SDOS".

3. Unless an error occurs, this function should never return at all.

4. Note that all the upper-level formatted I/O functions ("printf", "fprintf", "scanf", and "fscanf") now use "_spr" and "_scn" for doing conversions. While this leads to very modularized source code, it also means that calls to "scanf" and "fscanf" must process ALL the information on a line of text if the information is not to be lost; if the format string runs out and there is still text left in the line being processed, the text will be lost (i.e., the NEXT scanf or fscanf call will NOT find it.)

An alternate version of "_spr" (the low-level output formatting function) is given in the file FLOAT.C for use with floating point numbers; see FLOAT.C for details. Since "_spr" is used by "printf", this really amounts to an alternate version of "printf."

Also note that temporary work space is declared within each of the high-level functions as a one-dimensional character array. The length limit on this array is presently set to 132 by the define MAXLINE statement in BDSCIO.H; if you intend to create longer lines through printf, fprintf, scanf, or fscanf calls, be SURE to raise this limit by changing the define statement.

5. Note that the "gets" function (which simply buffers up a line of console input at a given buffer location) terminates the line with a null byte ('\0') WITHOUT any CR or LF.

6. The conventional CP/M text format calls for each line in a file to be terminated by a carriage-return/linefeed combination. In the world of C programming, though, we like to just use a single linefeed (known as a "newline") to terminate lines. AND SO, the functions which deal with reading and writing text lines from disk files to memory and vice-versa ("fgets", "fputs") take special pains to convert CR-LF combinations into single '\n' characters when reading from disk ("fgets"), and convert '\n' characters to CR-LF combinations when writing TO disk ("fputs"). This allows the C programmer to do things in style, dealing only with a single line terminator while the text is in memory, while maintaining compat- ibility with the CP/M text format for disk files (so that, for example, a text file can be "type"d under the CCP.)

7. Remember to put out a CPMEOF (control-Z or 0x1a) byte at the end of TEXT files being written out to disk.

8. Watch out when reading in text files using "getc". While a text file is USUALLY terminated with a control-Z, it MAY NOT BE if the file ends on an even sector boundary (although respectable editors will now usually make sure the control-Z is always there.) This means that there are two possible return values from "getc" which can signal an End-of file: CPMEOF (0x1a) or ERROR (-1, or 255 if you assign it to a char variable) should the CPMEOF be missing.

9. Since the "_spr" function is used to form the output string, and then "puts" is used to actually print it out, care must be taken to avoid generating null (zero) bytes in the output, since such a byte will terminate printing of the string by puts. Thus, a statment such as:

```
printf("%c foo",'\0');
```

would not actually print anything at all.

10. The "%s" termination character has been changed from "any white space" to the character following the "%s" specification in the format string. That is, the call

```
sscanf(string, "%s:", &str);
```

would ignore leading white space (as is the case with all format conversions), and then read in ALL subsequent text (including newlines) into the buffer "str" until a COLON or null byte is encountered.

11. fgets is a little tricky due to the CP/M convention of having a carriage-return AND a linefeed character at the end of every text line. In order to make text easier to deal with from C programs, this function (fgets) automatically strips off the CR from any CR-LF combinations that come in from the file. Any CR

characters not immediately followed by a LF are left intact. The LF is included as part of the string, and is followed by a null byte. There is no limit to how long a line can be here; care should be taken to make sure the string pointer passed to fgets points to an area large enough to accept the largest expected line length (a line must be terminated by a newline (LF) character before it is considered terminated).

The value NULL, NOT EOF, is returned on end-of-file, whether it be a physical end-of-file (attempting to read past last sector of the file) OR a logical end-of-file (encountered a control-Z.)

12. The "fputs" function writes a string out to a buffered output file. The '\n' character is expanded into a CR-LF combination, in keeping with the CP/M convention. If a null ('\0') byte is encountered before a newline is encountered, then there will be NO automatic termination character appended to the line, thus allowing partial lines to be written.

13. When managing overlays, the "swapin" function may be used by the root segment to swap in overlay code segments from disk. The provided version does NOT check to make sure that the code yanked in doesn't overlap some data areas that may lie above the swapping area in memory.

14. The storage allocation routines were taken from chapter 8 of K&R, but simplified to ignore the storage allignment problem and not bother with the "morecore" hack (a call to "sbrk" under CP/M is a relatively CHEAP operation, and can be done on every call to "alloc" without degrading efficiency.) Note that compilation of "alloc" and "free" is disabled until the " define ALLOC_ON 1" statement is un-commented in the header file ("BDSCIO.H"). This is done so that the external storage required by alloc and free isn't declared unless the user actually needs the alloc and free functions.

15. The random-record file I/O functions are a direct interface to the random-record BDOS functions provided by CP/M versions 2.0 and above, but not available for pre-2.0 CP/M systems. Because of the non-portability of these functions, they have not been heavily advertised in the BDS C User's Guide (i.e., they are not mentioned at all). The "rread", "rwrite", "rseek" and "rtell" functions work just like the functions "read", "write", "seek" and "tell", respectively, except that they do things via the random-record fields of the file's FCB. The "rsrec" and "rcfsiz" function simply take a file descriptor of an open file and perform their namesake BDOS operation on the given file, but in addition they also return the value computed. Thus, "rcfsiz" may be used to quickly compute the size of a file under CP/M 2.x.

16. The "execv" function no longer prints out "Broken Pipe" upon error; instead, it has the more conventional behavior of returning -1 (ERROR) and letting the user perform diagnostics.

17. "fabort" should not be used under systems like MPM-II in which all files MUST be closed, whether they are open for input or output, in order not to run out of file descriptors and hang the system.

18. New for v1.46 (see the v1.46 documentation addenda sheet for details.)

19. Modified for v1.46 to detect when "NOBOOT" has been invoked on the currently executing program, and return an adjusted value for the end of available user-memory.

20. When the DIO package is linked in to a program, alternate versions of "getchar" and "putchar", whose sources are in DIO.C, get used.

BDS C User's Guide Addenda
v1.46 Edition -- March, 1982

Leor Zolman
BD Software
33 Lothrop st.
Brighton, Massachussetts 02135

There have been several new sets of features added to BDS C v1.46. The new features fall into three catagories: preprocessor enhancement, CP/M-specific compiler performance enhancement by selective overwriting of the CCP (Console Command Processor), and new utility programs (including CASM.C, which provides for the creation of CRL-format object files out of assembly language source files WITHOUT the need for MAC.COM and the old CMAC.LIB macro package).

The preprocessor enhancements are as follows:

0. Parameterized #defines are now supported. This allows a macro in the form of a function call to be expanded (before compilation) into an arbitrary string, with the original parameters substituted into the string. For example, the sequence

```
#define foo(x,y) x * 3 + y
.
.
.
z = foo(bar,zot());
```

results in the final line actually reading:

```
z = bar * 3 + zot();
```

0.5 One feature of "#define" substitution has been slightly changed: when a symbolic constant appears in the definition of ANOTHER symbolic constant, then the substitution of the first constant does not take place until the substitution of the second does. This means that in a sequence such as

```
#define FOO 1
#define BAR FOO+1
```

the string that gets substituted for "BAR" depends upon the current definition of "FOO"; if "FOO" got re-#defined at some point, "BAR" would change accordingly. Given the above example, in past versions of BDS C "BAR" became "1+1" at its definition point and would not have changed even if "FOO" were re-#defined, unless "BAR" was also re-#defined after "FOO".

1. The

```
#if <expr>
```

conditional compilation directive is now supported, but only with a special

limited syntax for the expression argument, defined as follows:

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

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

The <constant> may be a symbolic constant, but is treated as a logical value always...i.e, 0 is false and any non-zero value is true (1). This allows users to write system-dependent conditional expressions without having to resort to #ifdef/#ifndef and commenting/un-commenting #define statements to yield the desired conditions.

2. Nesting of conditional compilation directives is now allowed, and incorrect nesting attempts will now draw an appropriate error instead of doing random things to the source text. Note that each and every #else directive MUST be followed by a matching #endif (unlike C's control structure syntax, in which an if...else chain may be extended as long as desired.)

****

The following enhancements to the v1.46 compiler and linker affect the USAGE of the compiler, not the C language syntax it accepts:

In the past, the compiler and linker have performed a CP/M warm-boot after every compilation had either been completed or aborted due to an error. For v1.46, a warm-boot will only take place when the memory occupied by the Console Command Processor (CCP) is actually needed for the task. Since there is usually plenty of memory left over after a compilation or linkage, I decided to eliminate the pain of having to wait for the system to re-boot after each and every usage of the compiler or linker.

On certain "fake" CP/M systems (I believe the CROMIX CP/M emulator is one such case), the non-warm-booting return to the CCP does not work correctly, probably because the system does not pass a valid stack pointer to transient commands. The symptom is crazy behavior after CC1, CC2 or CLINK complete execution; the output files will have been written OK, but attempting to return to the system via the passed SP bombs the system. To correct this problem, it is necessary to make a patch to each of the three command files forcing them to re-boot when finished. The patches are as follows:

file	address	old data	new data
CC1.COM	03AD	2A C6 03	C3 00 00
CC2.COM	0239	2A 0A 01	C3 00 00
CLINK.COM	0F39	2A 73 13	C3 00 00

One feature of BDS C in the past has been that it automatically aborted any pending "SUBMIT" file after compilation when an error had been detected during the compilation. This had required the compiler to seek to the directory track on disk and erase "$$$.SUB" before re-booting, but the extra time thus spent was negligable

since a seek to the low tracks was coming up soon anyway in order to do the warm-boot. Now, since a warm-boot isn't standard anymore, and the compiler is often used without being in a "submit" file, the compiler no longer AUTOMATICALLY aborts "submit" files following an error. The feature IS availalable, though, through the new "-x" option to CC1. If "-x" is given on the CC1 command line, then "submit" files will be aborted following an error. Any time CC1 is used in a "submit" file, "-x" should appear on the command line in the "submit" file. If CC1 is used stand-alone, then "-x" should not be used (it would just cause some needless disk activity upon error.) MAKE A NOTE OF THE "-X" OPTION UNDER THE CC1 OPTIONS SECTIONS OF THE BDS C USER'S GUIDE. Since CLINK is not aborted very often, it has not been given a "-x" option and (as in previous versions) will always abort pending "submit" files when prematurely terminated.

Note that both the compiler and linker now send a bell character (control-G) to the user console after completing a task in which one or more errors have occurred. This is to alert the user in the case of a premature completion and return to command level (as when a fatal error is detected by the compiler), since audible warm-boots no longer serve to notify the user of compiler termination.

On some interrupt-driven systems, type-ahead during operation of CC1, CC2 or CLINK does not work because each of these commands look at the console input to see if a control-C has been typed, in order to determine if the user wants to abort the comand. If any character other than a control-C is detected, that character is thrown away because there is not way to push it back under CP/M. If you wish to disable the control-C-polling feature of the BDS C commands, so that the console input is never sampled and type-ahead works correctly, make the following patches to the commands:

file	address	old data	new data
CC1.COM	0995	E5	C9
CC2.COM	04A6	E5	C9
CLINK.COM	061C	F5	C9

Note that after these patches are made, typing control-C will only abort a CC1, CC2 or CLINK invokation if provision is made in your interrupt-driven BIOS for general-purpose program interruption by control-C.

****

The major new utility program included with v1.46 is CASM.C, an assembly-language-to-CRL conversion preprocessor. CASM takes a specially-formatted assembly lanaguage source file having extension ".CSM" as input, and puts out an ".ASM" file which may then be assembled using the standard CP/M assembler (ASM.COM), to eventually produce a CRL-format object file. Note that sources to the assembly-language portion of the BDS C library are now provided as ".CSM" files instead of ".ASM" files, and a "submit" file named "CASM.SUB" has been provided to automate the entire process of "CSM"-to-"CRL" conversion. A separate document detailing the operation of CASM is included with the BDS C v1.46 package.

A new wild-card expansion utility, named WILDEXP.C, allows ambiguous file names to be specified on the command line to C-generated programs; then by a simple function call, the ambiguous references are expanded to include all filenames on the current disk that match the specification. Exceptions may also be specified.

A new utility named NOBOOT.C is also included: when NOBOOT.COM is invoked upon a COM file produced by the C compiler, it will make some magic changes so that the COM file no longer performs a warm-boot after completing execution. The changes involve forcing the run-time stack to begin BELOW the CCP, and having the program save the system stack pointer passed to it by CP/M so that the SP may be restored after execution and control can pass directly back to the CCP. NOBOOT should be used ONLY with programs linked using the standard, supplied form of the run-time package (C.CCC). Note that the "topofmem" library function has been modified to recognize when NOBOOT is in effect at run-time, and should return the correct value for the end of available user memory in all cases.

****

The following bugs have been detected and corrected for BDS C v1.46:

1. CC1 had crashed when an "#include" file was not terminated with a carriage-return/linefeed sequence.

2. CLINK no longer complains about not being able to find "DEFF3.CRL" when there are undefined function references in a linkage; if DEFF3.CRL does exist, it will be searched, but if it does not exist, that fact will no longer draw an error.

3. Literal strings having continuation lines might have confused the CC1 preprocessor in some versions, to the effect that a "#defined" symbol name that happened to match a character sequence within the continuation line of the string was incorrectly substituted for by the preprocessor, and such a symbol appearing AFTER the end of the string was NOT substituted for.

4. In the DIO package, the variable "c" in the "getchar" function was incorrectly declared as a "char" instead of an "int"; this caused a physical EOF to be returned as the value 255 instead of -1. Note that this problem only appeared when the text file was not terminated by a CPMEOF (control-Z) character.

5. Another DIO-related bug: when text containing both carriage-returns and linefeeds was fed to the DIO "putchar" function, an extra linefeed character was appended to each line and resulted in an extra blank line between each actual line of the output file. This has been fixed by building some state information into the DIO version of "putchar" so that the redundant linefeeds are not generated.

6. CLINK now warns the user when the address of the end of the external data area falls above the effective "top of memory" address (and thus not leaving any room for the run-time stack) to prevent hair-pulling confusion if such a condition is not noticed by the user. If you are generating special-purpose code in which you purposely tell the linker that the top of memory is below the external area, then just ignore the error message.

7. The "execl" function had two bugs which have been corrected: it had bombed if an attempt was made to pass more than six parameters, and it had not detected when the total size of supplied parameters exceeded the amount of space available for that text during the chaining operation (about 83 characters). Now any number of parameters are handled correctly, and a text overflow will cause "execl" to print a special message to that effect and also return a value of ERROR (-1) to

the calling routine.

8. The "gets" library function has been modified to use the stack during its BDOS call to get a line of text, and then copy the result into the supplied buffer area. This means that the buffer area passed to "gets" need no longer be 2 bytes longer than the longest expected string; but, "gets" still does not know how long the buffer you give it really is and you must make sure to supply a large enough buffer (when "gets" calls BDOS function 10, it supplies the BDOS with a 135-byte buffer on the stack, and as much of this as is filled up is copied to the user-supplied buffer upon return from the BDOS call).

   A new alternative to "gets" has been supplied, called "getline", which works just like the "getline" function shown in Kernighan & Ritchie. The format is:

```
int getline(strbuf,maxlen)
char *strbuf;
int maxlen;
```

   "Getline" collects a line of text from the user, where the maximum allowed length of the line is "maxlen" characters (where "maxlen" is supplied as a parameter). The return value is the length of the entered line. Since "getline" also uses BDOS function 10 to collect the line, a call such as "getline(str,135);" would work the same as "gets(str);". Use "getline" either to limit the line length to some small number, or to allow longer lines (up to 255 characters) than the maximum of 135 that "gets" allows.

   Note that both "gets" and "getline" will return immediatly if the number of characters typed reaches the maximum allowed (135 for "gets" or 'maxlen' for "getline"), even if no newline (carriage-return in this case) is typed by the user. This is due to the behavior of the BDOS, and there aint' nuthin to be done about it short of writing an entire "gets" from scratch in terms of low-level character I/O, and that just isn't worth the trouble.

BDS C User's Guide Addenda
v1.45 Edition -- December, 1981

Leor Zolman
BD Software
33 Lothrop st.
Brighton, Massachussetts 02135

Here are the bug fixes and extensions for BDS C version 1.45.

Note: If you are running under MP/M II, be sure to see item 10 below!

0. Expressions of the form

```
!(expr || expr)
or  !(expr && expr)
```

may not have worked correctly when a VALUE was required for the expression; i.e., when used in some way other than in a flow control test. For example,

```
x = !(a || b);
```

might have failed, but

```
if (!(a || b)) return 7;
```

would have worked, since the expression was used for flow control.

1. Declarations of pointer-to-function variables for functions returning a CHARACTER value caused only one byte of storage to be reserved for the pointer, instead of two bytes (all pointers-to-functions require two bytes of storage, by virtue of being pointers). For example, in the sequence:

```
char c1, (*ptrfn)(), c2;
...
ptrfn = &getc;
```

the assignment to `ptrfn' would have incorrectly overwritten the `c2' character variable, since only one byte would have been reserved on the stack for the `ptrfn' variable while the assignment operation would have assumed there were two bytes reserved.

2. A bug in the ternary operator evaluator (?: expressions) caused the high-order byte of a 16-bit result to be incorrectly zeroed in the following situation: given a ternary expression of the form

```
e1 ? e2 : e3
```

where `e2' evaluated to a 16-bit value (int, unsigned or pointer) and `e3' evaluated to a character value (type char only), the entire expression was treated as having type char...so if `e1' was true and `e2' was bigger than 255, then the value of the expression ended up as only the low-order byte of the value of `e2'. For version 1.45, whenever `e2' and `e3' do not BOTH evaluate to character values the type of the overall expression is guaranteed not to be char.

3. A sequence of two `!' (logical `not') operators in a row did not always produce the correct result in an expression. For example,

```
x = !!n;                    /* convert n to a logical (0 or 1) value */
```

might have produced the wrong result (0 instead of 1, or vice-versa).

4. A stack-handling bug in CC2 caused problems at run time when a sufficiently complex sub-expression appeared in any but the final position of an expression involving the comma operator (","). For example, the following statement would not have worked correctly:

```
for (i = 0; i < 10; x += y, i++) ...
```

5. CC1 has not been recognizing illegal octal character constants as such; digits such as `8' and `9' within an octal constant will now draw an error in cases where they would have been ignored before. Also, certain other forms of illegal constants (aside from character constants) are now better diagnosed than before.

6. I found one more case where an internal table overflow during code generation was not detected, causing the final command file to bomb as soon as it was executed (either by crashing the machine or immediately re-booting.) This occurred when a single large function containing many string constants was compiled. All fixed now.

7. An extension to the linker: CLINK now recognizes "DEFF3.CRL" as an automatic library file, similar to DEFF.CRL and DEFF2.CRL. Note that there is NO DEFF3.CRL file included with the BDS C package; this feature has been added to allow you to fit more custom functions into your library than just what fits in DEFF.CRL and DEFF2.CRL (which are getting rather full.)

   Also, CLINK will now search ALL default library files (DEFF.CRL, DEFF2.CRL and DEFF3.CRL [if it exists]) when a carriage-return is typed in interactive mode. Previously, only the file DEFF.CRL was searched in response to carriage-return.

8. It has been brought to my attention that the ^Q-CR sequence required by CLINK in interactive mode (to abort the linkage in progress) cannot be typed in under MP/M systems, since ^Q is used to detach a process. If you are running MP/M, then just type control-C instead of ^Q-CR; this will also work for CP/M systems...the only difference is that when ^Q-CR is used, then any currently active "submit file" processing is automatically aborted by CLINK before returning to command level, as a convenience (I assume that if you abort the linkage, you don't want to continue with your submit

file...). Under MP/M, you'll have to type characters quickly at the keyboard (after ^C-ing CLINK) to abort any pending submit file activity.

9. A slight bug in CLIB.COM (The C Library manager program) made it hard to exit CLIB from within a submit file (assuming XSUB is in use). The problem was that CLIB requires a confirmation character, `y', to be typed after the `quit' command is given. CLIB was getting the confirmation character by doing a single direct BDOS console input call, which required the user to manually type in the letter before any pending submit file processing could continue. This has been fixed by having CLIB get an entire line of input (using BDOS call 10) when seeking a confirmation; now the `y' may be inserted into submit files. Note that the `quit' command and the `y' confirmation must be placed on separate consecutive lines in the submit file. If not using a submit file, the only difference is that now a carriage-return is required after typing the `y'.

   Another minor problem with CLIB: function names longer than 8 characters were not being truncated when entered for operations such as renaming, resulting in too-long CRL file directory entries. All names are now properly limited to 8 characters.

10. A problem with file I/O under MP/M Version II has come up: The run-time package routine "vclose", called by the library function "close" whenever a file needs to be closed, has been optimizing for files open only for reading by NOT actually performing a "close" operation through the BDOS. This worked fine under CP/M, because CP/M didn't care whether or not a file that has had no changes made to it was ever closed; MP/M II, on the other hand, DOES seem to want such files to be explicitly closed...so by running many programs that didn't close their Read-only files, BDS C programs eventually caused MP/M to not allow any more files to be opened.

    This problem has been fixed by adding a conditional assembly symbol, called "MPM2", to the CCC.ASM source file. If you are running under MP/M II, you should set the "MPM2" equate to true (1) and reassemble CCC.ASM, yielding a new C.CCC after loading and renaming (you should only need ASM.COM for this, although MAC.COM works also). The change does NOT affect the size of C.CCC, so the libraries do not have to be reassembled as is usually the case when the run-time package is customized. The change simply causes a single conditional jump to be turned into three nop's, so that ALL files are always closed, instead of only the ones open for writing. My apologies to MP/M users who may have had confusing troubles because of this bug.

11. A bug was found in the `_scn' library function (affecting `scanf'): when a lone carriage-return (newline) was typed in response to a "%s" format conversion, the format conversion was totally ignored. This caused the target string to remain unchanged from its previous contents, instead of correctly having a null string (consisting of a single zero byte) assigned to it.

12. A bug was found in the `_spr' library function (affecting `printf', `sprintf', and `fprintf'): The default field width value was 1, causing a null string to be printed as a single space when the standard "%s" format conversion was used. For example, the statement:

```
printf("Here is a null string: \"%s\"\n","");
```

would have produced the output:

```
Here is a null string: " "
```

instead of:

```
Here is a null string: ""
```

The default field width value has been changed to 0, so null strings will now print correctly. An explicit field width may always be given in any format conversion, of course.

13. When the library function "sprintf" (formatted output directly into a memory buffer) is used, a null byte is appended onto the end of the output text. I'm not absolutely sure whether or not this is a "desired" characteristic; at least one user has complained about it, but it turns out that "sprintf" on the large-scale Unix system I have access to does the same thing and I can think of applications where the trailing null is useful. So, the null stays in.

14. In several library functions, as well as at one point in the run-time package, calls were made to BDOS function number 11 (interrogate console status) followed by an "ani 1" instruction to test bit 0 of the value returned by BDOS. I've been told that on some systems, testing bit 0 is not sufficient since sometimes values other than 0 and 1 (or 0 and 255) are returned. SO, all such sequences have been changed to do an "ora a" instead of an "ani 1", so that a return value of exactly 00h is interpreted as "no character ready" and any other value is interpreted as "yes, there is a character ready". The library functions that were modified this way are: `kbhit', `putchar', `srandl', `nrand', `sleep' and `pause'. The sequence to clear console status in the run-time package (CCC.ASM), near the label "init:", has likewise been changed (but a "nop" instruction was added to keep all addresses consistent with earlier versions of the run-time package.)

15. When customizing the run-time package (CCC.ASM) with the "cpm" symbol equated to zero, several symbols (named "SETNM" and "SETNM3", at the routine labeled "PATCHNM") were undefined; this has been fixed by adding some conditional assembly directives to insure that the labels in question are not referenced under non-"cpm" implementations, while the total code size remains constant so that the addresses of later run-time package utility subroutines stay exactly the same for all implementations.

16. 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 C 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. 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. So, 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 version 1.45 was 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). You might want to make a note in the User's Guide library section (page 30) to the effect that A and B are now ignored. 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 (with MAC.COM, CMAC.LIB and BDS.LIB) 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.

17. The "creat" library function had been creating new files and opening them for writing ONLY; this caused some confusion, so `creat' has been modified to open files for both reading AND writing following creation. PLEASE MAKE A NOTE OF THIS UNDER THE `CREAT' ENTRY IN THE STANDARD LIBRARY SECTION OF THE BDS C USER'S GUIDE.

18. The "execv" function has been changed to return ERROR (-1) on error, instead of forcing an error message ("Broken pipe") to be printed to the standard error device. The reason I originally had it printing "Broken Pipe" was because I was too lazy to figure out how to fix the stack after passing all the arguments; following some justified bitching from Scott Layson I went in there and fixed it so it does something reasonable. PLEASE MAKE A NOTE OF THIS UNDER THE `EXECV' ENTRY IN THE STANDARD LIBRARY SECTION OF THE BDS C USER'S GUIDE.

19. The DIO (directed I/O and pipes) package contained an obscure bug: if a pipe operation was aborted before completion, leaving a "TEMPIN.$$$" file in the directory, then the next pipe operation performed had gotten its own output mixed up with the output of the aborted pipe....the old output was used as input to the new next command, and the

new output was lost. The new DIO.C has been fixed. (Note: DIO.C has also been slightly changed to properly interact with the new version of the "execv" library function.)

20. Another change has been made to the DIO package: the "getchar" function, when used without input redirection to read characters directly from the console, had not allowed for line editing in previous versions. I.e., each character was obtained by a direct BDOS call and none of the special line editing characters (delete, ^R, ^U, etc.) were recognized. For version 1.45, an optional line buffer mechanism has been added to the DIO package so lines of console input can be fetched at one time by using the "read console buffer" BDOS call and all editing characters now function as expected. Operation of the package using buffered console input is still the same as before, except for one thing: to enter an end-of-file character (control-Z), it is now necessary to also type a carriage-return after the control-Z.

    To enable console input buffering when using the DIO library, it is necessary to un-comment a line in the DIO.H file and re-compile DIO.C. See the comments in DIO.C for more information.

21. The special case handler for the code generator has been improved to more efficiently handle relational binary operations where exactly one of the operands is a constant. The operators affected are: "<", ">", "<=", ">=","==" and "!=", for both signed and unsigned data types. The improvement is mainly in the speed of execution of such comparisons; statements such as:

```
if (i < 1234) ...
```

    execute much faster. This results in speedier execution of programs such as the Seive of Eratosthenes benchmark in the September '81 issue of BYTE: the current version of BDS C, using the -e and -o compiler options with variables made external, does it in 15.2 seconds (see SIEVE.C on the distribution disk.)

    Also, multiplication by a constant that is a low power of 2 (2,4,8,16) is now done by DAD H sequences instead of calls to the run-time package multiply routine [so that expressions such as (i * 8) and (i << 3) each compile to the same code].

22. Two new functions have been added to the standard library:

```
int setjmp(buffer)
char buffer[JBUFSIZE];

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

    When "setjump" is called, the current processor state is saved in the JBUFSIZE-byte buffer area whose address is passed as the argument ("JBUFSIZE" is defined in BDSCIO.H), and a value of zero is returned. Whenever a subsequent "longjump" call is performed (from ANYWHERE in the current function or any lower-level function) with the same buffer argument, the CPU state is restored to that which it was during the "setjmp" call, and the program behaves as if control were just returning from the "setjmp" function, except that the return value this time is "val" as passed to "longjmp". 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, to make sure

that a particular exit routine (e.g., the directed I/O "dioflush" function) is always performed. It is a nifty facility that should have been available long ago. THESE FUNCTIONS ARE NOT DOCUMENTED IN THE BDS C USER'S GUIDE; PLEASE MAKE A NOTE OF THEM IN THE STANDARD LIBRARY SECTION OF THE GUIDE.

23. A new linker for BDS C called "L2" (a substitute for CLINK.COM) is now available from the BDS C User's Group. L2, written by Scott Layson (of Mark of the Unicorn) in BDS C, has several interesting features:

    1. L2 can link programs that are up to about 8K larger than CLINK: if there isn't enough room in memory to hold the entire program while building an image in memory, L2 performs a disk-buffering second pass. This means that the resulting COM files can be as large as the entire available TPA on the target machine.

    2. The number of functions per program is no longer limited to 255.

    3. While CLINK uses jump tables at the beginning of functions to resolve references to other functions, L2 totally eliminates the jump tables and instead generates direct external calls. This shortens programs by anywhere from 3% to 10%, and also speeds them up a little.

    4. Since L2 is written in C, you can customize it yourself.

    The L2 package comes with source code, a special overlay generator program and documentation. It is available to BDSCUG members for the nominal cost of media and shipping (currently $8). See the next note for information on joining the BDSCUG.

24. The BDS C User's Group membership forms should now be included with the BDS C package...this makes life easier for everyone, since it is no longer necessary to write to the Group first just to ask for forms before being able to order library disks. BDS C User's Group members receive the Group newsletter approximately 6 times per year, and are entitled to compiler updates and library disks for low prices (typically $8 per disk).

BDS C User's Guide Addenda
v1.44 Edition -- April, 1981

Leor Zolman
BD Software
33 Lothop St.
Brighton, Massachussetts 02135
(617) 782-0836

Please note my NEW new address and phone number...some earlier versions of the new documentation have said that my new city and zip code were Allston, 02134, which is where I THOUGHT I was. Actually, I'm in Brighton, 02135, and any mail sent me addressed to Allston may have been returned to the sender stamped with something like "No such address known." Sorry about that.

Here are the bug fixes/extensions for version 1.44:

1. (Applies to v1.43a only): the character sequence \\ appearing at the END of a quoted string caused the preprocessor in CC1 to screw up and stop stripping comments for the rest of the source file. For example, the statement:

```
printf("This backslash would cause big trouble: \\");
```

   would have done it.

2. The "qsort" library function didn't work when the total size of the data array being sorted exceeded 32K bytes. This has been fixed by changed the declarations of certain variables in qsort from "int" to "unsigned".

3. CC1, CC2, and CLINK may now be aborted in the middle of execution by typing a control-C.

4. A new CLINK option has been added (as if there weren't enough of them already...) The "-f" option, when specified immediately before the name of an extra CRL file to be searched, FORCES all functions in that CRL to be loaded into the current linkage--even if they haven't been previously referenced. This provides a simple solution to the backwards-reference problem; a typical case when this would be used comes up when you want to use a special version of a low-level function such as "putchar." If you have a complete program such as:

```
main()
{
        printf("this is a test\n");
}
```

   and would like your OWN version of putchar to be loaded from a library called, say, SPECIAL.CRL (which you have previously compiled), then simply saying:

```
clink test special <cr>
```

would NOT work, because the "putchar" function doesn't become "needed" until AFTER the library file DEFF.CRL, which contains "printf", is searched...which doesn't happen until AFTER special is searched! So the "putchar" finally loaded would come from DEFF2.CRL, which is the library file automatically searched after DEFF.CRL. To make this do what you want, all you'd have to do now is:

```
clink test -f special <cr>
```

which would force everything in SPECIAL.CRL to be loaded right away, before the DEFF files are scanned. Then, when "printf" gets loaded from DEFF.CRL, the correct "putchar" function will already have been loaded and the one in DEFF2.CRL will be ignored.

5. The "rename" library function had a rather serious problem: whenever executed, it would zero out the three bytes of code immediately after the end of the function (i.e., the first jump instruction of the next function in memory would get clobbered.) This problem was fixed by increasing the amount of storage declared in the "ds" at the end of "rename" from 49 bytes to 53 bytes.

6. The "setfcb" function requires that the buffer allocated to hold the resulting fcb is AT LEAST 36 BYTES LONG! "Setfcb" zeroes out the random-record field bytes of the fcb just in case the CP/M 2.x random-record file I/O mechanism is later used. But whether you use the random stuff or not, the fcb you allocate still has to be 36 bytes long.

7. This bug applies to v1.43 only: A character constant consisting of the double-quote character enclosed in single quotes ('"'), when encountered by ccl, caused ccl to stop stripping comments while reading in the rest of the source file from disk. This was a bug in the v1.43 code added to allow comment delimiters within quoted strings.

8. Whenever the type information for a function definition was placed on a line separate from the actual name of the function, then the compiler would "lose" a line of code and all errors found past that point in the source file would be reported with an incorrect line number. For example, the following kind of function definition would've caused this problem:

```
char *
foo()
{
        ...
}
```

9. A new library function, "execv", has been added to the package (source is in DEFF2.ASM). This function allows chaining to another COM file with a variable number of command line parameters (note that "execl" requires all of the arguments to be explicitly passed as string pointer parameters to the function, so that one particular call can only have the number of arguments that it was written with.) The format of the "execv" function is:

```
execl(prog,argvp)
char *prog, **argvp;
```

where `prog' points to the name of the COM file to be chained to, and `argvp' is an `argv'-like pointer to an array of pointers to text parameters. The final pointer in the list must be followed by a null pointer. As an example, note that the "execl" call

```
execl("stat","badspots","$r/o",0);
```

can be written in terms of "execv" as follows:

```
char *args[3];
...
args[0] = "badspots";
args[1] = "$r/o";
args[2] = NULL;
execv("stat",args);
```

10. Directed I/O and pipes, of sorts, are now available to BDS C programmers. The files DIO.C and DIO.H make up a cute little directed I/O package, allowing for directed input, directed output and pipes (a la Unix) on the command lines to programs compiled with this special I/O package. See the comments in DIO.C for complete details. Note that the presence of this package does NOT contradict certain comments made in the User's Guide about kludging advanced Unix features under CP/M; those comments were directed toward systems in which the I/O redirection/generalization is forced upon the user, along with all the entailing overhead, when the redirection isn't needed or wanted for many applications. The DIO package, being written in C and separately compiled, lets YOU the USER decide when you want it and when you do not. If you don't want it, it takes up zero space; if you do, it takes up a bit of room and yanks in all the buffered I/O, but it DOES give you redirection and pipes!

11. A "standard error" buffered I/O stream number has been added to the list of special devices recognized by the "putc" buffered output function. An iobuf value of 4 causes the character given to be written to the CP/M console output, always, while an iobuf value of 1 causes the character to be written to the standard output (which might be a file if the DIO package is being used.) Note that 4 was used instead of the Unix Standard-error value of 2 because 2 had already been taken (by the CP/M LST: device.)

12. String constants may now contain zero bytes within them. Previous versions have flagged lines such as

```
foo = "Jan\0Feb\0Mar\0Apr\0May\0Jun\0Jul\0Aug\0Sep\0Oct\0Nov\0Dec\0";
```

with the error message:

```
Zero bytes are not allowed within strings; to print nulls, use \200
```

Note that allowing the above kind of string constant makes it easier to initialize a table of homogenously-sized strings; the example with the months could be part of a function that returns a pointer to the name of some month n, where n is a passed

value ranging from 0 to 11 (or from 1 to 12, or whatever...)

BDS C User's Guide Addenda
v1.43 Edition -- March, 1981

Leor Zolman
BD Software (New Address!)
33 Lothrop st.
Brighton, Ma. 02135
(617) 782-0836

Before getting on with the business at hand (where I shamelessly display all the horrible bugs that have plagued previous versions of the compiler), I'd like to take a moment to answer one of the more common questions that have been asked of me by users and potential users of BDS C. Hopefully, this will save some of you the expense of a phone call (which can run pretty high when I get to rambling...)

Q. What is the royalty arrangement for software developed using BDS C?
A. There is NO royalty arrangment AT ALL. Both the BDS C runtime package and function libraries, in either source or object form (or both), may be freely distributed with commercial (or non-commercial) application programs. The reason for this policy is to promote the use of C for anything and everything, without wrapping up potential applications in miles of red tape and ineffective security measures. Software authors: PLEASE include the source listings to your software with your packages! I understand that there are some markets where such generosity is considered suicidal, and I sympathize in many cases, but I also want to see BDS C selling more copies, and providing the source to applications programs will encourage users to obtain the compiler. Hopefully, some of them may even BUY it.

OK, now it's time for the bug reports. Following, in decreasing order of severity, are the bugs found and fixed for v1.43, and some additional notes:

0. Another logical-expression-related bug caused incorrect code to be generated when a subexpression of a binary operation used the && or || operators. For example,

```
if (x > (i==5 && j<7)) printf("Foobar\n");
```

might have caused a crash when executed.

0.5 A bitwise or arithmetic binary operation in which the left argument was a logical expression of any kind and the right argument was a binary expression of higher precedence failed to evaluate correctly. For example,

```
if (!kbhit() & a<5) printf("foo\n");
```

didn't work.

1. A missing comma, such as in the statement:

```
sprintf(dest "x = %d\n", x);
```

went undiagnosed and caused wierd code to be generated. (The bug fixed in the last release had only corrected the case of a missing comma AFTER a format string specification, not BEFORE it...)

2. If a comment was begun on a line which contained an "#include" preprocessor directive, and not terminated until a later line, then CC1 became confused. 2a. Several users have complained about not being able to put the character sequence `/*' into a quoted string. This is a justifiable gripe, but I'm afraid you'll have to say things like "/*" to get the same effect. The reason comment delimiters are not tolerated within quotes

3. Mismatched curly-braces in a source file now draw a more meaningful diagnostic than the previous "Unexpected EOF encountered" message: a pointer is now provided to the line at which the badly-balanced function begins.

4. When an illegal constant was encountered by CC1 at any place where a constant is required, an incorrect "Unmatched left parenthesis" diagnostic was displayed with an impossibly large line number. (Actually, the correct line number was obtainable by subtracting the exact size of the text file from the given line number. Guess what I forgot to initialize between passes...)

5. When using the "-w" option with CLINK, a terminating control-Z was NOT put out to the SYM file when the length of the SYM file worked out to be an exact multiple of 128 bytes. This gave CLINK a headache when "-y" was used to read the SYM file back in.

6. There was another bug in the "getc" library function that caused some trouble when the "fgets" function was used to read in lines from a text file that wasn't terminated with control-Z (CPMEOF). This was fixed by changing the line:

```
return ERROR;
```

to:

```
return iobuf->_nleft++;
```

7. Mismatched square brackets in an expression had drawn an "Unexpected EOF encoutered" error instead of something more meaningful.

8. The word "main" is NO LONGER A KEYWORD. In previous versions, the fact that "main" was treated as a keyword made its use in any situation other than as the first line of a "main" function impossible. I.e, attempts to call "main" recursively were not accepted by the compiler. There is now no longer anything special about the word "main". In addition, previous versions had substituted an undocumented one byte code (9D hex) for the name "main" in CRL file directories, thereby probably causing a lot of confusion. This bizarre scheme is no longer used, although the linker will still recognize the special 9D code as meaning "main" when encountered in a CRL file (of course, "MAIN" will now also be recognized...)

9. A bug in the "-y" option handler in CLINK caused CLINK to crash when there wasn't enough room in the reference table to hold all the symbols being read in from a SYM file. Sorry about that, chief. Note, by the way, that the POSITION of "-y" on the command line IS VERY SIGNIFICANT. If the "-y" option appears to the right of names of CRL files to search, then the SYM file specified will not be used until AFTER the previous CRL files have already been scanned and loaded from. I.e., the "-y" option should appear BEFORE the names of any CRL files that contain functions that might not need to be loaded (due to their definition in the SYM file). A new feature of CLINK is that whenever a previously defined symbol is encountered in the process of loading the symbols from a SYM file, a message to that effect will be printed, allowing the user an opportunity to rearrange the command line so that the SYM file is read in earlier and some redundancy possibly eliminated.

10. An obscure feature of the "printf", "sprintf" and "fprintf" library functions, as described in the Kernighan & Ritchie book, is that a field-width specification value preceded by a '0' caused 0-fill instead of space-fill. I'd never NOTICED that before, until a user brought it to my attention (and conveniently provided a fix.) Note that this solves a problem often encountered when printing hex values. Now, the following "printf" call:

```
printf("%4x; %04x\n",8,8);
```

will produce the output:

```
8; 0008
```

11. The body of a function definition now MUST be enclosed in curly-braces. Formerly, the following sort of thing was tolerated as a function definition, but no more:

```
putchar(c) bdos(4,c);
```

12. A bug in the CMAC.LIB macro package had NOT allowed lines such as:

```
exrel <lxi h,>,putchar
```

while the following kind of lines were properly handled:

```
exrel call,putchar
```

13. A new low-level character I/O function package, named CIO.C, has been added for greater flexibility in console interaction, especially for game-type applications. Note, however, that code generated using this facility is NON-PORTABLE from one system to another unless the "other" system is also equipped with a C compiler. If you HAVE to, go ahead and use it, but please resist the temptation to give out a copy of the compiler to your friends along with your source code.

14. Quoted strings containing an open-comment delimiter sequence (`/*') had caused CC1 to think an actual comment was intended. I.e, the statement

```
printf("this is an open-comment sequence: /* \n");
```

would have drawn a "string too long..." error. Not any more.

15. The handling of string constants by the code generator has been improved. Now, instead of putting the text right where it is used and generating a jump around it, the compiler accumulates up to 50 text strings in a function and places them all at the end of the function. If more than 50 strings appear, then after the 50th it goes back to doing it the old way for the remainder of the function (there's only so much table space worth allocating to hacks like this.)

16. Speaking of hacks, here's one that'll get you either excited or sick: You say you need some "static" variables? Consider the following method of simulating a "static array of characters":

```
char *static;
...
static = "0123456789";
...
```

The result is that the variable "static" may be used just like a static array of ten characters. If declared as an "int" instead of a "char", it could be used as an array of five integer variables (or ten, if you make the quoted string twice as long...). Steve Ward makes use of this technique in his CIO.C library. Kludgey, yes, but it gets the job done and it's even portable...

17. The default CC1 symbol table size for modified versions of the compiler (v1.43T) has been upped from 6K to 7K. The "-r" option still lets you explicitly set the table allocation, if you want to.

```
*****************************************************************
*                                                               *
*    The New Dynamic Overlay Scheme.....for BDS C v1.4          *
*                        August, 1980                           *
*                                                               *
*****************************************************************
```

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 new set of capabilities has been built into the CLINK program. Normally, the run-time environment of an executing C program looks 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)
-----------------------------------------------------------------
```

Note that "xxxx" is the first location following the program code and "y" is the amount of memory needed for external variables.

To implement overlays, the first thing necessary is to 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 from where the externals ended, making it difficult to determine the lowest memory location safe to swap into. The scheme suggested then for handling overlays was to leave sufficient room between the end of the root segment code (the root segment contains the "main" function and run-time package; it loads at the start of the TPA, always remains in memory, and controls the top level of overlay swapping) and start of the external data area to accommodate the largest possible swapped-in segment combination. This is still a viable scheme for version 1.4; here is the modified memory map, accommodating this first 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)
-------------------------------------------------------------------
```

Note that "zzzz" is where segments get swapped in, guaranteed that the longest segment doesn't reach "xxxx".

With version 1.4, it is just as feasible to put the overlay area AFTER the externals. The memory map for this alternative configuration would be:

```
-------------------------------------------------------------------
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)
-------------------------------------------------------------------
```

If you plan to use the storage allocation functions (alloc, free, sbrk, rsvstk) in your

program, then this second scheme would require you to call the "sbrk" function with argument "ssss" (the size of the overlay area) since, by default, storage allocation always begins with the area immediately following the end of the externals. For the remainder of this document, I will assume the FIRST of the above two schemes is being used.

OK, with the generalities out of the way, let me say something about just how to create "root" segments and "swappable" segments with BDS C. First of all, we would like all functions defined within the root segment to be accessible by the swapped 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 the externals to hold the largest swapped-in segment at run time (I'm using the term "below" in the sense that low memory is "below" high memory; graphically, in the preceding memory maps, "below" means toward the top of the page.) If the -e option is omitted, CLINK will assume the external data starts immediately after the end of the root segment code; this is OK only if you're using the SECOND scheme.

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 (for the first scheme.) 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 "foo" 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("foo",0x3000) != -1) /* check for load error */
     (*ptrfn)(args...);         /* if none, call the segment */
...
```

The "swapin" routine returns -1 when a load error occurs. Note that the swapped-in code might not return any value, 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 args 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 allowed entry point for the segment.

A simple "swapin" function is given in STDLIB2.C. It can be made shorter by skipping all the error testing, or 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 low-level hackery, you get the value of the 16-bit address at location ram+115h by using indirection on a pointer-to-integer (or -unsigned.) Note that location RAM+115h ALWAYS contains the address of the base of the external data area.

Now we know how to do everything except actually create a swappable segment.

OK, a swappable 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 a swappable segment (the C.CCC in the root segment will be shared by everyone.) The other difference between a swappable segment and the root segment is the load address; while the root segment always loads at ram+100h (where "ram" is 0 for standard CP/M, or 4200h for the "modified" CP/M), a swappable segment may be made to load anywhere. Once you've compiled the swappable segment, you give a special form of the CLINK command to link it:

```
A>clink segmentname -v -l xxxx -y symbolfile [-s ...] <cr>
```

where "segmentname" is the name of the CRL file containing the segment, "-v" indicates to CLINK that a swappable 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.

Since you'll probably want to yank in the symbol file created by the linkage of the root segment, use the -y option to do so. If you don't, then CLINK will yank in fresh copies of functions like "PRINTF" and "FOPEN", etc., even if they have already been linked into the root segment. It would be a waste to have multiple copies of those memory hogs in there at the same time! 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 swapped-in segment. The root segment, though, cannot know about functions belonging to the swapped-in segment 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 the segment, you may specify -s to generate a stat 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.

Example: (The addresses given in this example are for a RAM at 0000h CP/M; if you have the modified 4200h CP/M, fudge accordingly.)

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 (no "-v" option used), 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 segment would appear as:

```
A>clink seg1 -v -l 3000  -y root -s -o seg1. <cr>
```

The command line tells CLINK that SEG1.COM is to be a swappable segment (the "-v" option) 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.)

BDS C File I/O Tutorial

Leor Zolman
BD Software

The file I/O library functions provided with BDS C fall into two catagories: "raw" and "buffered." The raw file functions, typically coded in assembly language for best performance, are essentially a CP/M-oriented low-level interface where data transfers always occur in multiples of full CP/M logical sector (128 byte) quantities. The buffered functions (written in C) provide a byte-oriented, sequential file I/O system geared especially for "filter"-type applications; buffering allows you to read and write data in whatever sized quantities are most convenient while invisible mechanisms worry about things like sector buffering and actual disk I/O; thus the buffered I/O functions are usually more convenient to deal with than the raw functions, but they generate a lot of overhead by being slow and hogging up quite a bit of memory for code and buffer space.

Since buffered I/O is composed of raw I/O functions plus some extra code, I'll first present the raw I/O in detail, and then go onto the buffered functions.

The raw functions are characterized by their concern with "file descriptors". A file descriptor (fd) is a small integer value that becomes associated with a currently active file. This fd is always obtained by calling either the "open" or "creat" functions; their usage is:

```
fd = open(filename,mode);   /* `filename' can be either a literal */
                            /*  string or any expression that     */
fd = creat(filename);       /*  evaluates to a character pointer  */
```

The former is used to open an already existing file (usually, a file that has some data in it) for reading or writing or both, and the latter is used to create a brand new file and open it for writing. In both cases, the fd is the value returned by the call. If some kind of error occurs and the specified file cannot be opened or created, a value of ERROR (-1) is returned instead. For example, if "open" cannot find the file on disk whose name is pointed to by the first argument, ERROR will be returned.

All other raw functions require an fd to specify the file to be operated on (except "unlink" and "rename", which take filename pointers). The "read" and "write" functions are used to transfer data to and from disk. Their typical usage is:

```
i = read(fd, buffer, nsects);      /* `fd' must have been obtained by */
j = write(fd2, buffer2, nsects2); /*  a previous call to "open"      */
```

The first call would try to read, into memory at `buffer', `nsects' sectors from the file whose `fd' is specified. The second call would try to write `nsects2' sectors from memory at `buffer2' to the disk file whose fd is `fd2'. Unless an error occurs (such as when an illegal fd is given or an attempt is made to read past the end of a file), the above functions cause an immediate disk transfer to happen. This is one of the main differences between raw and buffered I/O: raw functions always cause immediate disk activity, as long as what they are asked to do is possible, while buffered functions only go to disk when a buffer fills up (when writing) or becomes exhausted (when reading.)

For each file opened under raw I/O, there exists an invisible "r/w pointer" to keep track of the next sector to be written or read. Immediately after a file is opened, the r/w pointer always starts at sector 0 (the first sector) of the file; it is bumped after "read" and "write" calls by the number of successfully transfered sectors, so that (by default) the next transfer happens sequentially. One nice extension of the BDS C raw I/O functions over their REALLY-raw CP/M equivalents is the elimination of the concept of "extents"; Instead of "extent numbers" and "sector numbers within the current extent" to be reckoned with for every file, there is only a single 16-bit r/w pointer to be considered. The value of a file's r/w pointer may be obtained by calling the "tell" function, and modified by calling "seek".

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>
```

This will take the file named by `filename' and create a copy of it named by `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 arguments were typed on the command line. It is sometimes convenient to summarize a program in a half-C/half-English pseudo code form to avoid going in blind; 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 that implements the above procedure:

```
#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 we shuffle chunks of disk files through memory. The size of the buffer is computed as the sector size (defined in BDSCIO.H) times the number of sectors of buffering desired (defined at the top of this program as BUFSECTS).

In the "main" function, the first thing to do is make sure the correct number of

arguments were given 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 arguments given PLUS ONE, we test to make sure it is equal to three (i.e, that two arguments were given). If argc is not equal to three, we call "perror" to print out a complaint and abort the program. "Perror" interprets its arguments as if they were the first two arguments to a "printf" call, performs the required "printf" call, aborts operations on the output file (this wouldn't have any effect if called before the file is opened; this would be the case if the "argc != 3" test succeeds), and exits to CP/M.

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 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 assignment to `fd1' of the return value from "open", and then the test to see if that value was ERROR. If the value was NOT equal to ERROR, control will pass onto the next `if' statement; otherwise, the appropriate call to "perror" diagnoses the problem and terminates the program. Creating the output file follows exactly the same pattern.

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 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 [EOF] condition) up to the number of sectors requested (in this case, BUFSECTS), with a value of ERROR being returned on disaster (when the disk drive door pops open or something). Whatever this value may be, it is assigned to `oksects' for later examination. In the special case when it is equal to zero, indicating EOF, the "while" loop will be exited. Otherwise, we enter the loop and attempt to write back out the data that we just read in. First, though, we want to make sure no gross error 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 the number of sectors we want 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.

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.

The raw file I/O functions 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 133 bytes, or maybe even more if you're dealing with some really fancy printing device. Anyway, some convenient method to read and write these text-lines to and from disk files would be a very useful thing for text processing applications. Ideally we'd like to be able to call a single function, passing to it some kind of file descriptor and a pointer to a

text-line, and let the function write the text-line into the file so that it immediately follows the last line written to that file. Also, to prevent a time-consuming disk access every time a line is written, it would be nice to have our function collect up a bunch of lines and toss 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 some disk file into a given place in memory; here, also, it would greatly improve performance if an invisible buffer was managed by the text-line-grabbing function so that disk activity is minimized. The functions described here are, in fact, "fputs" and "fgets" from the library: two of the "buffered I/O" functions.

The spotlight in the world of buffered I/O is a structure called, amazingly, an "I/O buffer". Within this structure is a large, even-sector sized character array within which the data being transferred is stored, and several assorted pointers and descriptors to keep track of "what's happening" in the data array portion of the buffer. There's a file descriptor to identify the file in raw I/O operations, there's a pointer into the data array to tell where the next byte shall be read from or written to, and there's 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. (1)

Buffered I/O functions use pointers to I/O buffers just as the raw functions use file descriptors. There are six functions that perform all actual buffered I/O for single bytes of data; the other buffered I/O functions (such as "fputs" and "fgets") do their stuff in terms of the six "backbone" functions.

For reading files we have "fopen", "getc", and "fclose". "Fopen" is called to associate an existing input file with a user-provided I/O buffer area by initializing all the variables in that buffer. "Getc" grabs a byte from the buffer, first refilling the data array from disk whenever the array is found to be empty, and returns a special value (EOF) when the end of the file is reached. "Fclose" closes the file associated with an I/O buffer.

For writing files there are "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 data. The data is written to the buffer via calls to "putc", one byte at a time. When all the data has been "putc"-ed, "fflush" is called to dump out the contents of the not-yet-full I/O buffer to the disk file. Finally, "fclose" wraps things up by closing the associated file.

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".

Let's look at a simple first example. The following program prints a given text file out on the console, with line numbers generated on the left margin:

---

1. The devious user may wonder why there is space taken for a byte counter, when the data pointer could just as well be compared to the last array address to detect a full/empty buffer. Actually, it ends up being more efficient with the counter, because the code required to compare two addresses is usually bulkier than the code required to decrement a counter and test for zero.

```
/*
        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 within the BDSCIO.H header file, tells how many bytes an I/O buffer must contain; this value will vary with the number of sectors desired for data buffering. See BDSCIO.H for instructions on how to customize the buffered I/O mechanism for a different buffer size (the default is eight sectors).

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' 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 cycle of reading and printing lines 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:

```
#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);
        fflush(obuf);
        fclose(obuf);
        fclose(ibuf);
}
```

This time there are two buffered I/O streams to be dealt with: the input file and the output file. The first thing to do is check for the correct number of arguments (in this case, two: 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 (none that I use) 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. Should "getc" ever return 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 that 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 for 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 that says 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.

Once having determined 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. Next, "fflush" is called to flush the output buffer. This must always be done before closing a buffered output file, to make sure that all characters sent to "putc" since that last time the buffer filled up get written to disk. Finally, "fclose" is used to close the input and output files.

For more examples of the usage of buffered I/O, see CONVERT.C, CCOT.C, TABIFY.C and TELNET.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.

BDS C Console I/O: Some Tricks and Clarifications

Leor Zolman
BD Software
Cambridge, Massachussetts

In this document I will attempt to remove some of the mystery behind the CP/M console I/O mechanisms available to BDS C users. When the major documentation for BDS C (i.e. the User's Guide) was being prepared, I had mistakenly assumed that users would automatically realize how the "bdos" and "bios" library functions could be used to perform all CP/M and BIOS functions, especially direct console I/O (by which the system console device may be operated without the frustrating unsolicited interception of certain special characters by the operating system.) In fact, the use of the "bios" function for such purposes might only be obvious to experienced CP/M users, and then only to those having assembly language programming experience with the nitty-gritty characteristics of the CP/M console interface. Let's take a look at what really happens during console I/O...

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; they are named `CONST' (check console status), `CONIN' (wait for and read a character FROM the console), and `CONOUT' (send a character TO the console). The way to get at these subroutines when you're writing on the assembly language level is rather convoluted, but the BDS C library provides the `bios' function to make it easy to access the BIOS subroutines from C programs. To check the console status directly, you use the subexpression `bios(2)', which returns a non-zero value when a console character is available, or zero otherwise. To actually get the character after `bios(2)' indicates one is ready, or to wait until a character is ready and then get it, use `bios(3)'. To directly write a character `c' to the console, you'd say `bios(4,c)', but note that the BIOS doesn't know anything about C's convention of using a single '\n' (newline) character to represent a logical carriage-return/linefeed combination. The call `bios(4,'\n')' will cause ONLY a single linefeed (ASCII 0x0A) character to be printed on the console.

Making sure that all console I/O is eventually performed by way of these three BIOS subroutines is the ONLY way to both keep CP/M from intercepting some of your typing and insure the portability of programs between different CP/M systems. (1)

The BDOS (Basic Disk Operating System) operations are the next higher level (above the BIOS) on which console I/O may be performed. 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, and this is where most of the confusion arises. 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 essentially the same 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

---

1. Even so there's no way to know what kind of terminal is being used--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.

Console Status" to determine if there is a character available from the console input (BDOS function 11). 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 a control-S character present on the console input during a console output call, then it will sit there and wait for another character to be typed on the console, and gobble it up, before returning from the original console output call. This may be fine if you want to be able to stop and start a long printout without having to code that feature into your C program, 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 what it does reveals some interesting facts: since it must be able to detect control-S on the console input, the BDOS must read the console whenever it sees that a character has been typed. If the character ends up not being a control-S (or some other special character that might require instant processing), then that character must be saved somewhere internally to the BDOS so that the next call to `Console Input' returns it as if nothing happened. Also, the BDOS must make sure that any subsequent calls made by the user to `Get Console Status' (before any are made to `Console Input') indicate that a character is available. This leads to a condition in which a BDOS call might say that a character is available, but the corresponding BIOS call would NOT, since, physically, the character has already been gobbled up by the BDOS during a prior interaction with the BIOS.

If this all sounds confusing, bear in mind that it took me several long months of playing with CP/M and early versions of the compiler before even I understood what the hell was going on in there. My versions of `getchar' and `putchar' are designed for use in an environment where the user does NOT need total direct control over the console; given that the BDOS would do some nice things for us like control-S processing, I figured that I might as well throw in some more useful features such as automatic conversion of the '\n' character to a CR-LF combination on output, automatic abortion of the program whenever 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's would no longer stop the program. Much later it became evident that neither `putchar' nor `putch' suffice when CP/M must be prevented from ever even sampling the physical console input. At this 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 it. 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. If you include these in your C source program, then the linker will use them instead of the standard library versions of the similarly named functions--provided that some direct reference to them is made before the default library file (DEFF2.CRL) is scanned. Usually, in programs where such functions are necessary, there will be many explicit calls to `getchar' and `putchar' to insure that the library versions aren't accidentally linked. A good example of a case where trouble might occur is when the entire program consists of, say, a single `printf' call followed by a custom version of `putchar'. Since the linker won't know that `putchar' is needed until after `printf' is loaded from the library, the custom version of `putchar' will be ignored and the old (wrong) version will be picked up from the DEFF2.CRL library file. The way to avoid such a problem is to insert, somewhere in the source file, explicit calls to any functions that are a) NOT explicitly called otherwise, and b) named the same as some library function. This isn't an expecially neat solution, but it gets the job done.

OK, with that out of the way, let's consider some more sophisticated games that can be played with customized versions of the console I/O functions. For starters, how about a set that performs conversions just like the library versions, detects control-C, and throws away any characters typed during output (except control-C, which causes a reboot)? No problem. What's needed is automatic conversion of '\n' to CR-LF on output; conversion CR to '\n' and ^Z to -1 on input with automatic echoing; and re-booting on control-C during both input and output.

```
/*
        Vanilla console I/O functions without going through BDOS:
        (`kbhit' would be the same as the above 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 ^C, reboot  */
        if (c == CPMEOF) return -1;             /* turn ^Z into -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 ^C typed, */
                bios(0);                        /* then reboot  */
}                               /* else ignore the input completely */
```

Now, if you wanted to have 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, you're probably better off going back to the original library versions of `getchar' and `putchar', which let the BDOS handle all that grungy stuff.

Incidentally, CP/M version 2.x has a new BDOS function which supposedly makes it easier to perform some of the direct console I/O operations that required the BIOS calls for CP/M 1.4. While this might be useful for people having CP/M 2.x, it would render any software developed using the new BDOS feature autistic when run on CP/M 1.4 systems. Please keep that in mind if you ever write any software on your 2.x system for use on other (perhaps non-2.x) systems.

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 evident in the sample program RALLY.C; the case there is that 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 the X-ON/X-OFF protocol 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 BDSCIO.H header file to define the unique port numbers, bit positions and polarities of the I/O hardware controlling his console. It would have been SO much easier if the BIOS contained just one more itty bitty subroutine to test console output status...but NoooooOOOOOOOoooooo, they had to leave that one OUT so we have to KLUDGE it...

Sorry. I get carried away sometimes. Oh well...I hope this has helped to demystify some of the obscure behavior sometimes evident during console I/O operations. For the low-down on how the library versions of `getchar', `putchar', etc. really work, see their source listings in DEFF2.ASM. 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.)

Good luck.

How To Avoid Warm-Boots After C Programs Finish Executing

Leor Zolman, 12/81

As most users of BDS C have probably noticed, C-generated COM files always perform a warm-boot when finished with their tasks. This is because the stack is usually placed in high memory just below the BDOS, wiping out part of the CCP (console command processor) during execution and requiring a warm-boot to bring back the CCP from the system tracks on disk. The following patches to the C.CCC run-time package file provide a way to generate COM files that do NOT perform a warm boot after execution, but instead return directly to a non-clobbered CCP. The price of avoiding a warm-boot is that there is less memory space available during execution (3000 bytes less by default); the advantage is that there is no waiting for the disk to seek and load the CCP every time the program is finished, improving overall performance and preserving the nerves of impatient hackers.

The procedure for generating non-booting programs is as follows:

1. Make a copy of your normal version of C.CCC (the run-time package binary image) under some other name.

2. Use DDT or SID to change your C.CCC file according to the patches listed below, and keep this new version of C.CCC for CLINK to use when linking your non-booting programs.

3. Compile and link your programs normally, but do NOT use the "-t" CLINK option; it won't work correctly for non-booting programs.

4a. After linkage is complete, use DDT or SID to change the first four bytes of the resulting COM file as follows:

```
100: 21          (was 2A)
101: 00          (was 06)
102: 00          (was 00 or 42)
103: 39          (was F9)
```

This MUST be done even if you've already changed some of these bytes in step 2, because CLINK itself sets the first 4 bytes of the COM file it generates to instructions that don't work in the non-booting variation. So, this step changes them back to what they need to be for all this to work.

4b. (optional): If you REALLY need to put the run-time stack someplace special, patch in the following sequence at location 107h (or 4307h for modified systems) after making the mainline patches described above:

```
107: 31                          (was CD)
108: <stack addr, low byte>      (was 34)
109: <stack addr, hi byte>       (was 01 or 43)
10A: 00                          (was F9)
```

Once this patch is made to C.CCC, it will remain in effect throughout later linkages, but the modification in step 4A must be made after each linkage.

5. The COM file should now be ready to execute. Try a simple one-line "printf" program the first time to test out the C.CCC patches; if working correctly, the output line should be followed immediately by a return to the system ("A>" should be printed)

without ANY disk activity having occurred. If anything else happens, re-check your patches. Remember that step 4 must be done after EVERY linkage.

6. Remember to restore the original C.CCC file when generating programs that need the extra stack space and/or need a warm-boot performed after execution.

Here are the C.CCC patches for non-booting COM files:

```
********* Changes to C.CCC for a non-warm-booting version ************
* (Some of the values in the "NORMAL (OLD)" coluumn may be different *
* from those shown if you've reassembled C.CCC on your own earlier)  *
**********************************************************************

-------------------------------------------------------------------
ADDR    NOBOOT  mnemonic        comments                NORMAL (OLD)
----    ------  --------        --------                ------
0100:   21      lxi h,0         ;get system SP into HL  31
 .               .
 .               .
 .               .
0103:   39      dad sp                                  00
0104:   22      shld spsave     ;save until exit        00
0105:   79                                              00
0106:   05 or 47                ;(47h for modified CP/M) 00
0107:   CD      call sppatch    ;compute new SP value   00
0108:   34                                              00
0109:   01 or 43                ;(43h for modified CP/M) 00
010A:   F9      sphl            ;place into SP reg      00
 .               .
 .               .
 .               .
        retpatch:
012F:   2A      lhld spsave     ;this is a patch from   C3
0130:   79                      ;the "vexit" routine,   FB
0131:   05 or 47                ;to restore system SP... 0C
0132:   F9      sphl                                    CD
0133:   C9      ret             ;...and return to CCP   96

        sppatch:
0134:   2A      lhld ram+6      ;get bdos pointer       0D
0135:   06                                              FE
0136:   00 or 42                ;(42h for modified CP/M) 38
0137:   11      lxi d,-3000     ;offset to bypass       CA
0138:   48                      ;the CCP                7B
0139:   F4                                              0C
013A:   19      dad d           ;leave new SP value in HL E6
013B:   C9      ret             ;in HL and return       08
013C:   00                                              C4
013D:   00                                              82
013E:   00                                              11
 .               .
 .               .
 .               .
0443:   C3      jmp retpatch    ;ready to exit...now reset C3
0444:   2F                      ; SP and return to CCP  00
0445:   01 or 43                ;(43h for modifed CP/M) 00
-------------------------------------------------------------------
```

Documentation for use with BDS Telnet v2.1

Leo Kenen
172 Churchills Lane
Milton, Mass. 02186
2/1/80

Setting up the machine:

To use the TELNET program effectively it is necessary for the hardware of your system to be properly configured. The current version will work with any modem which is connected to the microcomputer via a status driven port. This includes S-100 modems such as the PMMI or the D.C. Hayes, even though many of the neat features of these modems can not be used with this release.

On most systems the modem will be connected to the computer via a standard serial port and will run at 30cps (300 baud). A suitable cable must be made to connect the modem to the computer. This is usually a simple cable having one DB-25 (25 pin) connector at each end. The connectors may be either male or female depending on the requirements of your hardware. The standard wiring procedure is to connect pin 2 of one connector to pin 3 of the other (this goes both ways) and to put jumpers on each of the DB-25's. These jumpers should be between pins 4 and 5, and another jumper connecting pins 6,8 and 20.

Once the hardware is set up, it is then necessary to alter the **#define** statements in the TELNET.C source file to fit your configuration. When all the necessary changes have been made to the program, you are ready to compile it and test it out.

Initial test:

Turn on the modem and set it to HALF duplex (or better, TEST mode). Run the TELNET program (after its been compiled and linked) by typing TELNET. The program will then ask you if you expect an echo from the other computer or from the modem. Your reply should be 'y', since in this test we are hoping for an echo. Now type some keys on the console and see if they are displayed on the screen. If they are, then you have a working copy of TELNET. If nothing happens, there must be a problem with either the hardware or the software. If your modem has a test mode you should hear "blips" from the modem when keys are typed. If you do not, try reversing the wires on pins 2 and 3 of one of the DB-25 connectors. If the hardware looks good, check (and *double* check) the **#defines** in the program to be sure that they are correct for your system.

Communication Mode:

As soon as the program comes up you are in communication mode. In this mode anything that you type will be sent to the modem (except for the SPECIAL character, which causes TELNET to prompt for a special function code). Everything that arrives from the modem is also displayed on your screen. In this mode your computer is a sim-

ple dumb terminal. For most applications this is the most common mode of operation.

SPECIAL mode:

To enter SPECIAL mode from communication mode it is necessary to type single SPECIAL character (defined for your particular implementation within the **#define** section of the TELNET.C source.) This character should be one which you are not likely to need to type while in communication mode with another system. On most systems this character ends up being the NULL (0x00), ↑A (0x01) or ↑↑ (0x1f).

Typing an unknown command letter after hitting the SPECIAL character will display a list of legal commands on the screen. To send the special character to the other system (just in case it ever becomes necessary), just type it twice. The following commands (issued after typing the SPECIAL character) can be used to receive and transmit files and to perform many other useful functions.

Command Summary:

O Open an Output file for a data transfer. This function can be used to begin receiving programs or data from another computer or just keep a record of the things that you did while on line. When this command is given TELNET will ask several questions concerning the protocol that should be used during this transfer. The first thing that TELNET needs to know is the name of the file that should be used to store the data which is received. The filename you specify should be in the standard CP/M format:

Filename: foo.bar	*opens FOO.BAR on the current drive*
Filename: b:foo.bar	*opens FOO.BAR on B:*

When the file is opened, any old file with the same name will be lost. If this file can be opened, you will be asked if the transfer will involve TEXT (ascii data which is suitable for printing) or binary data. If your response is 'n' (to indicate binary) then the data received from the modem will not be displayed on the console until the transfer is completed. If you just want a record of the session's activity you must tell TELNET that text is going to be transfered (or you will not be able to see what you are doing).

If the transfer is going to be in checksum mode, then there must not be any echo coming from the other system or your modem. TELNET will believe it if you say there is no echo, but if there really *is* an echo then the chances of making a good transfer are nil.

If you do *not* choose checksum mode, then all incoming data will be buffered up in memory (except when *pausing*). Since the program cannot monitor incoming data while data is being dumped to disk, the normal procedure is to wait until you know there will not be any data coming in for a while (for instance, when you are talking to a host machine and it has just printed its prompt character) and then give the

dump command (D) to flush the buffer contents to disk. See also the D and C command descriptions.

D Dump (append) current contents of the collection buffer to the disk file (opened with the O command), leave the file open for more data, and clear the collection buffer. This function is useful if the file which is being transfered is larger than the buffer space. This is only needed if the transfer is *not* in checksum mode, since TELNET manages the buffer automatically when in checksum mode. After the buffer is dumped, collection will continue although any data that is sent while the disk is active will be lost forever.

C Close Output file. This function first forces an automatic dump of the memory buffer to the open file, after which the the file is closed. This command will also clear the memory buffer, permitting another file to be opened. *Close* is only needed if the transfer is *not* in checksum mode. An error in writing the file (such as running out of disk space) will result in the loss of the data.

T This command is the complement of the Open command, used for transmitting a file from your system out to the modem and beyond. It prompts for the name of the file to be transferred and for information regarding transfer protocol. These questions are analogous to those asked by the *Open* command described above. If the file can be opened, then it will be sent to the other computer using the protocol selected. If the transfer involves binary data, then a status message will appear on the console after each 128-byte sector is sent.

To *abort* or *pause*, use the A or P commands.

P Pause from file transfer. If a file has been opened (using the O command) in non-checksum mode, then this suspends the collection of incoming text in the memory buffer until the R command is issued to resume collection. If a file is being transmitted (in either checksum or non-checksum mode) then the transfer is suspended, to be continued when R is given. It is not good practice to *pause* during a checksummed transfer, but it is possible to recover provided: the transmitter pauses first, he waits for the receiver to pause before typing anything, the receiver resumes first, and then the transmitter resumes. Messy but at least feasible.

The main use of *pause*, though, should be during non-checksummed text file output.

R Resume from a *pause*.

A Abort current transfer. Use of this command will terminate any transfer which is currently in progress. If there is no transfer progress, a short message to that effect will be printed. If you are receiving data (via the O command) this command will also send out an ETX (↑C) to the transmitter to terminate that process also. While transmitting this command will send out enough ETX's to inform the receiver that the transfer has been terminated. If, however, the receiver is out of sync (probably

because of a slow terminal) when the transmitter aborts, then the receiver may have to terminate manually after seeing nothing happen for a long enough period.

V View the collection buffer. All contents of the collection buffer will be displayed on the console. Following the display of the data, the amount of free space left in the buffer will be announced. This is useful for verifying that a text file has been transferred properly.

K Kill (erase, delete, throw away, ZAP) contents of the text buffer.

Q Quit Telnet and return to CP/M. This function will dump any buffers that are being used for buffered I/O and then close the associated files. After all the housekeeping has been done the system will warm boot.

H Set Half/Full Duplex. Use this command to tell TELNET whether or not you are getting an echo from either the modem or from the other system. When this is set to *half duplex*, all data sent to the modem from your system will be simultaneously sent to your console output (except during binary data transfers). When in *full duplex*, it is assumed that the other system will echo what you type, so TELNET does not do it. There is no default for this command so TELNET will request the information from you at the start of a session.

7 Select protocol concerning the Parity bit. This function permits the parity bit to be preserved or to be masked out. In text files it is normal to mask out the MSB (ani 7fh). During a transfer this mode is set automatically.

N Select protocol regarding Nulls. This function is used to tell TELNET to either disregard nulls (for text) or to notice nulls (needed in binary and some other applications). When the system is noticing nulls, then they will be placed in the text buffer and saved when the buffer is dumped to disk. Ignoring nulls reduces the amount of storage necessary since nulls will not be placed into the buffer.

F Select linefeed protocol. Asks whether or not the linefeeds which follow carriage-returns in CP/M text files should be transmitted. Many remote systems would not appreciate those linefeeds.

L Enable/disable CP/M list device. If enabled, anything going to the console (except TELNET control messages) is also sent to the list device (usually a printer.) The printer's baud rate should be higher than the modem's.

SPECIAL Transmit the SPECIAL character to the modem.

CUG GUIDELINES

DISTRIBUTION

Please send check or money order with request. Fees are:

Media	domestic	foreign
8-inch diskette	$8.00	$12.00
5-1/4 inch diskette		
1-disk release	$8.00	$12.00
2-disk release	$12.00	$16.00
User's Manual	$15.00	$25.00

CONTRIBUTIONS

Please submit on hard-copy or 8-inch diskette. (If you haven't translation facilities contact us BEFORE you make the submission on some other media).

Include these files:

- CATALOG.DOC--a brief (two sentence) description of all source or data files.
- ENVIRON.DOC--a description of the environment in which the programs were developed (what version BDS, what machine, if applicable, what version of CPM). Particularly, document any changes you may have made in the standard libraries.
- READ.ME--(optional) anything else you'd like to tell the group or the reviewers.

COMPILER UPDATES & other licensed software

Members who have submitted a copy of the appropriate license with their membership form need only refer to it in their request. Others should include an 8-1/2 x 11" photocopy of the original diskette (very hazardous) or the envelope in which it was shipped. The product name, version number and serial number are the portion of the envelope we need to see.

Be certain to include your phone number and mailing address on all correspondence. Address to:

Robert Ward
BDS 'C' Users' Group
c/o Dedicated Micro Systems, Inc.
409 E. Kansas
Yates Center, Ks 66783

Membership Form

Please include a check or money-order for $10.00 and a copy of your license (see guideline sheet under compiler updates)

NAME_____ OCCUPATION_____

OFFICE/APT_____ HOME PHONE_____

STREET_____ BUSINESS PHONE_____

CITY_____ STATE_____ ZIP_____

SOFTWARE

Check each that applies, give specific information where applicable.

__1.4CPM __BASIC (intrp.) __FORTH __Other language
__2.x CPM __BASIC (compiled) __MACRO ASM _____
__MPM __PASCAL __OTHER ASM
__BDS C, serial_____ OTHER O.S._____

HARDWARE

MAKE_____ ___disk drives. MAKE_____

MODEL_____ SIZE____, DENSITY____, SIDES___
TRACKS_____

CPU_____ RAM____k

PRINTER_____

___Serial ports___Parallel ports __Modem__Other I/O_____

INTERESTS

Rank the following 1=very instersting to 4=worthless

__Music __Language Dev. __Games
__Graphics __Systems Dev. __Simulations
__Process Control __Record-Keeping __Artificial Intelligence
__Communications __Word Processing __Other_____
__Education __Design Aids

GOALS

Rank from 1 to 6 as you feel the group should tend to each (1 is highest priority, 6 the lowest)

__Compiler/Development tool development __Games
__Application Program development __Education
__Group Purchasing __Other
__Consumer warnings _____

PARTICIPATION

Indicate work you would be willing to perform for the group.

__write for newsletter __Develop networks
__document and edit disks __write/adapt games
for distribution __write/adapt ed. programs
__copy disks __disk format translation
__handle correspondence __write/adapt applications programs
relating to a speciality
what?_____

__Other_____

COMMENTS:

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